Electromagnetic Fault Injection On Two Microcontrollers: Methodology, Fault Model, Attack, and Countermeasures

by

Haohao Liao

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

Waterloo, Ontario, Canada, 2020

© Haohao Liao 2020

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner: Yunsi Fei
Professor, Dept. of E & CE,
Northeastern University

Supervisor(s): Catherine Gebotys
Professor, Dept. of E & CE,
University of Waterloo

Internal Member: Guang Gong
Professor, Dept. of E & CE,
University of Waterloo

Internal Member: Mark Aagaard
Associate Professor, Dept. of E & CE,
University of Waterloo

Internal-External Member: Alfred Menezes
Professor, Dept. of C & O
University of Waterloo

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

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

Abstract

Cryptographic algorithms are being applied to various kinds of embedded devices such as credit card, smart phone, etc. Those cryptographic algorithms are designed to be resistant to mathematical analysis, however, passive Side Channel Attack (SCA) was demonstrated to be a serious security concern for embedded systems. These attacks analyzed the relationship between the side channel leakages (such as the execution time or the power consumption) and the cryptographic operations in order to retrieve the secret information. Various countermeasures were proposed to thwart passive SCA by hiding this relationship.

Another different type of SCA, known as the active SCA is Fault Injection Attack (FIA). FIA can be divided into two phases. The first one is the fault injection phase where the attacker aims at injecting a fault to a target circuit with a specific timing and spatial accuracy. The second phase is the fault exploitation phase where the attacker exploits the induced fault and forms an attack. The major targets for the fault exploitation phase are the cryptographic algorithms and the application-sensitive processes. Over the last one and a half decades, FIA has attracted expanding research attention.

There are various techniques which could be used to conduct an FIA such as laser, Electromagnetic (EM) pulse, voltage/clock glitch, etc. EM FIA achieves a moderate spatial resolution and a high timing resolution. Moreover, since the EM pulse can pass through the package of the chip, the chip does not need to be fully decapsulated to run the attack. However, there remains a lack of understanding of the fault injected to the cryptographic devices and the countermeasures to protect them. Therefore, it is important to conduct in-depth research on EM FIA.

This dissertation concentrates on the study of EM FIA by analyzing the experimental results on two different devices, PIC16F687 and LPC1114. The PIC16F687 applies a two-stage pipeline with a Harvard structure. Faults injected to the PIC16F687 resulted in instruction replacement faults. After analysis of detailed experiments, two new Advanced Encryption Standard (AES)-128 attacks were proposed and empirically verified using a two-step attack approach. These new AES attacks were proposed with less computational complexity unlike previous Differential Fault Analysis (DFA) algorithms. Instruction specific countermeasures were designed and verified empirically for AES to prevent known attacks and provide fault tolerant protection.

The second target chip was the LPC1114, which utilizes an ARM Cortex-M0 core with a three-stage pipeline and a Von Neumann structure. Fault injection on multiple *LDR* instructions were analyzed indicating both address faults and data faults were found. Moreover, the induced faults were investigated with detailed timing analysis taking the

pipeline stall stage into consideration. Fault tolerant countermeasures were also proposed and verified empirically unlike previous fault tolerant countermeasures which were designed only for the instruction skip fault.

Based on empirical results, the charge-based fault model was proposed as a new fault model. It utilizes the critical charge concept from single event upset and takes the supply voltage and the clock frequency of the target microcontroller into consideration. Unlike previous research where researchers suggested that the EM pulse induced delay or perturbation to the chip, the new fault model has been empirically verified on both PIC16F687 and LPC1114 over several frequencies and supply voltages.

This research contributes to state of the art in EM FIA research field by providing further advances in how to inject the fault, how to analyze the fault, how to build an attack with the fault, and how to mitigate the fault. This research is important for improving resilience and countermeasures for FIAs to secure embedded microcontrollers.

Acknowledgements

Doing research work in University of Waterloo is definitely one of the best decisions I made. During the past four years, there are so many people who offered me support, help, guidance, and friendship. I would like to take this opportunity to give my sincere thanks to them.

First, I would like to express my sincerest gratitude to my supervisor Professor Catherine Gebotys. She always provides valuable suggestions for my research. She is always patient and she always holds a positive attitude. It is possible to fall into depression due to the failed experiments. Her endless encouragements at this time give me the confidence to continue on the research project. I truly appreciate the smiles she gave to me during every meeting. Without her help, this dissertation will never be possible.

I also would like to thank Professor Guang Gong, Professor Mark Aagaard, Professor Alfred Menezes, and Professor Yunsi Fei for serving as the committee of my thesis defense. Their valuable comments and precious time are highly appreciated.

I also want to thank Professor Douglas Stinson for serving as the committee for my background and proposal examination.

Additionally, I want to thank my colleagues who worked with me in the same lab. I am grateful for all the insightful discussions and the friendship with Mahmoud Khalafalla, Mustafa Faraj, Mahmoud Elmohr and Bahareh Ebrahimi.

Thanks also go to my friends in Waterloo. I am grateful for all the time I spent with Bo Yang, Dongxu Ma, Junling Li, Jianan Zhang, Weisen Shi, Huaqing Wu, Yuhui Lin, Dr. Wen Wu, Dr. Qiang Ye, Dr. Feng Tian, Ming Li, and so many others. I truly appreciate the friendship, support provided by them.

I greatly thank my beloved cousin Mei Huang for her love and encouragements. I also want to thank my other family members for their support during my life.

Last, I want to express my sincere thanks to my dearest girlfriend Ziyu Zhao for her love, patience and continuous support during my PhD study. Thanks for always being considerate with me.

Haohao Liao
January 1, 2020
Waterloo, Ontario, Canada

Dedication

This Ph.D dissertation is dedicated to my beloved parents, Huaqiang Liao and Jing Chen, and grandmothers Shiru Rao (†April 5, 2003) and Furong Zhang

Table of Contents

List of Tables xii

List of Figures xvi

List of Listings xx

Acronyms xxi

1 Introduction 1

1.1 Motivations . . . 1

1.1.1 Security Concerns for Embedded Devices . . . 2

1.1.2 EM FIA vs other FIAs . . . 3

1.2 Background . . . 3

1.2.1 Taxonomy of SCA . . . 4

1.2.2 Active SCA—FIA . . . 5

1.3 Summary and Overview of the Thesis . . . 7

2 Theory of the EM FIA and Previous Research 9

2.1 Theory of the EM FIA . . . 9

2.2 The Fault Injection . . . 13

2.2.1 Previous Research on FIA . . . 13

2.2.2 Fault Models . . . 17

2.3 The Fault Exploitation 21
2.3.1 Fault Exploitation on the Cryptographic Algorithms 21
2.3.2 The Application-sensitive process 27
2.4 Countermeasures 28
2.5 Summary 29

3 Experimental Setup and Terminology 31
3.1 Experimental setup 31
3.2 Terminology and Characterization of the Probe 34
3.3 Limitations of the Equipment and Assumptions 38
3.4 Summary 40

4 Experimental results on PIC16F687 41
4.1 Overview of the PIC16F687 41
4.2 Handheld Experiment on Chip #1 46
4.3 Identify the Best Location for Injecting the Fault on Chip #2 50
4.3.1 Effect of the V_p and the Probe Location 51
4.4 Case Study: Analysis of The Flt_inst 55
4.4.1 Test Program One 55
4.4.2 Test Program Two 56
4.4.3 Test Program Three 59
4.5 The $XORWF$ Instruction on Chip #2 61
4.5.1 AES Attack Targeting the $XORWF$ Instruction 62
4.6 The $DECFSZ$ Instruction on Chip #2 67
4.6.1 Bypass security checks attack 69
4.6.2 The AES Attack Targeting the $DECFSZ$ Instruction 70
4.7 Extend The AES-128 Attack To Chip #3 73
4.7.1 The AES-128 Attack Targeting the $XORWF$ 74

4.7.2 The AES-128 Attack Targeting the $DECFSZ$ 75
4.8 Countermeasure 77
4.8.1 Countermeasure for the $MOVWF$ Instruction 77
4.8.2 Countermeasure for the $MOVLW$ Instruction 78
4.8.3 Countermeasure for the $XORWF$ Instruction 79
4.8.4 Countermeasure for the $DECFSZ$ Instruction 81
4.8.5 Countermeasure to Protect the AES 83
4.9 Summary and Comparison 83
4.9.1 Comparison with Previous Research 85

5 Experimental results on LPC1114 88

5.1 Introduction to the LPC1114 88
5.2 Experiment on Chip #1 91
5.2.1 Handheld Experiment on Chip #1 91
5.2.2 Automated Platform Experiment on chip #1 95
5.3 Experiment on Chip #2 101
5.3.1 Experiments Targeting the $LDR\ PC-relative$ Instruction 101
5.3.2 Targeting the $LDR\ <Rt>$, $[<Rn>, \#<imme>]$ Instruction . 106
5.3.3 Targeting the $LDMIA\ <Rn>!, <registers>$ Instruction 109
5.4 HardFault 112
5.4.1 Complexity of the HardFault 114
5.4.2 HardFault on Chip #1 with $F_{clk} = 104.4\ MHz$, $V_{DD} = 3.3\ V$, $D_{t2p} = 274\ ns$. 115
5.5 Countermeasure 117
5.5.1 Duplication Countermeasure 117
5.5.2 Fault Tolerant Countermeasure 118
5.5.3 The HardFault Handler As a Countermeasure 119
5.6 Summary 120
5.6.1 Comparison with Previous Research 120

6 Charge-based Fault Model **123**

6.1 Charge-Based Fault Model 123

6.2 Empirical Verification on PIC16F687 125

6.3 Empirical Verification on LPC1114 128

6.4 Summary 130

7 Conclusions **131**

7.1 Summary, Discussion and Limitation 131

7.2 Contributions 133

7.3 Future Research Directions 134

References **136**

Appendices **147**

A Hardware Countermeasures for EM FIA **148**

B Additional Experiments on PIC16F687 **153**

B.1 Instruction/Macro description 153

B.2 z-distance Effect at Best Coordinate 154

B.3 Test Program for the $XORWF$ Instruction 155

B.4 Test Program for the $DECFSZ$ Instruction 157

B.5 Combined Countermeasure for both the $XORWF$ and $DECFSZ$ 158

C Unsuccessful Experiment Details Targeting the $LDR \ <Rt>, [<Rn>, \# <imme>]$ and $LDMIA \ <Rn>!, <registers>$ Instructions on Chip One of LPC1114 **161**

List of Tables

1.1 Some fault injection techniques [1] . . . 3

1.2 Some FIAs and the target cryptographic algorithms . . . 7

2.1 List of EM FIA research papers (“FS”= Front side, “BS” = Back side “-” = Not specified) . . . 16

2.2 Some fault models proposed in FIA . . . 17

2.3 Notations for AES . . . 22

3.1 Terminologies and the associated parameters used in the thesis . . . 35

4.1 Instruction description and the associated opcode . . . 45

4.2 Best probe position (xy) for largest number of faults injected using cumulative shmoo plot on right hand side of Figure 4.12 . . . 54

4.3 Possible faults injected on test program one . . . 56

4.4 Hamming distance between Tgt_inst and Flt_inst of test program one . . . 56

4.5 Fault injection result on test program two . . . 58

4.6 Hamming distance between Tgt_inst and Flt_inst of test program two . . . 58

4.7 Instructions defined with opcode bit[13:7] = 0000000 . . . 58

4.8 Reconstruct Table 4.6 with undefined instruction executed as NOP . . . 59

4.9 Fault injection result on test program three targeting $MOVLW$ . . . 60

4.10 Hamming distance between Tgt_inst and Flt_inst of test program three . . . 60

4.11 Statistical result of the fault injected on test program three targeting $MOVLW$ 61

4.12 Notations for AES targeting $XORWF$ instruction . . . 63

4.13 Fault injection result on AES-128 code targeting the $XORWF$ instruction 64

4.14 AES-128 attack result with the attack algorithm 1-a targeting $XORWF$. 66

4.15 AES-128 attack result with the attack algorithm 1-b targeting $XORWF$. 67

4.16 Experimental result of the bypass security check test . . . 70

4.17 AES-128 attack result with the attack algorithm 2 targeting $DECFSZ$ . . 73

4.18 AES-128 attack result with the attack algorithm 1-c targeting $XORWF$ on chip #3 . . . 75

4.19 AES-128 attack result with the attack algorithm 1-b targeting $XORWF$ on chip #3 . . . 75

4.20 AES-128 attack result with the attack algorithm 2 targeting $DECFSZ$ on chip #3 . . . 77

4.21 The instructions that had been tested with all the observed Flt_insts in different test programs . . . 84

5.1 Target instruction description and the associated encoding for LPC1114 . . 89

5.2 Simulation result for test code 1-c . . . 95

5.3 Summary of the fault injection result by using UART to read the faulty output with D_{t2p} set as 274, 284, 294, 304 ns . . . 97

5.4 Simulation result for test code 1-d, 1-e and 1-f . . . 98

5.5 Delay from trigger to the four cycles for $LDR\ R4/R5$ instruction . . . 99

5.6 Faulty values in R3 with EM pulse injected in cycle 28 . . . 102

5.7 Simulation result for test code 1-g and 1-h . . . 102

5.8 Summary of the faulty value for test program 1-g with different delay settings (correct value should be FEDCBA98) . . . 103

5.9 Summary of the faulty value for test program 1-h with different delay settings (correct value should be FEDCBA98) . . . 104

5.10 Hamming distance between the final faulty value and the value on the data bus after the execution stage . . . 105

5.11 Summary of the faulty value for test program 2-a with different delay settings (correct value should be 5A5A5A5A for both registers) . . . 108

5.12 Faulty value in R1 with test program three 110
5.13 Faulty value in R3 with test program three 110
5.14 Two groups of register information output from UART on injecting Hard-Fault to chip #2 115
5.15 Register information from simulation and received from UART after injecting the fault with EM pulse 115
5.16 Register information output by the HardFault handler after injecting the fault with the EM pulse (All values are in hexadecimal) 116
5.17 Simulation result for test code cm-1 118
5.18 Faulty value in R4 with duplication countermeasure 118
5.19 Instructions that have been tested on LPC1114 120
5.20 Research papers with FIA result on ARM 122

6.1 Experiment result for the charge-based fault model verification on PIC16F687, F_{clk} in MHz, delay in ns and V_{DD} in V 127
6.2 Experimental result for the charge-based fault model verification on LPC1114, F_{clk} in MHz, delay in ns and V_{DD} in V 129

A.1 Summary of countermeasures designed for EM FIA 148

B.1 Instruction description and the associated opcode 153
B.2 Faulty data and the associated Flt_inst (The original value at 0x20 is 0x55 before executing the $XORWF$ instruction) 155
B.3 Hamming distance between Tgt_inst and Flt_inst of test program four . . 156
B.4 Statistical result of the fault injected on $XORWF$ $0x20,\ F$ in test program four 156
B.5 The average and standard deviation of the occurrence for each Flt_inst . . 156
B.6 Faulty data and the associated Flt_inst for test program five 157
B.7 Hamming distance between Tgt_inst and Flt_inst of test program five . . . 157
B.8 Statistical result of the fault injected on $DECFSZ$ instruction in test program five 158

B.9 The average and standard deviation of the occurrence for each *Flt_inst* of *DECFSZ* in test program five 158

C.1 Simulation result for test code where a *LDMIA* instruction is the target . 162

C.2 Simulation result for test code which uses a register + immediate offset as the destination address 164

List of Figures

1.1 SCA model . . . 4
1.2 A typical PIN check program [2] . . . 7
2.1 EM probe tip . . . 10
2.2 Magnetic flux density generated by a coil of loop current [3] . . . 10
2.3 Sampling fault model [4] . . . 19
2.4 Difference between delay (or timing) fault and sampling fault [4] . . . 20
2.5 Probability of the faulty response [4] . . . 20
2.6 Faulty key byte propagation [5] . . . 24
2.7 Propagation of the faults in the last two rounds after $K_{10}^{0,0}$ is injected with a fault [5] . . . 25
2.8 Secure boot process [6] . . . 27
2.9 Modification of the external flash memory [6] . . . 28
3.1 Typical experimental setup used in EM FIA [7] . . . 32
3.2 (a): Handheld experiment setup. (b): Automated platform with a CNC machine as x-y-z stage . . . 32
3.3 Control flow of the automated FIA platform . . . 34
3.4 Main dialogue of the control software [8] . . . 35
3.5 Oscilloscope plots for BPS201 probe . . . 36
3.6 Oscilloscope plots for BPS202 probe . . . 37

4.1 From top to bottom: backside decapsulated chip, frontside decapsulated chip, cavity chip and original chip [9] 42
4.2 Backside view of the PIC16F687 die [9] 43
4.3 Clock/Instruction cycle [10] 43
4.4 Pipeline flow [10] 44
4.5 Voltage-Frequency graph [11] 44
4.6 Large probe tip from BPS201 over a front side cavity chip, chip #1 of PIC16F687 46
4.7 Program for handheld experiment 47
4.8 Timing analysis of the processor executing program 4.1 with $D_{t2p} = 274\ ns$, showing EM pulse occurring during instruction cycle 5 48
4.9 Partial memory dumps indicating (a):fault injected at address 0x2B and (b): fault injected at addresses 0x2F, 0x3D and 0x68 49
4.10 The probe tip protected by the cover over the target chip 50
4.11 Shmoo plot for number of fault been injected at each coordinate over 10 rounds for the experiment scan over the die of chip #2 under different V_p 52
4.12 Cumulative shmoo plots over all V_p settings. Left: scan over the die; Right: scan over the sensitive area 53
4.13 Shmoo plot for number of fault been injected at each coordinate over 10 rounds for the experiment scan over the sensitive area of the chip under different V_p 54
4.14 Timing diagram for test program one 56
4.15 Timing diagram for test program one 57
4.16 Instructions used in ARK_i of software implementation of AES, $m \in [0 : F]$ 62
4.17 Shmoo plot for number of faults injected with Flt_inst as (a): NOP, (b): $MOVWF$, (c): $SUBWF$ and (d): all injected fault. 65
4.18 Shmoo plot for the number of faults injected with Flt_inst as (a): NOP, (b): $MOVWF$, (c): $COMF$ and (d): $DECF$ (e): all injected fault. 72
4.19 Shmoo plot of chip #3 for number of faults injected with Flt_inst as (a): NOP, (b): $MOVWF$, (c): $SUBWF$ and (d): all injected fault. 74

4.20 Shmoo plot for number of faults injected with Flt_inst as (a): NOP, (b): $DECF$, (c): $INCF$ and (d): 0xE1 fault (e): all injected fault. 76

5.1 Original chip and backside decapsulated chip 90
5.2 Three stage pipeline flow for ARM Cortex-M0 [12] 91
5.3 BPS201 probe tip over the die of the LPC1114 92
5.4 Timing diagram for test program 1-a, 1-b and 1-c 93
5.5 Shmoo plot for the number of faults injected at each point over 10 rounds for the scan over the die of the LPC1114 under different V_ps 96
5.6 Timing diagrams of test program 5.1 where EM pulse is injected in cycle 29, 30, 31 and 32 98
5.7 Timing diagrams of test program 1-d and 1-g 101
5.8 Fault distribution illustration for test program 1-g 105
5.9 Data value stored in the memory with the LDR $<Rt>$, $[<Rn>$, $\# <imme>]$ instruction 107
5.10 Timing diagram for test program 2-a where N=26 107
5.11 Timing diagram for test program three 110
5.12 Fault distribution in R1 and R3 111
5.13 Timing diagram for code in listing 5.5 114

6.1 Charge based fault model 124
6.2 Example for empirical verification of the charge-based fault model 125
6.3 The nine clock edges in the prefetch cycle of the target instruction for PIC16F687 126
6.4 Test code 1-d used for empirical verification of the charge-based fault model 129

A.1 Glitch detector [13] 149
A.2 (a): Normal operation when no extra delay is induced (b): Timing diagram when a glitch is detected and alarm is on [13] 149
A.3 Structure of the Hogge phase detector [14] 150

A.4 Structure of the dual complementary flip flop [15] 151
A.5 The structure of the D-type flip flop countermeasure [16] 151
A.6 The full schematic of the D-type flip flop countermeasure [16] 151

B.1 Effect of the z-distance 154

C.1 Data value stored in the memory when targeting *LDR* immediate offset instruction 163

Listings

4.1 Program used for handheld experiment 48
4.2 Test program one 55
4.3 Test program two 56
4.4 Test program three 59
4.5 Revised ARK_i part for AES-128 63
4.6 AES-128 attack algorithm 1-a targeting $XORWF$ 65
4.7 Assembly code for AES-128 encryption subroutine [17] 68
4.8 The assembly program to mimic the code shown in 1.2 69
4.9 The revised encryption subroutine with a trigger signal 70
4.10 The AES-128 attack algorithm 2 targeting $DECFSZ$ 72
4.11 Countermeasure for MOVLW instruction 78
4.12 Countermeasure for protecting the XORWF instruction 79
4.13 Countermeasure for protecting the DECFSZ instruction 81
5.1 Test program one for handheld experiment targeting LPC1114 92
5.2 Test program 2-a targeting the LDR immediate offset instruction 106
5.3 Test program three targeting the $LDMIA$ instruction 109
5.4 Test program four targeting the CMP instruction 111
5.5 Test program five for HardFault handler 113
5.6 Test program CM-1 119
6.1 Program used for empirically verify the charge-based fault model on PIC16F687 125
6.2 Test program 1-d used to empirically verify the charge-based fault model on LPC1114 128
B.1 Test program four 155
B.2 Test program five 157
B.3 Countermeasure for protecting the AES from attacks proposed in Section 4.5.1 and 4.6.2 159
C.1 Test program two targeting the LDMIA instruction 162
C.2 Test program three targeting the LDR immediate offset instruction 163

Acronyms

AES Advanced Encryption Standard iv, ix, x, xii, xiii, xvii, xx, 1, 6–8, 12–14, 16–18, 21–24, 30, 38, 41, 50, 61–64, 66–70, 73–75, 77, 79, 81, 83–86, 120, 133, 134, 148, 152, 158–160

AHB Advanced High-performance Bus 89, 100, 104, 105, 108, 116, 120, 128

BPS Burst Power Station xvi–xviii, 31, 33, 36–40, 46, 47, 50, 61, 84, 88, 92, 94, 95, 105, 126, 127, 129

CNC Computer Numerical Control xvi, 32, 33, 38, 49, 51, 91, 95

CRT Chinese remainder theorem 6, 7, 14, 16, 86

DEMA Differential Electromagnetic Analysis 4

DES Data Encryption Standard 3, 22

DFA Differential Fault Analysis 1, 6, 7, 14, 16, 21–23, 84, 86, 120, 135

DPA Differential Power Analysis 4, 29

ECC Elliptic Curve Cryptography 6, 7

EEPROM Electrically Erasable Programmable Read Only Memory 41, 42, 45, 47, 49, 55, 60

EM Electromagnetic iv, v, viii, xi–xiv, xvi–xviii, 2–5, 7–13, 15–19, 21, 29–35, 38, 40, 42, 43, 47, 48, 50, 51, 53, 55, 57–60, 63, 66–70, 73, 75–77, 83–86, 88, 89, 91, 92, 97, 98, 100–102, 104–106, 108, 113–117, 121–124, 126, 127, 129–135, 148–150, 152, 154, 156, 158, 159, 162

FIA Fault Injection Attack iv, v, viii, xi, xii, xiv, xvi, 1–10, 12–18, 21, 22, 28–34, 40, 42, 46, 48, 50, 69, 83–85, 88, 89, 91, 117, 119, 121–123, 127, 129–135, 148–150, 152

FPGA Field Programmable Gate Array 12, 16, 18, 21, 135, 148, 150, 152

ISA Instruction Set Architecture 132, 135

NVIC Nested Vectored Interrupt Controller 89

PC Program Counter 28, 44, 89, 91, 95, 106, 114–116, 120–122, 162

PLL Phase Lock Loop 148–150

PSR Program Status Register 115, 116

RSA Rivest Shamir Adleman 6, 7, 14, 16, 86, 135

SCA Side Channel Attack iv, viii, xvi, 1, 3–6, 29, 133, 134

SEMA Simple Electromagnetic Analysis 4

SHA Secure Hash Algorithm 6, 7

SPA Simple Power Analysis 4

SRAM Static Random Access Memory 41, 42, 45–48, 56, 57, 59, 107, 108, 163

UART Universal Asynchronous Receiver/Transmiter xiii, xiv, 8, 90, 94, 95, 97, 112–115, 117, 128, 162, 163

Chapter 1

Introduction

Secure cryptographic algorithms are not secure from physical attacks if they are implemented incorrectly. In the past 15 years, Side Channel Attack (SCA) was one of the major security concerns in embedded devices [18]. Different from traditional cryptanalysis, in SCA, attackers either passively record and analyze the unintentional physical leakage from the cryptographic module, or actively inject a malicious faulty data to change the normal operation of the cryptographic module to reveal the secret information. This chapter introduces the motivation of the research along with some necessary background information. Finally, the outline of this thesis is provided.

1.1 Motivations

Embedded devices have been applied to a wide spectrum of applications and they play an important role in our daily life. With the widespread use of the embedded devices, handling sensitive data and interconnecting within the wireless infrastructure, security problems have become a growing concern. After the first announcement of passive SCA [19], many countermeasures have been proposed and applied such as dual rail [20], masking [21], adding random delays [22], etc. However, these countermeasures may not provide sufficient protection against active attacks such as the Fault Injection Attack (FIA) .

FIA has been applied to many cryptographic algorithms. Bao et al. proposed an FIA against the public key cryptosystem [23]. Even Advanced Encryption Standard (AES) was vulnerable to FIA by using the Differential Fault Analysis (DFA) [24]. FIA could also be applied to bypass the security checks such as checking the correctness of the PIN or the

secure boot [25]. These concerns along with limited research in this area motivate us to conduct research into FIA, specifically the Electromagnetic (EM) FIA. Few countermeasures have been proposed for FIA such as error detection code, instruction duplication, etc [26]. These countermeasures will be further described in Chapter 2.

In this section, we first provide a brief introduction to the security concerns for the embedded devices and then talk about the reasons for choosing EM FIA as the attack technique.

1.1.1 Security Concerns for Embedded Devices

Unlike the standard personal computer which performs general purpose tasks, the embedded devices are usually specifically designed with fixed tasks [27]. The study of the security requirement of embedded devices has a long history. However, it is still a hard process to determine the security requirement for an embedded device. Usually, researchers specify the security requirement based on known attack models while new threats are unforeseen [28]. Additionally, the security capabilities must be defined before the development. Some highly constrained embedded systems may not be able to incorporate the added area or energy cost necessary for incorporating security. For example, the RFID tags usually have limited hardware resources [29]. Nevertheless, the embedded device may suffer from a high risk of attack if it has insufficient security [30]. There are various attacks which may be launched based on different kinds of embedded devices and attack objectives. Typically, these attack could be [31]:

- Unauthorized access to the asset. The attacker could steal the personal information like the password or some other digital contents without authorization.
- Communication with unauthorized devices. The attacker could copy sensitive information to his own unauthorized hard drive or USB drive.
- Execute unauthorized code or firmware. Malicious software could be capable of executing unauthorized applications.
- Clone the device. The attacker could steal some secret codes or the IP cores from the device and then clone it.

1.1.2 EM FIA vs other FIAs

There are numerous approaches for conducting an FIA. Table 1.1 shows five widely used techniques for FIA. A typical successful FIA usually requires the attacker to inject a fault at a specific location with a precise timing [32]. For example, Jeong et al. proposed an attack which was based on the assumption that the fault could be injected in a round counter register so that the number of steps needed for the compression function in HMAC was reduced [33]. Thus, from Table 1.1, Laser beam could possibly be one of the best choices to produce a successful FIA since the required fault must only be injected to the round counter register where a high spatial accuracy is required. However, it requires access to the silicon since the laser cannot pass through the chip package and the setup of the experiment is usually more complex than other techniques. Additionally, the target device may be hard to obtain because it is from another country or it is of military purpose, or the target device uses a unique key for each single device. Under these circumstances, the attacker has to compromise the attack accuracy for ensuring the device under attack is not damaged. Hence, EM FIA might be a good attack technique since it has a lower possibility of damaging the chip. In the meanwhile, it provides a higher spatial resolution and timing resolution compared with voltage/clock glitch.

Table 1.1: Some fault injection techniques [1]

Technique	Accuracy[space]	Accuracy[Time]	Cost	Damage
Laser beam	High	High	High	Probably
Clock glitch	Low	High	Low	No
Voltage glitch	Low	Moderate	Low	No
Temperature	Low	Low	Low	Possibly
EM pulse	Moderate	High	Moderate	Possibly

1.2 Background

Figure 1.1 shows the security model of an SCA. The traditional cryptanalysis usually exploits the weakness or the usage of the cryptographic algorithms. For example, if the application uses Data Encryption Standard (DES) to encrypt the data, it could suffer from the brute force attack analyzing the input and output data in Figure 1.1 with a simple personal computer [1]. Different from the traditional attacks, in an SCA, the adversary tries

[1]Lecture notes in ECE 710 topic 21: Communication Security

to take advantage of unintentional side channel leakages shown in Figure 1.1 for breaking unprotected (or insufficiently protected) implementations of cryptographic algorithms, which include power consumption, EM emanation, timing analysis, etc. Additionally, the adversary could also use FIA (Fault injection in Figure 1.1) to actively alter the normal behavior of the cryptographic module and analyze the input and/or faulty output data. In the following sections, the taxonomy of the SCA will be provided. The definition of FIA is presented along with a basic introduction of the FIA usages.

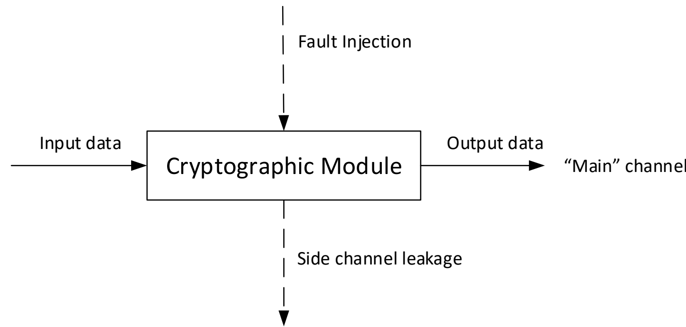


Figure 1.1: SCA model

1.2.1 Taxonomy of SCA

There are mainly two ways to categorize the SCA based on two different criteria. The first criterion is whether the adversary actively changes the behavior of the cryptographic module. Based on this criterion, we could categorize the attack into the passive attack or the active attack:

- **Passive attack**: In a passive attack, the adversary does not change the operation of the cryptographic module and only uses its leakage information to reveal the secret information. Power analysis (Simple Power Analysis (SPA) [34], Differential Power Analysis (DPA) [19]), and EM analysis (Differential Electromagnetic Analysis (DEMA) [35], Simple Electromagnetic Analysis (SEMA) [36]), acoustic SCA [37] are typical passive attacks.

- **Active attack**: In an active attack, the cryptographic module is manipulated and it does not operate within its specification. The adversary can then obtain the secret

information through analysis of the "faulty" outputs or behavior [38]. FIA is a typical active attack.

Another criterion is based on the level of decapsulation of the chip package required by the attack. SCA can be divided into three different categories based on this criterion:

- **Non-invasive Attack**: The attacker does not change or damage the package of the cryptographic module [39]. Consequently, the cryptographic module is running normally. For instance, the adversary may only use the power pin or the clock pin to conduct the attack.
- **Invasive Attack**: The attacker needs to decapsulate the chip and have physical access to the data transferred inside the chip by using a probe or other direct electrical contacts. For example, the microprobing attack [40] involves using direct physical contact with the die surface of the chip to read or manipulate the data in the device.
- **Semi-invasive Attack**: The attacker needs to make moderate changes to the package of the chip [41, 38]. Semi-invasive attack is a greater threat to cryptographic systems since it is almost as efficient as the invasive attack and could be of low cost at the same time [42]. EM FIA is a typical semi-invasive attack where the attacker does not need to have physical contact with the die surface of the chip but decapsulating part of the chip package may be required [43].

Since the focus of this thesis is on EM FIA, only the active FIA will be further detailed in the next section.

1.2.2 Active SCA—FIA

Before introducing FIA, it is beneficial to give a clear definition of the concept of *fault, error* and *failure* from the research area of the fault tolerant system. A *fault* is a physical defect or flaw which occurs in the software or hardware system [44]. An *error* is the result of a fault. If the error causes abnormal behavior of a system, it is called a *failure*. With these terminologies, FIA is defined below:

Definition. *FIA is an active SCA which involves using physical methods such as laser, EM pulse, voltage glitch, temperature, etc. to artificially inject a fault to the physical electronic device. The induced fault causes errors which consequently lead to security failures [45].*

FIA involves two phases to form a complete attack. In the fault injection phase, the attacker aims at injecting a fault at a suitable time during a security operation such as ciphering, authentication, etc. The injected fault is usually target specific [45]. The second phase is the fault exploitation phase which involves exploiting the injected fault by investigating the erroneous output or the failure [45]. The major targets for the fault exploitation are cryptographic algorithms and application-sensitive process where the data (which normally is the data in the control flow) cannot be modified [45]. With both targets, we could divide the FIA usages as follows:

1. **Attack the cryptographic algorithms:** The attacker aims at injecting a fault during the computation of the output of a cryptographic algorithm and then analyze the faulty output to retrieve the secret information. Various DFA algorithms were published for different cryptographic algorithms such as AES [46], Secure Hash Algorithm (SHA)-3 [47]. The attacker may also try use FIA to modify the round register in the cryptographic module to reduce/increase the round of operations and make the module output the ciphertext earlier/later. Some research papers applied FIA to break cryptographic algorithms such as HMAC [33] and AES [48, 49].

2. **Application-sensitive process**:

   (a) **Assist the passive SCA:** FIA could also be used to assist the passive SCA. One of the example is to disable the countermeasure for the passive SCA by using FIA. Yao et al. applied FIA to assist the passive SCA towards a software AES implementation embedded with masking countermeasure [50]. FIA forced the processor to disable the mask.

   (b) **Bypass security checks:** FIA could be applied to skip an instruction in a security check program. For example, Figure 1.2 presents a sample code for a pin check program. If the attacker could use FIA to skip the **reducePinTryCounter()**, he could unlimitedly input the pin [2].

Various techniques were applied to inject a fault as shown in Table 1.1. FIA was applied to various cryptographic algorithms including AES, SHA2, SHA3, Elliptic Curve Cryptography (ECC) and Chinese remainder theorem (CRT)-based Rivest Shamir Adleman (RSA). Therefore, FIA raises security concerns for a wide variety of cryptographic applications. Table 1.2 shows several FIAs on different cryptographic algorithms.

```
boolean checkPin(char* pin) {
  charArray correct_pin = {1,2,3,4}
  for (i=0; i<length(correct_pin); i++)  {
    if (pin[i] != correct_pin[]){
      reducePinTryCounter()
      return false
    }
  }
  return true
}
```

Figure 1.2: A typical PIN check program [2]

Table 1.2: Some FIAs and the target cryptographic algorithms

Year	Cryptographic algorithm	Practically implemented
1997 [51]	CRT-based RSA	No
2000 [52]	ECC	No
2003 [53]	AES	No
2010 [46]	AES	Yes
2013 [33]	HMAC	No
2015 [47]	SHA3	No

1.3 Summary and Overview of the Thesis

In this chapter, we presented the motivation and the necessary background information for this research. The objective of this research is to investigate the EM FIA efficiency targeting the embedded devices, develop a deep understanding of the mechanisms of this attack, and finally propose and verify potential countermeasures to thwart EM FIA. The whole thesis is divided into seven chapters.

Chapter 2 talks about the theory of EM FIA at the beginning. Then, we present a summary of the previous research on FIA with a focus on EM FIA. The delay fault model and the sampling fault model are presented and two DFAs targeting the symmetric key cipher are explored. Finally, we give a brief summary about the countermeasures proposed for FIA.

Chapter 3 introduces the experimental setup and some additional experimental measurements required before running the attack. A handheld setup and an automated platform are both presented with detail. The EM pulse generation equipment is presented with their specifications. We also elaborate on the measurement equipment to analyze the jitter and the equipment delays to ensure a correct timing analysis could be performed when the

fault is induced.

The experimental results on PIC16F687 are presented in Chapter 4. We first present the handheld setup, which was used to test the fault injectability of the chip with the EM pulse system. The handheld experiment indicated that the fault was injected only when the chip was overclocked. The experiments with the automated platform were described next including a scan to find out the best location to inject the fault and statistical results on the faulty instruction. An EM FIA methodology was proposed which would be further used to inject the fault to the LPC1114. An AES-128 attack was physically implemented by targeting two different instructions. We finally present the empirically verified countermeasures.

Chapter 5 discusses the empirical results on the LPC1114 which contains an ARM Cortex-M0. Using the attack methodology proposed on the PIC16F687, a handheld experiment was done first. The Universal Asynchronous Receiver/Transmiter (UART) was utilized to send out the faulty data to the desktop for off chip analysis. The induced faults on the LPC1114 consisted of the function fault and the Hardfault. The function fault could further be divided into the address fault and the data fault. The fault injected to different LDR instructions were analyzed. A detailed timing analysis was performed by considering the pipeline stall stage. A fault tolerant countermeasure was proposed and verified.

The new charge-based fault model is presented in Chapter 6. The charge-based fault model applies the critical charge concept from single event upset research. It takes the clock frequency and the supply voltage into consideration. The fault model was empirically verified on both PIC16F687 and LPC1114.

Chapter 7 gives a brief summary to the thesis. The contributions and limitations of this research are also discussed. Some possible future research directions are proposed.

Chapter 2

Theory of the EM FIA and Previous Research

In this chapter, we first present the theory of EM FIA in Section 2.1. Also, some papers which analyzed the physical effect of the EM pulse on the injecting fault are discussed. Based on the definition of FIA, the attack is divided into the fault injection phase and the fault exploitation phase. Therefore, the previous research on FIA are divided into two parts. The fault injection part is introduced in Section 2.2 where we present different fault injection techniques including both non-EM FIA and EM FIA. Another important part of the fault injection phase is setting up the fault model. The discussion about the fault model is presented in Section 2.2.2. The fault exploitation in previous research is introduced in Section 2.3 targeting both cryptographic algorithms and application-sensitive process. Finally, the previous researched countermeasures are summarized in Section 2.4.

2.1 Theory of the EM FIA

The basic theory of EM fault injection is to use an EM probe to generate a fast changing magnetic flux and alter the operation of the target device [54]. Figure 2.1 presents an example of an EM probe tip with a cylindrical ferrite core. The magnetic flux is generated by sending a current through the coils on the probe tip. The magnetic permeability of ferrite is greater than air. Thus, by surrounding the coil over a ferrite core, it increases the magnetic flux density generated by the same current. Additionally, the smaller diameter at the end of the ferrite core helps localize the magnetic field [55]. The magnetic flux density

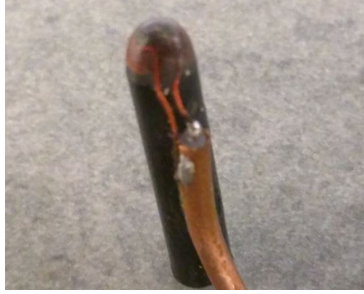


Figure 2.1: EM probe tip

generated by a coil of loop current is illustrated in Figure 2.2.

Assume the current through the loop is i, μ_0 is the permeability of the medium, the radius of the loop is a and the distance perpendicular to the loop plane is z, according to Biot Savart's law [56], the magnetic flux density $B(z)$ is:

$$B(z) = \frac{\mu_0}{2} \times \frac{a^2 i}{(a^2 + z^2)^{\frac{3}{2}}} \tag{2.1}$$

The magnetic flux cross a surface plane with an area of S is:

$$\Phi = B(z) S \cos\theta \tag{2.2}$$

where θ is the angle between the surface plane and the loop plane.

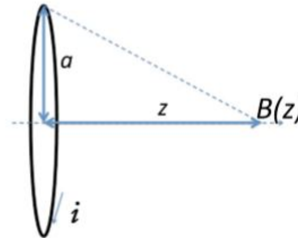


Figure 2.2: Magnetic flux density generated by a coil of loop current [3]

The target of FIA is an embedded device with a semiconductor chip running the cryptographic algorithm. The semiconductor consists of closed circuit loops. Based on Faraday's law of induction, the induced electromotive force is:

$$\varepsilon = -\frac{\mathrm{d}}{\mathrm{d}t}(\Phi) \tag{2.3}$$

Merging equation 2.1, 2.2 and 2.3, the induced electromotive force to the closed circuit in the semiconductor chip is:

$$\varepsilon = -\frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\mu_0}{2} \times \frac{a^2 i}{(a^2+z^2)^{\frac{3}{2}}} S \cos\theta\right) \tag{2.4}$$

The induced electromotive force generates a transient current in the closed circuit of the semiconductor chip which slows down or speeds up the circuit and causes a fault eventually. From equation 2.4, these following parameters may positively help induce the fault [54]:

- μ_0: A magnetic material with a higher magnetic permeability.
- θ: Place the probe tip to be perpendicular to the die surface which makes θ to be $0°$. Thus, $\cos\theta = 1$.
- z: A smaller distance between the probe tip and the die surface.
- v: A higher pulse voltage helps generate a larger loop current i and increases the magnetic flux.
- t_r: A shorter rise time of the voltage pulse generates a large magnetic flux within a shorter time which increases the induced electromotive force eventually.

Omarouayache et al. researched the EM probe design to inject faults [55]. With simulation, they found that the maximum magnetic flux density is along the perimeter of the current loop instead of at the center of the probe when $z = a/10$. Furthermore, they also found that a probe tip with a sharpened end is better to focus the generated magnetic field, which might be helpful in inducing a local fault. Another research paper further found that the magnetic field is mainly coupled with the chip's interconnections [57]. Their experiment utilized an on-chip voltage sensor which allowed a more accurate measurement.

Previous research also examined the EM pulse effect on the wire, chip and gate levels [58]. During the experiment of the wire level, they checked the voltage perturbation on a GPIO pin of the chip and found that the injected perturbation follows Faraday's law. From the chip level, the author found that there was no obvious perturbations on the power or ground pins when the attack point was far from these pins. They further hypothesized that the robust design of the power network could eliminate the noise or other coupling effects. However, other research papers pointed out that the effect of the injected EM pulse could be observed from the power trace of the target chip [59, 60].

Some previous research papers analyzed the parameters that might affect the EM induction to the chip such as the effect of the z-distance, pulse voltage, pulse width, etc. The analysis of these parameters were either done by using a second measurement probe to model the target chip or simply focusing on the fault occurrence. For example, Velegalati et al. applied a measurement probe to measure the induced voltage and found that with a larger distance, the induced voltage is smaller [32]. Similar measurement was done by Carlier [54]. Dehbaoui et al. analyzed the fault occurrence with regard to the pulse voltage [59]. Pulse width was analyzed by Moro et al. and it was suggested that the stress applied to the circuit is reduced with a longer pulse width [7]. This could be explained with the Faraday's law of induction if the pulse width is connected to the pulse rise time. However, the relationship between the pulse width and pulse rise time was not stated in the paper. The pulse width was also mentioned in other research papers but it was simply chosen to be smaller or equal to the clock period [61, 4].

However, some other parameters such as the shape of the probe, the diameter of the probe were not fully investigated. For example, Velegalati et al. found that the probe with a larger diameter may shut down the system running on the chip while the probe with a smaller diameter could inject graphics errors [32]. However, they failed to induce faults to their additive loop program running on the chip and they did not provide an explanation. Ordas et al. analyzed the effect of the probe shape [4]. The "sharp", "flat", and "cresent" probes were used in their experiments. They applied the latter two probes to induce fault to a Xilinx Field Programmable Gate Array (FPGA) with the AES running on it. With the "flat" probe, both bad ciphering fault and the "no-response" fault were observed while with the "cresent" probe, only the bad ciphering fault was injected. However, they did not provide an explanation for this phenomenon possibly due to either the complexity of the EM coupling between the EM pulse and the circuit under attack or the lack of understanding of the details of the target chip. Ordas et al. also performed EM FIA with different pulse polarities. They found that the susceptibility of the chip was different with different pulse polarities.

In summary, the attacker did not know the details of the silicon or the RTL design of the chip [61, 49, 49, 7, 32, 4] in most previous research. The parameters empirically analyzed and explained by the researchers were the pulse voltage, pulse injection time, and the rising time of the pulse. In our research work, the details of the target chips were also unknown. Moreover, we did not have control over the rising time of the pulse due to the limitation of the equipment. Also, the pulse polarity had minimal impact on our experimental results. Hence, we only analyzed the effect of the z-distance, pulse voltage, pulse injection time (for timing analysis), and the position of the probe as discussed in details in Chapter 4 and 5.

2.2 The Fault Injection

In this section, the previous research on fault injection ability and fault models will be discussed. First, we introduce different techniques that could be used to inject the fault in Section 2.2.1. Then, we present detailed investigation on previous fault models, especially the fault model proposed with EM FIA in Section 2.2.2 with a in-depth analysis of how each fault model was derived from the experimental result.

The main purpose of this section is to give introduction to different fault injection techniques and fault models presented in previous research. There are some research papers which utilized similar target chips or provide more details as we did in our experiments. These research papers will be discussed in Section 4.9.1 and 5.6.1 as a comparison with our experimental results.

2.2.1 Previous Research on FIA

As noted previously in Section 1.1.2, different physical techniques could be applied to perform FIA. In this section, we present some research papers which utilized these techniques to conduct FIA. The non-EM FIA techniques including power underfeeding, clock/power glitch, laser etc. are introduced first from the perspective of cost, experimental setup, difficulty, spatial/timing accuracy, level of invasion. Then, in Section 2.2.1.2, we present a table where some of the previous research papers in EM FIA are listed. More details of the associated paper will further be discussed in the subsequent sections.

2.2.1.1 Previous Research with Non-EM FIA Techniques

Barenghi et al. successfully applied the power underfeeding to retrieve the AES key from a self designed silicon die with 65 nm technology [62]. The AES was clocked at 1.3 MHz with an external clock source. The most important equipment used in their experiment was a power supply with a sufficient accuracy of 0.1 mV. The attacker gradually reduced the supply voltage with 0.1 mV per step starting from 0.45 V. With each supply voltage, they did ten thousand encryption operations and found the percentage of the faulty ciphertext increased gradually. It is worth noting that power underfeeding is different from power glitch. The power underfeeding reduces the supply voltage of the chip permanently while the supply voltage is only reduced for a short period of time (usually a single clock cycle) with power glitch [26].

Another two kinds of low cost FIA techniques are clock glitch and power glitch. These fault injection techniques require direct access to the clock pin or the power pin [26]. The power glitch may also require removing the external capacitors in the core power domain [6]. A trigger signal is usually output from the target chip to help insert the glitch at the correct time [63, 6]. For example, Korak et al. evaluated the clock glitch attack on the LPC1114 which embedded an ARM Cortex-M0 core and Atmel ATxmega 256 [63]. Both microcontrollers were running with a 24 MHz clock and a 3.3 V supply voltage. Then, at a specific pipeline stage of a target instruction, a clock glitch was inserted to fault the target instruction. They found that for the LPC1114, only when the supply voltage was reduced to 1.2 V (which was beyond the minimum nominal supply voltage 1.8 V), a reproducible fault could be injected. They also verified that without inserting a clock glitch, the LPC1114 worked correctly with a 1.2 V supply voltage. With the added trigger signal, they were able to perform analysis of the fault injected to different pipeline stages.

The above three attack techniques offer a low probability of damaging the device. Also, they do not require the decapsulation of the chip package. Therefore, they are typically classified as non-invasive attacks. One popular example of invasive attack is the laser FIA. Compared with other non-invasive attacks, the laser FIA typically requires a fully decapsulated chip to ensure the silicon die is optically exposed [64]. The laser FIA has a high spatial and timing resolution which allows the attacker to induce even a single bit fault [65]. This high precision FIA technique was applied in several research papers where the attack changed the normal memory operation [66], retrieved the private key of RSA [60] and revealed the AES key [67, 68]. However, the experimental setup is complex and requires significant technical skills [26]. The experiment also involves a collaboration of different equipment such as the microscope, camera, motorized x-y-z stage, optical table, etc. [64, 9]. Moreover, the increasing number of metal layers in the chip limited the laser FIA from the front side [4].

Another class of FIA is the semi-invasive attack. These attacks may require partially decapsulate the chip. The forward or reverse body biasing injection proposed in [69, 70] is a typical semi-invasive attack. This attack method applies a transient bias to the substrate. This transient bias could be injected through a tungsten needle by direct contact with the substrate [70]. Tobich et al. also pointed out that this bias could be used to induce local faults. It is considered as a local power glitch fault injection method [71]. The backside of the package must be opened such that the attacker could have direct access to the silicon substrate. A weighing scale was utilized to ensure the electrical contact between the probe and the substrate [70]. The forward body biasing attack was applied to a CRT-based RSA [69] running on a secure microcontroller with hardware countermeasures embedded. The associated DFA algorithm for CRT-based RSA typically relies on injecting faulty data

[51]. Only 0.22% of the faults injected resulted in an exploitable faulty answer. Also, the hardware countermeasures on the microcontroller were disabled.

2.2.1.2 Previous Research on EM FIA

EM FIA is typically a semi-invasive attack since in general, one does not need to fully decapsulate the chip but may need to thin the package such that the distance between the silicon die and the probe is reduced [43]. There are two different kinds of EM FIAs. The first one is called the harmonic EM FIA and the second one is the transient EM FIA. The harmonic EM FIA utilizes continuous sinusoidal EM waves to inject the fault while the transient EM FIA applies the high energy EM pulses [72]. Poucheret et al. applied the harmonic EM FIA to change the frequency of a ring oscillator which was implemented on a 90 *nm* prototype chip [73]. Several EM injection campaigns were conducted over the target chip. The frequency of the ring oscillator was increased by roughly 10% of the nominal frequency independent of the probe location. Hence, they considered that EM FIA was global, which was unlike the local effect discovered from most transient EM FIA in previous research [7, 59, 4]. They further assumed that the global effect was possibly related to the EM wavelength. The transient EM FIA is the type of injection technique utilized in this dissertation. Unless otherwise specified, EM FIA in this dissertation always refers to the transient EM FIA.

EM FIA has a higher spatial resolution compared with clock/power glitch FIA due to its local property [49, 7]. This local property may lead to different faulty responses by moving the EM probe over the chip [7]. Table 2.1 presents some of the previous research on EM FIA. These research papers focus on different areas of EM FIA including generally testing the feasibility of injecting a fault using the EM pulse [32] (row 8), elaborating fault models [7] (row 2), [4] (row 12 and 13), running attacks [74] (row 5), [49] (row 6), [75] (row 14), etc. These papers will further be discussed in associated sections. The experimental setup of EM FIA is similar in previous research and it will be further detailed in Chapter 3 as a comparison with our experimental setup. The faulty response from the chip depends on the probe tip, the parameters of the pulse (to be further detailed in Table 3.1 in Chapter 3) and the DUT. Generally, the EM pulse causes data corruptions and these corruptions lead to different results with different DUTs. Another important fact of EM FIA is the complexity of analyzing the faulty response. Due to the lack of ability to observe the real time behavior of the chip from the circuit level when inducing the EM pulse, the faulty response is complex and the hypothesis may be impossible to be verified [76].

Table 2.1: List of EM FIA research papers ("FS"= Front side, "BS" = Back side "-" = Not specified)

DUT	Decap? /side?	Probe tip diameter mm	F_{clk} MHz	V_{DD} V	V_p V	Pulse width /rise time (ns)	Timing resolution	Distance per step	Program or instruction	Fault Model	Special Notes
Cortex-M3 [7]	-	1	56	-	170 to 190	10/2	0.2	200	Additive loop, single load	Delay fault	-
Xilinx FPGA [13]	Yes /FS	0.5, 3	100	1.2	75 to 200	10 to 100/-	-	400	AES	Delay fault	-
AVR [59]	Yes /FS	0.5	3.57	-	1 to 100	10 to 100/5	100	-	AES	Delay fault	Low jitter (≤ 50 ps) pulse system used
Xilinx FPGA [59]	No /FS		100				-	500			
AVR [74]	Yes/ FS	1	3.57	-	50	20/5	10	-	AES	Byte fault	AES key was retrieved using DFA algorithm from [35]
Cortex-M3 [49]	-	1	24	-	180	20/2	0.1	-	AES	Instruction skip	AES key was retrieved with an additional faulty round
Cortex-A9 [32]	No /FS	1.5, 1.8	200	-	120 to 176	10/-	-	-	Additive loop	System crash and graphics error	-
ATMega 163+ [54]	No /FS	1.5, 1.7, 3, 4	1	2.9	60 to 90	0 to 500/-	-	-	Additive loop	"glitched" output or reset	The EM system was made by Riscure [77].
Cortex-M3 [54]			4	-							
Cortex-A8 [2]	No /FS	1.5	1000	-	0 to 450	-/-	10	-	NOP, MOVE, Branch, Compare, Add loops	Usable fault and exceptions	The EM system was made by Riscure [77].
Xilinx FPGA [4]	- /-	0.05, 0.45, 0.3 to 0.75	-	-	-200 to 200	8/-	-	300	No specific program	Bit set/reset	Clock was off during the EM injection. Pulse polarity affected the fault type
			25, 50, 100	1.2	50 to 190			100, 200	AES	Sampling fault	-
8-bit microcontroller [60]	Yes /FS, BS	-	-	-	-	-/-	-	100	CRT-based RSA	-	A spark gap was utilized to generate the EM pulse
Cortex-M3 [75]	Yes /FS	0.5	84	-	-100 to 100	6/2	1	-	AES, Load multiple	Bit set/reset/flip	AES key was retrieved successfully
PIC16F687	Yes /FS, BS	0.5, 2	8, 40 to 61.5	4 to 5	50 to 500	-/3	10	12.7	AES and others	Charge-based fault model	These are the target chips used in this dissertation
Cortex-M0	Yes /BS	2	48, 100 to 104.4	2 to 3.3	130 to 180	-/13			Load, Compare program		

2.2.2 Fault Models

The definition of the fault model varies with different research areas. Particularly in FIA, a fault model defines the type of the fault the attacker could inject or want to inject [7]. Fault model is important in conducting an FIA. If a fault model is not characterized based on real experiments, the associated attack algorithms may fail to be implemented. For example, Blömer and Seifert [53] present an attack which assumes a single bit fault can be injected each time after the first key addition in AES, which may not be feasible. In this section, a summary of the fault models used in different research areas is presented. Then, we provide a detailed description about the fault models proposed particularly for EM FIA.

Various fault models were presented in the last decade. Table 2.2 shows the summary of some of these fault models. These fault models were classified based on different categories. For example, the bit flip and byte fault are based on the data size affected by the induced fault. Similarly, the transient, semi-transient and permanent fault are defined with regard to how long the fault remains within the circuit. The delay fault model and sampling fault model were proposed specifically for EM FIA and will be discussed in Section 2.2.2.1.

Table 2.2: Some fault models proposed in FIA

Fault model	Description
Bit flip [78], set/reset [53]	Target data is manipulated by flipping or setting/resetting a bit
Byte fault [78]	Random bits of a target byte are altered
Instruction skip/replace [78]	Target instruction is skipped, executed as *NOP* or replaced
Transient fault [45]	The fault is only induced during fault injection
Semi-permanent effect fault [45]	The fault continues for a period of time
Permanent fault/stuck at '0'/'1' fault [45]	The fault permanently exists after it is injected
Sampling fault [4]	Fault occurs at writing to a register
Delay fault [59]	The propagation delay is increased causing a setup time violation

The fault model concept is also used in other research areas. For example, the single event upset is the type of error caused by charged particles [79] and the soft error is similar to the transient fault, but refers to memory faults.

Fault simulation and modeling is also utilized in the design for testing of digital systems [80]. For example, the fault could be categorized as single fault versus multiple faults or function fault versus structural fault [80]. The structural fault could be used to model

the change of the interconnections between component while the function fault models the change of the functionality of the circuit.

2.2.2.1 Fault Model in EM FIA

It is necessary to study the previous fault models proposed with EM FIA. The previous fault models are also listed in Table 2.1 (sampling and delay faults in last 2 rows). These fault models will be further explained based on the experimental result with the associated research paper.

Velegalati et al. [32] described a simple experiment that demonstrated an Android system running on a Samsung chip was successfully influenced by the EM pulse (row 8 in Table 2.1). A simple counter program was targeted by using three different types of probe tips. However, none of the probe tips corrupted the operation of the counter. Two fault models were observed. The first one was a graphics error where the display function was not running correctly and the second one was a total system shutdown. They discovered that a wider magnetic field might influence a larger part of the chip which resulted in a system shutdown. Only the probe tip which had a smaller diameter caused the graphics error while the remaining larger probe tips shutdown the whole system completely. Velegalati et al. [32] demonstrated that EM FIA was practical. However, the system crash or graphics error may not be utilized to conduct an attack.

Dehbaoui et al. [59] proposed a delay fault model to explain the fault mechanisms in their experiments. In this paper, Dehbaoui et al. [59] conducted the EM fault injection experiment towards both software and hardware implementations of AES. The parameters of the associated experiments are listed in row 4 and 5 of Table 2.1.

AES was first implemented in a 350 nm AVR microcontroller which used a Harvard architecture. The result demonstrated that the fault occurrence increased with the increase of the pulse voltage. Additionally, they also investigated the assembly code and explained the fault was caused by an instruction skip or replacement by using the EM pulse to affect the executing flow of the program. Furthermore, since the behavior of the induced fault was very similar to the fault injected by using clock glitch where the clock frequency was temporarily boosted, they hypothesized that the induced fault caused a timing constraint violation which eventually resulted in the instruction skip effect. Faults were also successfully injected to the last round of AES running on a Xilinx FPGA with 100 MHz clock. Their results implied that the EM pulse had a localized effect. Moreover, by positioning the probe tip at a specific location which contained the critical path, the fault occurrence was higher than other positions. They further hypothesized that the induced fault caused

a transient decrease in the supply voltage due to the coupling between the EM pulse and the power ground network of the chip. Eventually, it increased the propagation delay and led to a timing constraint violation. The same delay fault model was also observed by Moro et al. in 2013 by injecting the fault to a 32-bit microcontroller with an ARM Cortex-M3 core [7] (row 2 of Table 2.1).

Ordas et al. [4] showed that, the delay fault model was not as realistic as the proposed sampling fault model listed in row 13 of Table 2.1. The sampling fault model is presented in Figure 2.3. Both the EM pulse voltage and the injection time affect the fault injection result. If the EM pulse voltage is greater than V_{High}, the EM pulse is able to induce a large delay during the data propagation. Hence, whenever the fault injection happens, a timing fault is injected marked as A while no fault can be injected if the EM pulse voltage is less than V_{Low}. A sampling fault is injected if the EM pulse voltage is within the range from V_{Low} to V_{High} and the injection time is within the susceptible window marked as (1) in Figure 2.3.

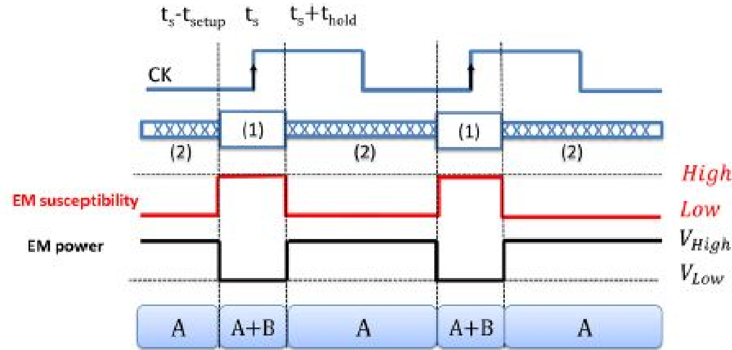


Figure 2.3: Sampling fault model [4]

The difference between the delay fault model (referred to as the timing fault in Figure 2.4 [59, 7]) and the sampling fault is illustrated in Figure 2.4. D_n denotes the time duration that the data needs to recover back to its normal state. For a delay fault, the attacker induces a delay during the data propagation. Therefore, as long as the fault injection time is before the next rising edge of the clock, fault can be successfully injected as shown in the upper part of Figure 2.4. No matter the EM pulse is induced at t_1, t_2 or t_3, fault occurs at the second rising edge because the induced delay causes a setup time violation. On the contrary, for a sampling fault, if the input voltage is within the range from V_{Low} to V_{High}, the EM pulse must be induced at t_1 or t_2 in order to inject the fault since the data has no sufficient time to recover back to its normal state.

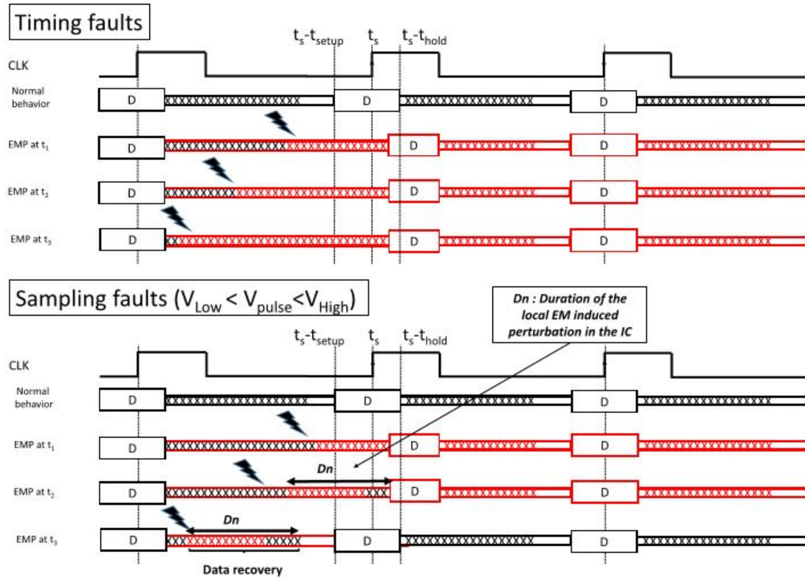


Figure 2.4: Difference between delay (or timing) fault and sampling fault [4]

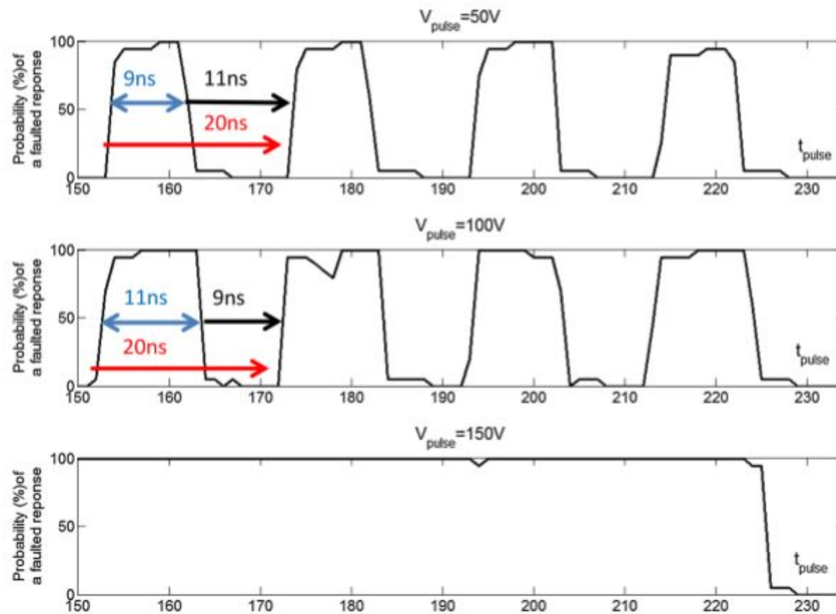


Figure 2.5: Probability of the faulty response [4]

Ordas et al. [4] further verified this fault model by conducting an experiment targeting a Xilinx FPGA running at 50 MHz. In Figure 2.5, the x-axis is the time when the EM pulse is induced and the y-axis is the probability that a faulty response happens. The experiment showed that V_{Low} was between 40 V and 50 V (since at 40 V, no fault was injected and at 50 V, fault could be injected). Additionally, V_{High} was below 150 V since faults can always be injected when the EM pulse voltage was set to 150 V. When the EM pulse was less than V_{High} but higher than V_{Low}, the probability of successfully injecting a fault was significantly high when the injection time was within the susceptible window and reduced to 0 when the injection time was outside of this window. Moreover, the duration of the susceptible window was around 9 ns when the EM pulse voltage was 50 V and increased to 11 ns when the EM pulse voltage was set to 100 V. The probability of the faulty response was similar to a periodic signal and its period was 20 ns which was the same as the clock period. Note that the susceptible window duration was empirically shown to be independent of the clock period and located on both sides of the clock edge.

2.3 The Fault Exploitation

In this section, previous research on the fault exploitation part of FIA is discussed. Two most widely analyzed targets for fault exploitation are the cryptographic algorithm and the application-sensitive process [45]. The DFA is the typical fault exploitation approach for the cryptographic algorithm and will be introduced in Section 2.3.1. We will use the attack on secure boot to show the fault exploitation on the application-sensitive process.

2.3.1 Fault Exploitation on the Cryptographic Algorithms

The cryptographic algorithm is a typical target for the fault exploitation by using the DFA. The attacker analyzes the correlation between the faulty output and the correct output of a cryptographic module to reveal the secret information. Numerous DFAs were published during the last two decades targeting both public key cipher [81] and symmetric key cipher. Both attacks are based on the ability to inject some faulty data during a specific part of the algorithm. In the following section, a detailed description of the AES DFA algorithm is presented.

2.3.1.1 DFA towards the Symmetric Key Cipher

AES was designed to replace the DES as a new standard for various applications [81]. Even though this cipher is extremely strong against brute force attack or other mathematical cryptanalysis, DFA makes it vulnerable to FIA. In the following sections, two different DFA attacks targeting the AES algorithm are discussed.

Table 2.3: Notations for AES

Notation	Meaning
M_i^j	$j-th$ byte of the $i-th$ state (after the $i-th$ round)
SR_i^j	$j-th$ byte after the $i-th$ shift rows
SB_i^j	$j-th$ byte after the $i-th$ sbox
MC_i^j	$j-th$ byte after the $i-th$ mix columns
K_i^j	$j-th$ byte of the $i-th$ round key
C^j	$j-th$ byte of the ciphertext
SB^{-1}	Inverse sbox
SR^{-1}	Inverse shift rows

For consistence, the notations shown in Table 2.3 are used for describing the DFA attack targeting AES. AES is running the intermediate calculation based on a 4*4 matrix. Denote the $i-th$ *state* matrix, *ciphertext* and $i-th$ *key* matrix as:

$$i-th\ state = \begin{pmatrix} M_i^{0,0} & M_i^{0,1} & M_i^{0,2} & M_i^{0,3} \\ M_i^{1,0} & M_i^{1,1} & M_i^{1,2} & M_i^{1,3} \\ M_i^{2,0} & M_i^{2,1} & M_i^{2,2} & M_i^{2,3} \\ M_i^{3,0} & M_i^{3,1} & M_i^{3,2} & M_i^{3,3} \end{pmatrix}$$

$$ciphertext = \begin{pmatrix} C^{0,0} & C^{0,1} & C^{0,2} & C^{0,3} \\ C^{1,0} & C^{1,1} & C^{1,2} & C^{1,3} \\ C^{2,0} & C^{2,1} & C^{2,2} & C^{2,3} \\ C^{3,0} & C^{3,1} & C^{3,2} & C^{3,3} \end{pmatrix}$$

$$i-th\ key = \begin{pmatrix} K_i^{0,0} & K_i^{0,1} & K_i^{0,2} & K_i^{0,3} \\ K_i^{1,0} & K_i^{1,1} & K_i^{1,2} & K_i^{1,3} \\ K_i^{2,0} & K_i^{2,1} & K_i^{2,2} & K_i^{2,3} \\ K_i^{3,0} & K_i^{3,1} & K_i^{3,2} & K_i^{3,3} \end{pmatrix}$$

Giraud et al. proposed a DFA algorithm against AES based on a bit flip fault model where only one bit is flipped before the beginning of the final round [24]. Assume AES-128 is the target, the whole DFA algorithm works as follows [24]:

1. Ciphertext $C = SR(SB(M_9)) \oplus K_{10}$.

2. Denote $C^{SR(j)}$ and $K_{10}^{SR(j)}$ as the data after applying shift rows transform to the index of the ciphertext and the last round key K_{10}, respectively. we have $C^{SR(j)} = SB(M_9^j) \oplus K_{10}^{SR(j)}$.

3. Assume one bit is flipped in M_9^j. Denote the $j-th$ faulty byte of the $9-th$ state and the $j-th$ faulty byte of the ciphertext as $\widehat{M_9^j}$, $\widehat{C^j}$, respectively. Denote the induced bit fault as e. Thus, $\widehat{M_9^j} = M_9^j \oplus e$. Similar to step 2, we can determine $\widehat{C^{SR(j)}} = SB(\widehat{M_9^j}) \oplus K_{10}^{SR(j)}$.

4. By combining step 2 and step 3, we have $C^{SR(j)} \oplus \widehat{C^{SR(j)}} = SB(M_9^j) \oplus SB(\widehat{M_9^j}) = SB(M_9^j) \oplus SB(M_9^j \oplus e)$. For other correct bytes in M_9, $C^{SR(i)} \oplus \widehat{C^{SR(i)}} = 0$ where i is the index of the correct byte in M_9

5. Guess the single bit fault e and find M_9^j which matches $C^{SR(j)} \oplus \widehat{C^{SR(j)}} = SB(M_9^j) \oplus SB(M_9^j \oplus e)$. For each M_9^j, increase the corresponding counter if it is a correct solution. Finally, with another set of correct ciphertext and faulty ciphertext, the correct M_9^j is expected to have a higher counter value.

6. Apply the fault injection to the remaining 15 bytes in M_9 and finally determine the value of M_9.

7. With M_9 known to the attacker, the last round key K_{10} is easy to be retrieved. Finally, the attacker could get the initial key from the last round key which eventually breaks the AES-128.

However, for this DFA attack, the fault model requires a precise space and timing to inject a single bit fault to one byte of M_9 without affecting other bytes. Hence, this attack might be difficult to implement.

Different from the previous DFA which aims at injecting a fault after the $9-th$ round, some other papers [82, 83, 5] target the key scheduling algorithm of AES.

Kim and Quisquater proposed another DFA based on a fault model where one random byte is corrupted at K_9. Assume $K_9^{0,0}$ is corrupted with a fault a. We denote $\widehat{K_9^{0,0}}$ as the faulty key byte. Figure 2.6 illustrates the propagation of this faulty byte across the key scheduling algorithm in the last two rounds. Four bytes in the $9-th$ round key and two bytes in the $10-th$ round key are corrupted with the same fault a. They are $\widehat{K_9^{0,0}}$, $\widehat{K_9^{0,1}}$,

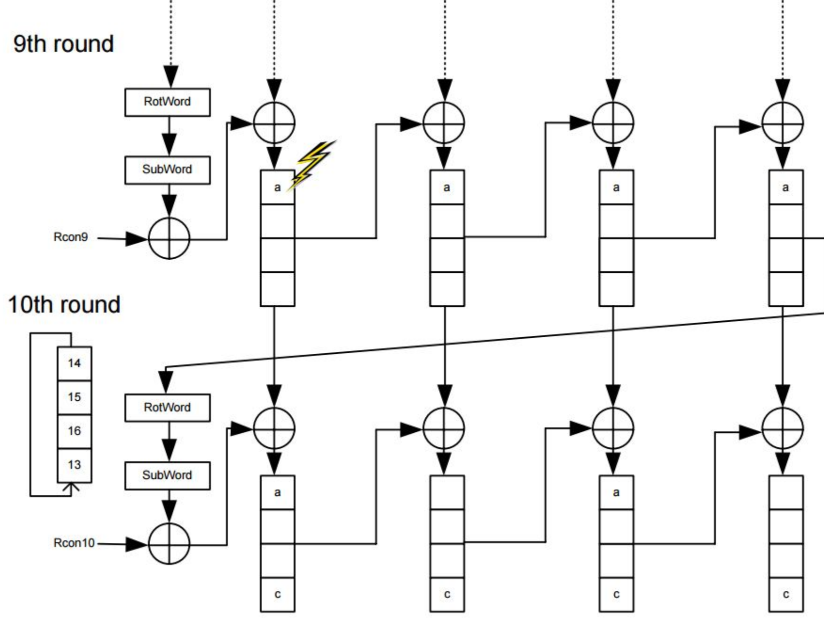


Figure 2.6: Faulty key byte propagation [5]

$\widehat{K_9^{0,2}}$, $\widehat{K_9^{0,3}}$, $\widehat{K_{10}^{0,0}}$ and $\widehat{K_{10}^{0,3}}$, respectively. The $RotWord$ performs a one byte circular and the $SubWord$ applies the AES sbox on the 4 input bytes from the last column of the $9-th$ round key. Thus, four other bytes are corrupted in the $10-th$ round key with a fault c. They are $\widehat{K_{10}^{3,0}}$, $\widehat{K_{10}^{3,1}}$, $\widehat{K_{10}^{3,2}}$ and $\widehat{K_{10}^{3,3}}$, respectively.

The faulty bytes in the round key also affect the intermediate calculation as shown in Figure 2.7 [5]. Assume the correct ciphertext is C and the faulty ciphertext is $\widehat{C}$. It is easy to observe that the relationship among a, the correct ciphertext, and the faulty ciphertext:

$$SB^{-1}(SR^{-1}(C^{0,i} \oplus K_{10}^{0,i})) \oplus SB^{-1}(SR^{-1}(\widehat{C^{0,i}} \oplus \widehat{K_{10}^{0,i}})) = a, i \in [0,1,2,3]$$

The following equation explains the relationship between the correct last round key and the faulty last round key:

$$K_{10}^{0,i} = \widehat{K_{10}^{0,i}}, i \in [1,3]$$
$$K_{10}^{0,i} = \widehat{K_{10}^{0,i}} \oplus a, i \in [0,2].$$

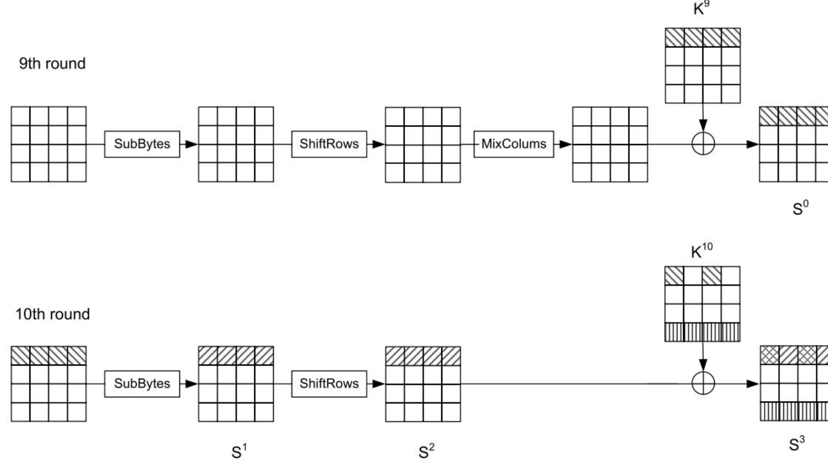


Figure 2.7: Propagation of the faults in the last two rounds after $K_{10}^{0,0}$ is injected with a fault [5]

Two pairs of the correct ciphertext and the faulty ciphertext are sufficient to retrieve the $K_{10}^{0,i}, i \in [0,1,2,3]$. Assume these two pairs of correct ciphertext and faulty ciphertext are denoted as C, $\widehat{C}$ and D, $\widehat{D}$ and the injected fault is a_1 and a_2, respectively. The first algorithm is to find the possible candidates for $(K_{10}^{0,1}, K_{10}^{0,3}, a_1, a_2)$ is described below:

1. Find the candidate for $(K_{10}^{0,1}, K_{10}^{0,3}, a_1, a_2)$. For all the possible combinations of $K_{10}^{0,1}, K_{10}^{0,3}$, calculate $\alpha_1, \alpha_2, \beta_1, \beta_2$:

$$\alpha_1 = SB^{-1}(SR^{-1}(C^{0,1} \oplus K_{10}^{0,1}) \oplus SB^{-1}(SR^{-1}(\widehat{C^{0,1}} \oplus K_{10}^{0,1})$$
$$\alpha_2 = SB^{-1}(SR^{-1}(C^{0,3} \oplus K_{10}^{0,3}) \oplus SB^{-1}(SR^{-1}(\widehat{C^{0,3}} \oplus K_{10}^{0,3})$$
$$\beta_1 = SB^{-1}(SR^{-1}(D^{0,1} \oplus K_{10}^{0,1}) \oplus SB^{-1}(SR^{-1}(\widehat{D^{0,1}} \oplus K_{10}^{0,1})$$
$$\beta_2 = SB^{-1}(SR^{-1}(D^{0,3} \oplus K_{10}^{0,3}) \oplus SB^{-1}(SR^{-1}(\widehat{D^{0,3}} \oplus K_{10}^{0,3})$$

2. If $\alpha_1 = \alpha_2$ and $\beta_1 = \beta_2$, put $(K_{10}^{0,1}, K_{10}^{0,3}, a_1, a_2)$ into a new list L_2 where $a_1 = \alpha_1, a_2 = \beta_1$.

After obtaining the new list L_2, the next step is to retrieve $K_{10}^{0,2}$ and build the third list L_3 which contains all the possible candidates for $(K_{10}^{0,1}, K_{10}^{0,2}, K_{10}^{0,3}, a_1, a_2)$.

1. For each possible $K_{10}^{0,2}$, choose a candidate from list L_2, calculate $\alpha_1, \alpha_2, \beta_1, \beta_2$ as follow:

$$\alpha_1 = SB^{-1}(SR^{-1}(C^{0,2} \oplus K_{10}^{0,2}) \oplus SB^{-1}(SR^{-1}(\widehat{C^{0,2}} \oplus K_{10}^{0,2} \oplus a_1)$$
$$\alpha_2 = SB^{-1}(SR^{-1}(C^{0,3} \oplus K_{10}^{0,3}) \oplus SB^{-1}(SR^{-1}(\widehat{C^{0,3}} \oplus K_{10}^{0,3})$$
$$\beta_1 = SB^{-1}(SR^{-1}(D^{0,2} \oplus K_{10}^{0,2}) \oplus SB^{-1}(SR^{-1}(\widehat{D^{0,2}} \oplus K_{10}^{0,2} \oplus a_2)$$
$$\beta_2 = SB^{-1}(SR^{-1}(D^{0,3} \oplus K_{10}^{0,3}) \oplus SB^{-1}(SR^{-1}(\widehat{D^{0,3}} \oplus K_{10}^{0,3})$$

2. If $\alpha_1 = \alpha_2 = a_1$ and $\beta_1 = \beta_2 = a_2$, put $(K_{10}^{0,1}, K_{10}^{0,3}, K_{10}^{0,3}, a_1, a_2)$ into a new list L_3 where $a_1 = \alpha_1, a_2 = \beta_1$.

Now the only unknown key byte in the first row of the $10 - th\ key$ is $K_{10}^{0,0}$. Note that other key bytes in the first row are also unknown but they have a limited number of candidates. The four key bytes in the first row can be retrieved after applying the last step:

1. For each possible $K_{10}^{0,0}$, choose a candidate from list L_3, calculate $\alpha_1, \alpha_2, \beta_1, \beta_2$:

$$\alpha_1 = SB^{-1}(SR^{-1}(C^{0,0} \oplus K_{10}^{0,0}) \oplus SB^{-1}(SR^{-1}(\widehat{C^{0,0}} \oplus K_{10}^{0,0} \oplus a_1)$$
$$\alpha_2 = SB^{-1}(SR^{-1}(C^{0,3} \oplus K_{10}^{0,3}) \oplus SB^{-1}(SR^{-1}(\widehat{C^{0,3}} \oplus K_{10}^{0,3})$$
$$\beta_1 = SB^{-1}(SR^{-1}(D^{0,0} \oplus K_{10}^{0,0}) \oplus SB^{-1}(SR^{-1}(\widehat{D^{0,0}} \oplus K_{10}^{0,0} \oplus a_2)$$
$$\beta_2 = SB^{-1}(SR^{-1}(D^{0,3} \oplus K_{10}^{0,3}) \oplus SB^{-1}(SR^{-1}(\widehat{D^{0,3}} \oplus K_{10}^{0,3})$$

2. If $\alpha_1 = \alpha_2 = a_1$ and $\beta_1 = \beta_2 = a_2$, output $(K_{10}^{0,0}, K_{10}^{0,1}, K_{10}^{0,2}, K_{10}^{0,3}, a_1, a_2)$ and stop.
3. Finally, $(K_{10}^{0,0}, K_{10}^{0,1}, K_{10}^{0,2}, K_{10}^{0,3})$ are revealed as the correct four bytes of the first row in matrix $10 - th\ key$.

The attacker only needs to inject another three faults in the $2nd, 3rd$ and $4th$ row of the $10 - th\ key$ and run the previous algorithms again to obtain the rest key bytes. Eventually, the last round key is revealed. Hence, the attacker could retrieve the initial key. Kim and Quisquater also simulated the attack on a 3.2 GHz Pentium 4 PC and successfully retrieved one row in around 0.5 seconds.

2.3.2 The Application-sensitive process

The fault exploitation could also be used to target application-sensitive process where the data processed should not be modified. The system security will be compromised if the data is modified [45]. The secure boot is one of this kind of application-sensitive process. In this section, we introduce the secure boot process and provide several examples of the fault exploitation on the secure boot.

The system on chip usually needs to communicate with external memory components which could not be trusted [6]. The secure boot process ensures the loaded data from external memory is not manipulated by the attacker by verifying its digital signature. Figure 2.8 presents a generic secure boot sequence [6]. Previous research proposed two possible attack approaches.

The first approach is attacking the signature verification or authentication process during the secure boot [25]. If the signature verification fails, the program jumps to an infinite loop. Assume the signature is invalid and the attacker is able to skip the branch instruction (to the infinite loop) as the result of a fault injection. In this case, the attacker could bypass the secure boot process.

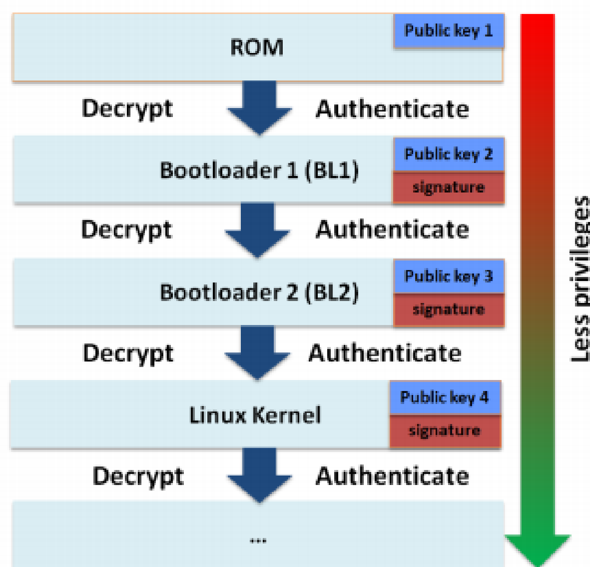


Figure 2.8: Secure boot process [6]

The second approach aimed at injecting a fault during copying of the data from the

external memory to the internal memory before executing the cryptographic verification [6].

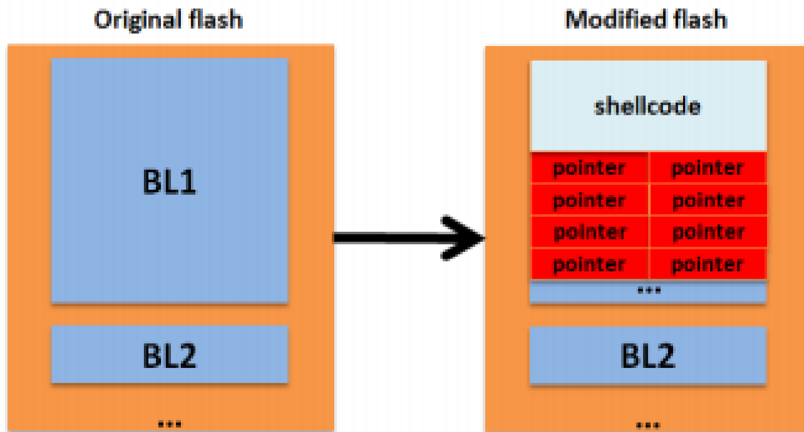


Figure 2.9: Modification of the external flash memory [6]

First, the attacker must know the destination address of the internal memory. This address is where the shellcode is going to be stored after being copied from external flash. Then, the attacker modifies the first level bootloader $BL1$ with a format of malicious shellcode plus pointers as shown in Figure 2.9. After that, the attacker keeps injecting fault during the copying phase. An exploitable injected fault may corrupt the load or store instruction and the pointer value is copied into the Program Counter (PC) register. Finally, the program successfully jumps to the shellcode and the cryptographic verification is bypassed.

2.4 Countermeasures

In this section, we provide a basic overview of the countermeasures designed for FIA. Countermeasures could be categorized based on different criteria. The first criterion is whether the countermeasure utilizes physical components or logical controls [84]. Based on this criterion, the countermeasures could be divided into physical countermeasures and logical countermeasures:

- **Physical Countermeasures**: Physical countermeasures aim at enhancing or monitoring the physical properties of the semiconductor to defeat or detect an FIA attempt. For example, a temperature sensor could be used to detect a sudden change of the operating temperature of the semiconductor which might be caused by a fault injection attempt (for instance, the attacker increases the operating temperature to increase the propagation delay and inject a fault). A real world example is the IBM 4764 Cryptographic coprocessor which applies different kinds of sensors to detect the tampering event as an FIA attempt [85]. However, this kind of countermeasure requires a mixed design which is usually very complex and of high cost [86].

- **Logical Countermeasure**: Logical countermeasures aim at detecting or correcting any induced fault by using parity check bits, cyclic redundancy check, and other kinds of error detection codes. Some designs duplicate the encryption or the instruction to compare the two results. Some fault tolerant designs such as the majority voter [87] can also be applied as the logical countermeasures. These countermeasures are easy to apply compared with the physical countermeasures. However, they may lead to a large overhead and may not detect all the faults injected [84, 26]. They may also aid the passive SCA such as the DPA [88].

Another criterion is the objective of the countermeasure. If the countermeasure only aims at detecting the FIA attempt, it is classified as the fault detection countermeasure. If the countermeasure not only detects the FIA attempt, but also corrects the faulty behavior of the chip after the fault is injected, it is called the fault tolerant countermeasure [61].

2.5 Summary

In this chapter, the basic theory of EM FIA was introduced. Additionally, we introduced the previous research on FIA based on its two phases: the fault injection and the fault exploitation. The countermeasures proposed were also summarized. Several hardware countermeasures designed for EM FIA are summarized in Appendix A.

In general, most previous research in the attack part of FIA could be divided into three categories as shown below:

1. **Research with a complete FIA**: In this category, the researcher successfully found vulnerabilities in an FIA and physically implemented FIA. Examples are the secure boot attack implemented with clock glitch [6], the fault assisted passive SCA where

the masking countermeasure was skipped using clock glitch [50], the round addition attack for AES using EM FIA [49].

2. **Research focused on the fault exploitation**: Some previous research hypothesized a fault model and evaluated the security of a cryptographic algorithm with regard to this particular fault model [24, 89, 82]. However, they did not physically run the attack with a particular FIA technique.

3. **Research focused on the fault injection**: The previous research focused on the fault injection by mainly evaluating different fault injection techniques like laser [66], clock glitch [63], and EM [59]. Some of them also focused on building the associated fault model with a specific fault injection technique [7, 4]. However, they did not physically implement an attack.

Even though some previous research formed a complete FIA, limited attacks were implemented with EM FIA [49, 74]. Some of the previous fault models did not take the clock frequency and the supply voltage into consideration [7, 4] and it was pointed out that reducing the supply voltage either help induce more reproducible fault with clock glitch [90] or make the target chip be more sensitive to clock glitch [6]. To the best of our knowledge, previous research did not analyze the combination of other fault injection techniques with EM FIA. Hence, there remains a lack of research on the impact of EM FIA with clock frequency, supply voltage and other parameters of the device under analysis.

The other part of the previous research is how to mitigate FIA. Some hardware countermeasures had a high fault detection rate [14], but they could not be implemented in an off the shelf microcontroller. It also had a high hardware overhead [15]. Some software countermeasures could be implemented in software to protect the microcontroller, but they were proposed based on the instruction skip fault model [91]. Software countermeasures for specific instructions were proposed and empirically verified by Moro et al. [61]. However, the countermeasure was not implemented on a complete cryptographic algorithm. Moreover, the countermeasure previously proposed for the LDR instruction detected the fault but did not correct the fault. Therefore, proposing and empirically verifying software countermeasures with a complete algorithm based on the understanding of the fault is important to protect the off the shelf microcontrollers.

The next chapter will provide details of the experimental setup used in the remaining chapters.

Chapter 3

Experimental Setup and Terminology

In this chapter, the experimental setup is presented in Section 3.1. Then, the characterization of the probe is discussed in Section 3.2 along with a table which presents the terminologies that will be used in the following chapters. Finally, the limitations of the equipment is summarized.

3.1 Experimental setup

The typical experimental setup for EM FIA is shown in Figure 3.1. The equipment required for EM FIA includes an oscilloscope, a pulse generator, a EM probe, a motorized x-y-z stage, etc. as shown in many research papers with similar setup [49, 59, 74]. The pulse generator can generate a voltage or current pulse which is later delivered to the EM probe to produce a strong transient EM pulse. A trigger signal is usually sent out from the DUT to synchronize the EM pulse [7, 59, 74, 58, 2, 54]. The EM pulse could be synchronized with the target clock cycle by using this trigger signal with some timing analysis. Due to the local property of EM FIA, most previous research applied a scan over the chip by using the motorized x-y-z stage to analyze the relationship between the faulty response and the probe location and find the best spot for injecting the fault [7, 4, 2, 59]. These scans typically require an automatic collaboration of different equipment to collect a large amount of data.

During this research, two stations were developed using two sets of probes and EM pulse generators from Langer EMV-Technik [8]. The first set of probes and pulse generator is a prototype system called Burst Power Station (BPS)201, whereas the BPS202 is a

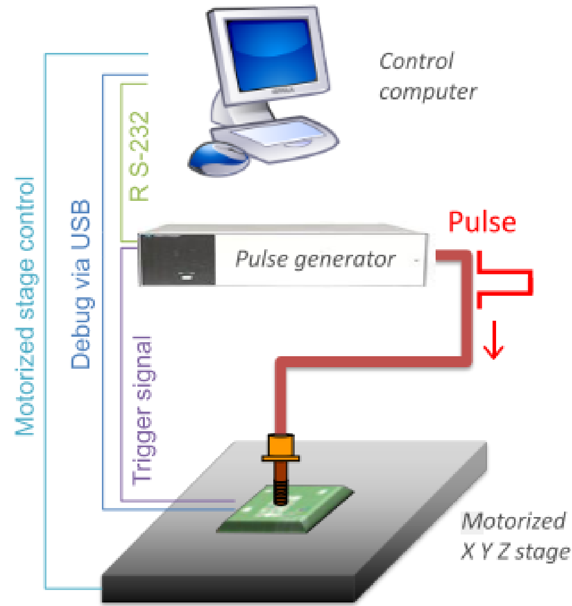
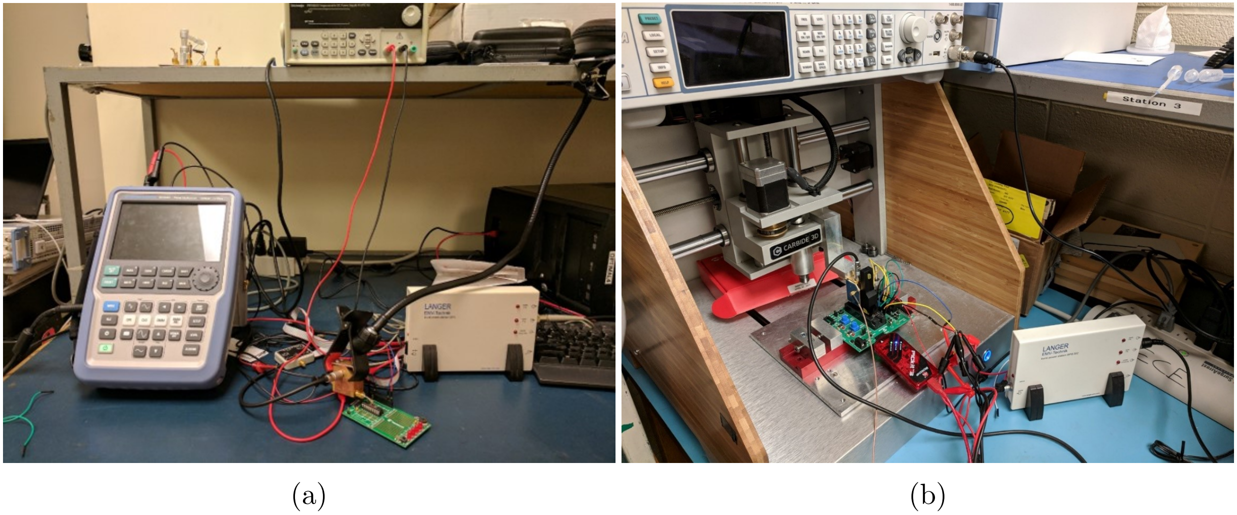


Figure 3.1: Typical experimental setup used in EM FIA [7]



(a) (b)

Figure 3.2: (a): Handheld experiment setup. (b): Automated platform with a CNC machine as x-y-z stage

completely functional EM injection system with remote control capability. Additionally, the diameter of the probe tip of the BPS202 is only 500 um which offers a better spatial resolution for running a scan over the silicon die to locate the best position for FIA.

The BPS201 offers a larger probe tip with an approximate 2.1 mm diameter which is more robust against damage. Thus, it was used for handheld experiments to identify if a fault could be injected under specific conditions. In contrast, BPS202 was used for running statistical analysis where large amounts of data were collected due to its remote control capability.

Figure 3.2 shows the two different kinds of EM FIA experimental setups which were developed. In Figure 3.2a, the prototype BPS201 system is shown and the probe is handheld (or held by the probe holder but moved by hand). A Computer Numerical Control (CNC) machine works as the x-y-z stage to move the probe tip as shown in Figure 3.2b for the BPS202 system. The resolution of the CNC machine is 12.7 um [92]. For this automated platform, a master python script acts as the main controller for the experiment as shown in Figure 3.3a. Figure 3.3b visually defines the position of the probe and the CNC machine. This control script will be discussed in subsequent chapters along with the associated experiments. This automated platform was utilized during the experiments for PIC16F687 detailed from Section 4.3 to Section 4.8. It was also used during injecting the fault to the LPC1114 which will be detailed in Section 5.2.2, however, the BPS201 was used in all subsequent experiments in Chapter 5 due to the damage of the probe tip of the BPS202.

Both BPS201 and BPS202 provide GUI to configure the EM pulse as shown in Figure 3.4. They could run in either the pulse mode where a single pulse is sent or the burst mode where a defined number of pulses are sent over a defined time duration. Other parameters can also be configured through this software such as the pulse frequency, pulse voltage, and trigger to pulse delay. Moreover, BPS202 also has the scripting capability to configure these parameters.

Both EM FIA systems have the external trigger mode which is applied to better control the timing of injecting the fault. In the external trigger mode, the voltage pulse, which is used to generate the EM pulse, is triggered with an adjustable delay after the BPS system receives the rising edge or the falling edge of the trigger signal. The trigger action could be set to either "single pulse" or "start pulsing". In our experiments, unless otherwise specified, all the pulses are generated by using a single pulse, rising edge trigger setting. Under this setting, only one pulse will be generated for each rising edge of the trigger signal.

In our experiments, the overclocking approach was applied to help inject the fault. This

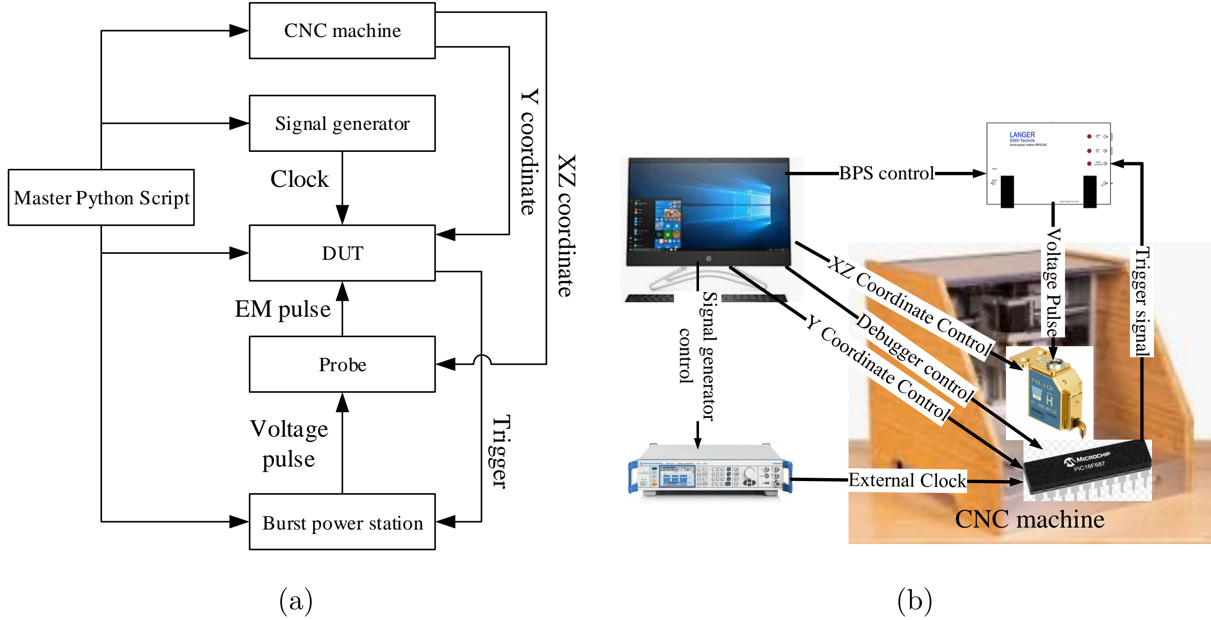


Figure 3.3: Control flow of the automated FIA platform

approach first determines a maximum clock frequency where the chip could run stably. Then, the clock frequency is reduced slightly (in all experiments) to further ensure fault-free stability of the chip without EM FIA. Hence, when EM pulses are delivered and faults are injected, the induced faults must be caused by the EM pulse instead of overclocking. Similar method was used in previous research where the supply voltage was reduced to help inject the fault with clock glitch [63].

3.2 Terminology and Characterization of the Probe

The terminology and characterization of the probe will be presented in this section. The probe characterization mainly helps us to verify the controllability and accuracy of the associated parameters such as voltage, delay, etc. These parameters significantly influence the result of a fault injection experiment as found in previous research [7, 59, 4]. Thus, characterizing the probe before conducting the fault injection experiment helps improve the efficiency of developing and verifying a fault model and mounting an FIA.

Table 3.1 gives a summary of the parameters and the associated terminologies that will be used in this section and also in the following chapters. The Tgt_inst is the instruction

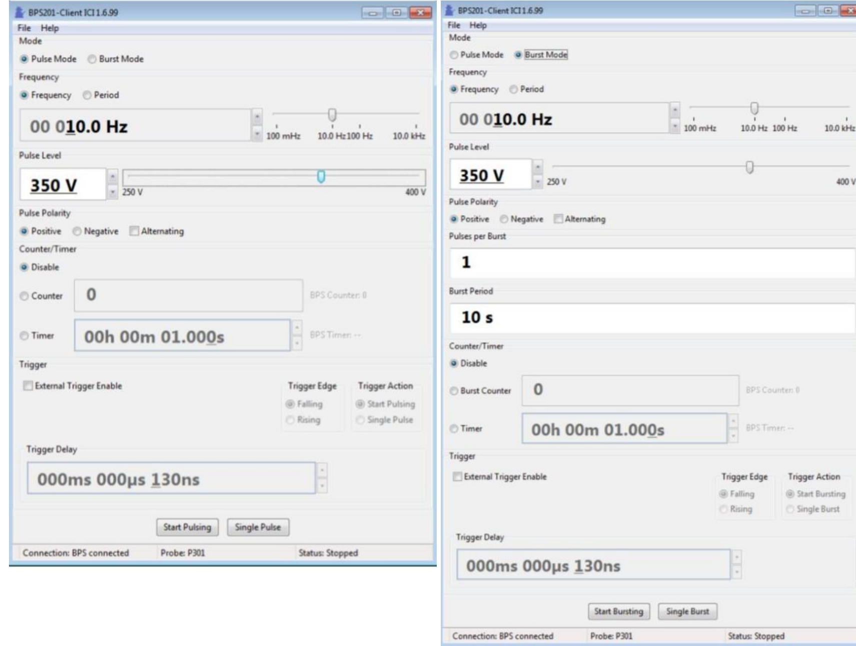


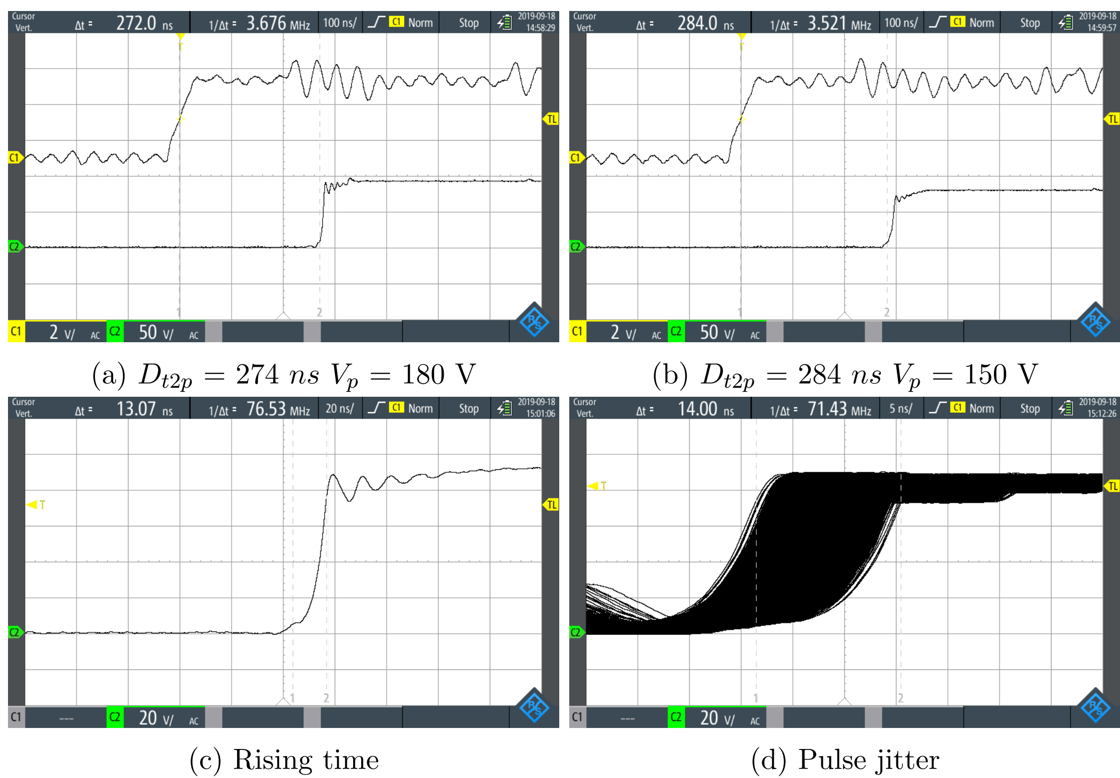
Figure 3.4: Main dialogue of the control software [8]

under attack. The EM pulse is delivered over a specific pipeline stage of the Tgt_inst. The Flt_inst is defined as a single instruction which functionally creates the faulty state which empirically occurred with the EM pulse injection of the Tgt_inst. If this Flt_inst could be found, the induced fault is classified as the instruction replacement fault.

Table 3.1: Terminologies and the associated parameters used in the thesis

Parameter	Description
F_{clk}	clock frequency of the external clock of the DUT
V_{DD}	Supply voltage of the DUT
V_p	Pulse voltage
D_{t2p}	Delay from the trigger to the pulse
P_{cycle}	Prefetch cycle
Tgt_inst	Target instruction
Flt_inst	Faulty instruction

The measurement of the voltage pulse sent out by the probe is shown in Figure 3.5 and 3.6. In these oscilloscope plots, channel one (C1 in Figure 3.5 and 3.6) was connected to the trigger signal and channel two (C2 in Figure 3.5 and 3.6) was connected to the voltage pulse port of the probe. From Figure 3.5a and 3.5b, the actual delay between the trigger



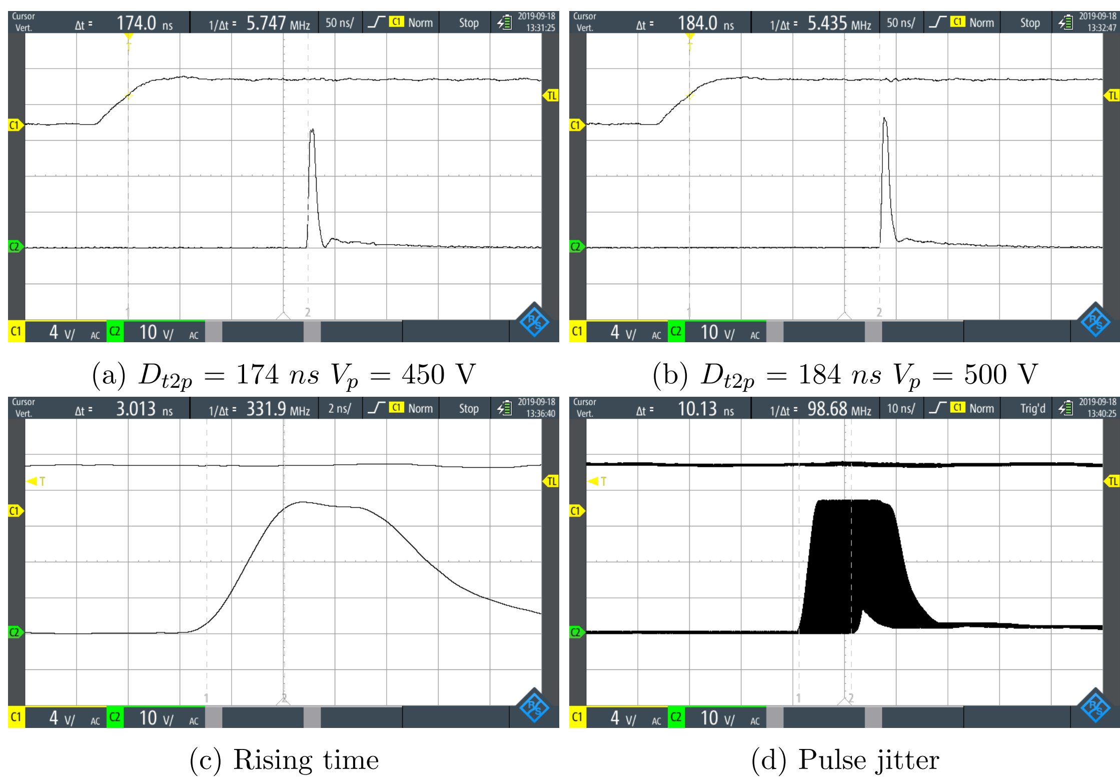
(a) $D_{t2p} = 274\ ns\ V_p = 180$ V

(b) $D_{t2p} = 284\ ns\ V_p = 150$ V

(c) Rising time

(d) Pulse jitter

Figure 3.5: Oscilloscope plots for BPS201 probe



(a) $D_{t2p} = 174\ ns\ V_p = 450$ V

(b) $D_{t2p} = 184\ ns\ V_p = 500$ V

(c) Rising time

(d) Pulse jitter

Figure 3.6: Oscilloscope plots for BPS202 probe

to the pulse is increased by 10 ns when we increased the setting of the D_{t2p} by 10 ns. Moreover, the pulse voltage was increased when we set V_p to a higher value by comparing Figure 3.5a with 3.5b. Similar behaviors were found for BPS202 as shown in Figure 3.6a and 3.6b. The rising time of the voltage pulse for BPS201 and BPS202 is approximately 13 ns and 3 ns, respectively (see figures 3.5c and 3.6c). Another important timing parameter is the jitter. The jitter of the voltage pulse was measured to be approximately 14 ns and 10 ns for BPS201 and BPS202, respectively (see figures 3.5d and 3.6d).

3.3 Limitations of the Equipment and Assumptions

The limitations of the equipment mainly exist in two areas. The first one is the timing limitation and the second one is the spatial limitation. Both will be introduced in this section along with some other limitations.

There are mainly three timing limitations. The first timing limitation is the minimum pulse period of the BPS system. The minimum period for two consecutive pulses is 100 us for BPS201 while it is reduced to 50 us with BPS202. Hence, a delay factor has to be inserted in our test programs in Chapter 4 and 5 to ensure the time duration between asserting the trigger is larger than the minimum period of the associated BPS system. Otherwise, the second pulse will not be generated. The second timing limitation is the resolution of changing the D_{t2p}. The resolution for both BPS201 and BPS202 systems is 10 ns. This resolution mainly limits two major experiments. The first one is the AES attack which will be further discussed in Chapter 4. If we would have had a higher resolution, such as 0.1 ns as utilized in previous research [49], we might have been able to scan through the associated clock cycle with a fixed clock frequency to further explore faulty responses. The second experiment is the empirical verification of the charge-based fault model in Chapter 6. In this experiment we had to change the clock frequency accordingly to make sure the delay between the pulse to the associated clock edge remained constant. The third timing limitation is the minimum D_{t2p}. The minimum D_{t2p} is 274 ns and 174 ns for BPS201 and BPS202, respectively. Hence, to correctly place the EM pulse over the target clock cycle, NOPs were often inserted between the instruction which asserts the trigger and the associated Tgt_inst.

There are two spatial limitations. The first spatial limitation is the difficulty of measuring the z-distance. The CNC machine could adjust the z-distance. However, we could not let the probe tip touch the chip directly since it may damage the probe tip. Hence, we can only estimate the z-distance. The second spatial limitation is the probe tip location

with regards to the silicon die of the target chip. This probe location can only be estimated visually with regards to the die and chip orientation.

In addition to the timing and spatial limitations, there are some other limitations brought by the equipment. Both pulse generators offer flexibility in controlling the pulse. However, the BPS201 could correctly generate the pulse based on the configuration in external trigger mode only when the V_p is set between 130 V to 180 V. In contrast, the operating range of V_p for BPS202 is from 50 V to 500 V. Both pulse generators could only change the V_p by 10 V per step. Moreover, the probe for BPS201 is not shielded, which produces a significant interference to other signals, equipment, etc. This interference could be identified by comparing the trigger signal captured by the oscilloscope in Figure 3.5a with Figure 3.6a. However, this interference is not the reason for the induced fault (later detailed in Chapter 4 and 5). The interference could always be observed from the oscilloscope as long as the BPS201 is generating the pulse, but only when the probe is close enough to the silicon die, the fault is injected. Hence, this interference is likely generated due to the coupling between the cable which connects the trigger signal to the oscilloscope and the BPS201 probe. Moreover, Even though the voltage pulse could be measured on both probes, the probe for BPS201 does not have a separate port to measure the voltage pulse. The probe tip has to be removed so that the oscilloscope could be connected to measure the generated voltage pulse.

The diameter of the probe tip for BPS202 is 500 um. Therefore, the probe tip needs to be placed close enough to the silicon die in order to inject the fault. Based on the feedback from the manufacturer, the distance between the DUT and the probe tip needs to be smaller than 100 um[1]. However, they also noted that this distance is target specific. At the same time, any direct contact between the probe tip and the silicon die needs to be avoided to prevent any damage to the probe tip. Hence, it is difficult to protect the probe tip and inject the fault at the same time.

The oscilloscope used in our experiments does not provide the functionality of measuring the jitter. Hence, the jitter measurement was done by setting the display model to infinite such that all the generated pulses were displayed in the oscilloscope screen. The jitter was measured by using the cursor. The D_{t2p} was also measured by using the cursor. Therefore, there might be some measurement errors.

During our experiments, the probe tip was damaged several times. The probe tip of BPS202 was sent back to the manufacturer twice for repair. The probe tips of BPS201 were also damaged several times. Though we could repair the probe tip by soldering the

[1]Based on the communication with Lars Glaesser from Langer EMV-Technik

copper wires back to the ferrite core, the repair process was extremely time consuming and the generated EM pulse was likely different due to the lack of proper tools.

The LPC1114 was placed over a custom breadboard. The chip was backside decapsulated and the pins were bent over. Hence, it was difficult to ensure a totally horizontal surface (PIC16F687 was equipped with a universal board). With this restriction, the experiments on chip #2 of LPC1114 were conducted with a handheld setup, which will be detailed in Chapter 5.

Based on our previous analysis of the system, two consecutive EM pulses could only be generated with at least a 100 us or 50 us period if we apply either the BPS201 or the BPS202 system, respectively. Assume only one fault injection system is used. Therefore, it is reasonable to assume that two consecutive EM pulse could not be generated with less than 50 us. To the best of our knowledge, the only available complete commercial EM FIA system is designed by Riscure and has a minimum period for generating the EM pulse of 1 us [77]. Hence, in Section 4.8 and 5.5 where the countermeasures were designed and verified, we assume that the fault could not be injected within 1 us. Moreover, the fault tolerant countermeasure (see Section 2.4) is further defined as the countermeasure which corrects the injected fault targeting a single specific instruction.

3.4 Summary

This chapter presented the experimental setup used in the following chapters. Both handheld experimental setup and automated platform were introduced. The master control python script was designed with different functionalities based on the purpose of the experiment, as will be detailed in subsequent chapters. The characterization of the probe was also presented and is important in running the detailed timing analysis in later experiments. The limitations of the system were analyzed along with the associated effects in our following experiments. In the next chapter, we will introduce our experimental results on PIC16F687.

Chapter 4

Experimental results on PIC16F687

In this chapter, we will introduce our experimental results on the PIC16F687 microcontroller. First, an overview of the target chip is presented. We present the experimental results with the handheld setup and also talk about the results from the automated platform. A two-step attack methodology is introduced with empirical results on attacking a software implementation of the AES-128. Countermeasures are proposed and empirically verified at the end of this chapter.

4.1 Overview of the PIC16F687

The target chip is PIC16F687 designed by Microchip [11]. It was chosen as the first target due to its simplicity and dual-in-line (DIP) package. It is designed with a Harvard structure where the program memory and the data memory are separated and accessed with separate buses. It has 128 bytes Static Random Access Memory (SRAM), 256 bytes Electrically Erasable Programmable Read Only Memory (EEPROM), and 2K*14 bits flash memory. In the datapath, the only register available is the W (or accumulator) register. The package of the chip is divided into four different types:

- Backside decapsulated chip: The backside package is removed along with the copper shield. The silicon substrate is exposed.
- Frontside decapsulated chip: The package of the chip is totally removed from the frontside and the die is exposed.

- Cavity chip: Part of the package is removed by drilling a cavity.
- Original chip: The package of the chip is not removed.

Figure 4.1: From top to bottom: backside decapsulated chip, frontside decapsulated chip, cavity chip and original chip [9]

These four kinds of chips are shown in Figure 4.1. The locations of the flash memory, EEPROM and SRAM in the silicon die viewed from the backside are identified as shown in Figure 4.2. The location of each memory is important for EM FIA since the orientation of the probe tip and the target circuit influences the induced electromotive force as described in Section 2.1.

According to the PIC16F687 user manual [11], there are four clock cycles denoted as $Q1, Q2, Q3, Q4$ to finish one instruction cycle denoted as T_{cy}. Additionally, the instructions are fetched and then executed in two instruction cycles. The PIC16F687 applies a two-stage pipeline where execution of the current instruction and fetch of the next instruction are done within one instruction cycle. These features are depicted in Figure 4.3. $OSC1$ denotes the original clock signal from the clock source which could either be an internal oscillator or an external oscillator. $OSC2/CLKOUT$ is output through PIN $RC4$ while the chip is running in RC mode. Note that the $OSC1$ or the actual Q cycles could not be determined or identified externally from the chip.

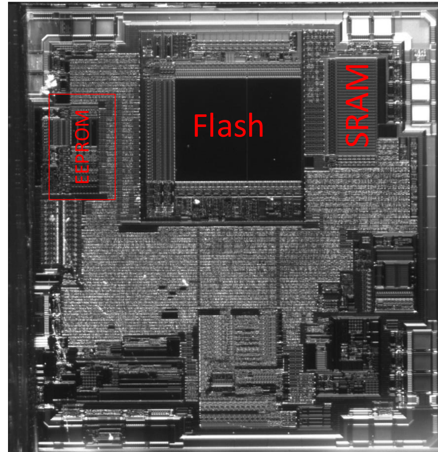


Figure 4.2: Backside view of the PIC16F687 die [9]

Additionally, during the execution cycle of a instruction, $Q1$ is always the decode stage. For most instructions, $Q2, Q3, Q4$ follows a read-modify-write process where $Q2$ is used to read the data (either from the register or from the literal), data processing is done at $Q3$, and the processor writes back to the destination register at $Q4$. It is likely that the processor loads the fetched instruction to the instruction register at $Q2$ after the instruction executed in parallel has been decoded [10]. Therefore, if the EM pulse changes the instruction, $Q2$ is likely the target clock cycle.

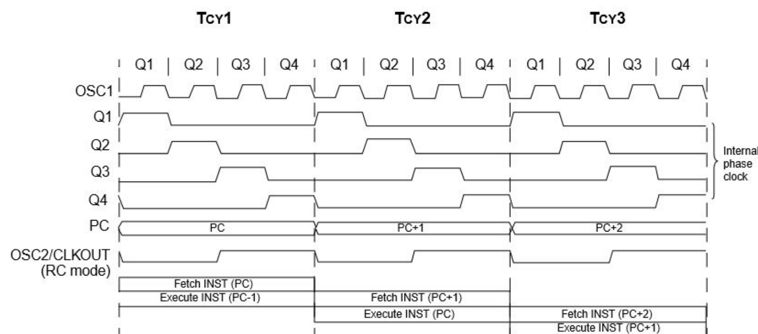


Figure 4.3: Clock/Instruction cycle [10]

Figure 4.4 presents a special case for the two-stage pipeline flow. Normally, each instruction needs one instruction cycle to be fetched and the second instruction cycle to be executed. With the two-stage pipeline, in each instruction cycle, one instruction will be

executed. However, if the executed instruction changes the PC, another instruction cycle is required to flush the fetched instruction. As shown in Figure 4.4, the $CALL\ SUB_1$ instruction changes the PC to execute the instruction addressed at label SUB_1. Thus, the fetched fourth instruction is executed as a NOP instruction in T_{cy4}.

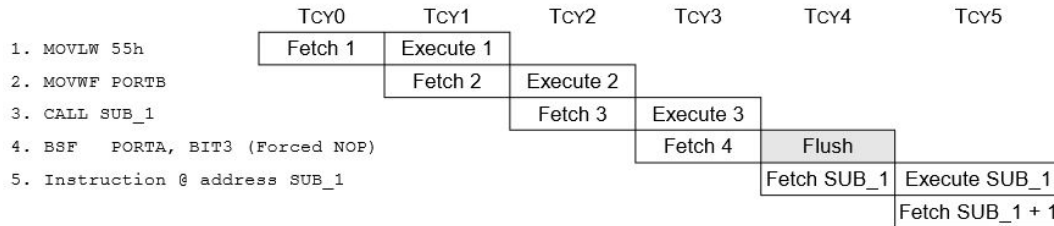


Figure 4.4: Pipeline flow [10]

The maximum clock frequency allowed under different supply voltages for PIC16F687 is shown in Figure 4.5. The maximum clock frequency is 20 MHz when the supply voltage is set to 5.5 V.

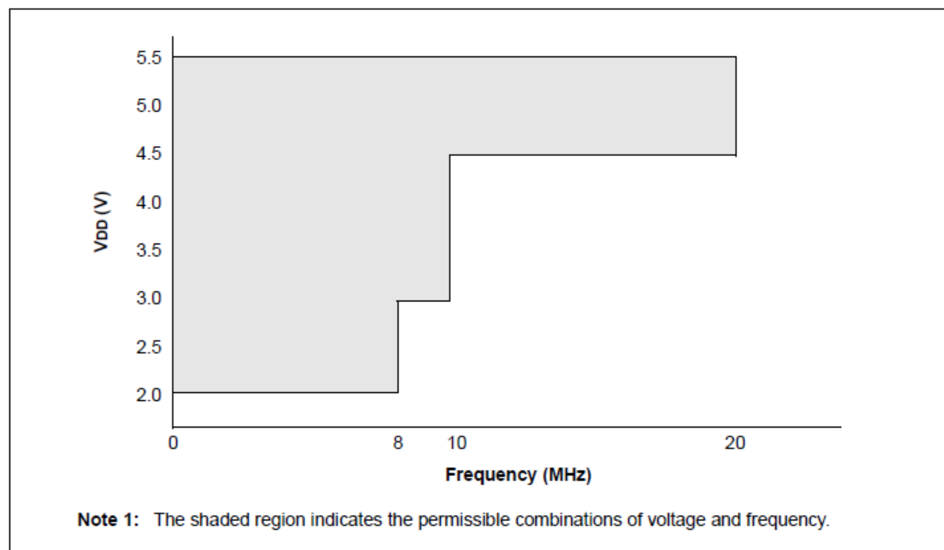


Figure 4.5: Voltage-Frequency graph [11]

Table 4.1 presents the instructions that have been analyzed during our experiments. Other instructions and macros are listed in Appendix B.1. These target instructions were

chosen to include memory reading/writing, ALU operations, and conditional operations. The PIC16F687 utilizes a 14-bit encoding for each instruction. The field f is the binary encoding of the register file address. Similarly, the field k is the binary encoding of the literal, constant or label. d is the destination selection bit (select f or w according to the value of d). Further analysis indicates that some instructions share similar encoding even though they have different functionalities. For example, the hamming distance between the opcode of $GOTO\ k$ and $MOVLW\ k$ is two when k is less than 256.

The PIC16F687 supports direct, indirect and relative addressing mode [11]. The indirect addressing mode is achieved by accessing the $INDF$ register. This register is not a physical register. Any instructions accessing this register actually access the data pointed by the File Select Register (FSR).

Table 4.1: Instruction description and the associated opcode

Instruction	Description	Opcode (14 bits)
MOVLW k	Move literal k to W register	11 0000 kkkk kkkk
MOVWF f	Move the content in W register to f	00 0000 1fff ffff
BCF f, b	Clear bit b of the content stored in f	01 00bb bfff ffff
GOTO k	Jump to the next instruction addressed at 0x80	10 1kkk kkkk kkkk
XORWF f, d	Exclusive or the contents of W register with f and then store the result back to f/W	00 0110 dfff ffff
SUBWF f, d	Subtract W from F, the result was stored back to F/W based on d	00 0010 dfff ffff
IORWF f, d	Inclusive or the contents of W register with f and then store the result back to f/W	00 0100 dfff ffff
DECFSZ f, d	Decrement f, skip the next instruction if f = 0 and store the result back to f/W	00 1011 dfff ffff
DECF f, d	Decrement f, result is stored back to f/w	00 1011 dfff ffff
INCF f, d	Increment f, result is stored back to f/w	00 1010 dfff ffff
COMF f, d	Complement f, result is stored back to f/w	00 1001 dfff ffff
NOP	No operation	00 0000 0000 0000

The PICkit3 programmer/debugger could be used to program/read/erase the PIC16F687 with scripting capability. However, the reading process only dumps out the program memory, user ID, device ID, configuration memory and the EEPROM memory [93]. Hence, this debugger does not provide access to the instruction register which stores important information for investigating the instruction replacement/skip fault. Also, the fault test program must have the functionality to transfer the faulty data from SRAM to EEPROM such that it could be further dumped out with the PICkit3 for off chip analysis. The

PICkit3 also supports reading the SRAM memory on the fly in debug mode. However, the debug function is only available with a specific header chip but not supported with PIC16F687. The test programs have to be carefully designed such that the faulty instructions could be identified.

Except for the terminologies shown in Table 3.1, D_{p2n} denotes the delay from the pulse to the end of Q_n cycle ($n \in [1, 2, 3, 4]$). In the experiments of PIC16F687, the instruction replacement fault was observed.

The experiments on PIC16F687 were conducted with three chips. Chip #1 is a front side cavity chip. This chip was utilized in the handheld experiment in Section 4.2. Both chip #2 and chip #3 are backside decapsulated. Chip #2 was applied in Section 4.3, 4.4, 4.5, 4.6, 4.8, and also from Appendix B.2 to B.5. Chip #3 was utilized in Section 4.7.

4.2 Handheld Experiment on Chip #1

The handheld experiment mainly investigated the fault injectability with the available BPS201. A front side cavity chip, chip #1, was utilized as shown in Figure 4.6. The cavity was drilled to decrease the distance between the probe tip and the silicon die since faults could not be injected with a regular chip. The diameter of this probe tip is around 2.1 mm and it does not have a sharpened end.

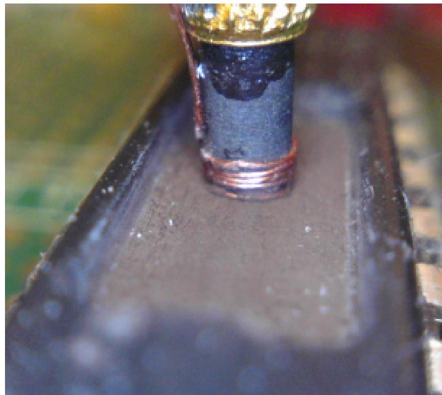


Figure 4.6: Large probe tip from BPS201 over a front side cavity chip, chip #1 of PIC16F687

The first step of the FIA experiment is to inject the first fault to the target chip. For better analysis of the fault, a specific program was developed. The flow of the program is

shown in Figure 4.7. The main functionality of the program is to write a fixed value to the SRAM memory, using $MOVWF$ as the Tgt_inst, and after writing 96 bytes to different memory addresses (see listing 4.1), check if the values stored in the SRAM memory are correct. The 96 bytes are written to SRAM memories from address 0x20 to 0x7F. The code that contains the loop to write the literal to the target SRAM and generate the trigger signal is shown in listing 4.1. As indicated by Velegalati et al.[32], the EM pulse may shut down the whole chip. Thus, LED 0 flashes every iteration to show that the program is still running when the data stored in the SRAM is fault free. If the data stored in the SRAM memory is changed, it confirms the presence of the fault and the program enters a fault handling loop where the faulty data is transferred from the SRAM to the EEPROM for off chip analysis. During the writing process, a trigger signal is asserted before writing each byte to the SRAM and de-asserted after each byte is written. This trigger signal is used to synchronize the EM pulse with the Tgt_inst.

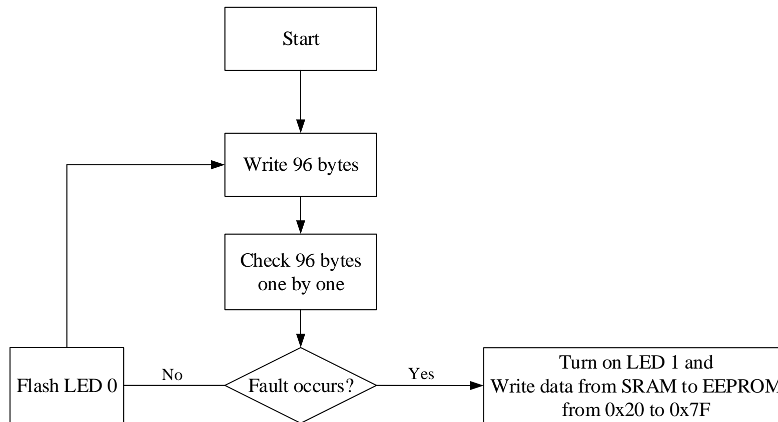


Figure 4.7: Program for handheld experiment

The fault could not be injected if the chip was not overclocked, even though the V_p was set to the maximum value. To make sure the fault was caused by the EM pulse instead of the overclocking (see Figure 4.5 for maximum nominal clock frequency), we verified that chip #1 could run the test program without faults with a 64 MHz clock (see Section 3.1 for the overclocking approach). To successfully inject the fault into chip #1, we set F_{clk} as 61.5 MHz resulting in a 16.26 ns clock period and a 4*16.26 = 65.04 ns instruction cycle.

The program was written in assembly language to provide a detailed analysis of the timing as shown in listing 4.1. The minimum D_{t2p} for BPS201 is 274 ns. With a 65.2 ns instruction cycle, there should be four instructions ($4 \times 65.04 < 274 < 5 \times 65.04$)

between asserting the trigger and the target $MOVWF\ INDF$. Hence, three additional NOPs were added to locate the EM pulse over the target $MOVWF\ INDF$ instruction. Assume the trigger signal was asserted at time 0 (at the end of instruction cycle 0 in Figure 4.8). The execution cycle (instruction cycle 5) of the $MOVWF\ INDF$ instruction started at 260.16 ns and ended at 325.20 ns as shown in Figure 4.8. Therefore, the EM pulse was delivered during the execution of the $MOVWF\ INDF$ (instruction cycle 5). The value written to the SRAM was 0x55.

```
1      MOVLW 0x20
2      MOVWF FSR ;Use indirect addressing mode.
3      LOOP2
4      BCF STATUS, RP1
5      BCF STATUS, RP0
6      MOVLW 0x20
7      MOVWF PORTC    ; Assert the trigger.
8      NOP
9      NOP
10     NOP
11     MOVLW 0x55 ; Move literal to W register before writing it to SRAM
12     MOVWF INDF   ; Target MOVWF instruction. Write content of the W reg to
       the register pointed by FSR.
13     MOVLW 0x00
14     MOVWF PORTC ; De-assert the trigger
15     INCF FSR,1 ; increment the address in FSR
16     CALL Delay1 ; The delay subroutine. Used due to the minimum pulse period
       .
17     BTFSS FSR,7 ; bit[7] of FSR =`1', indicating 96 bytes have been written
       to the SRAM
18     GOTO LOOP2
```

Listing 4.1: Program used for handheld experiment

Trigger RC5									
Instruction cycle		0	1	2	3	4	5	6	7
Prefetch	MOVWF PORTC	NOP1	NOP2	NOP3	MOVLW 0x55	MOVWF INDF	MOVLW 0X00	MOVWF PORTC	......
Execution	MOVLW 0x20	MOVWF PORTC	NOP1	NOP2	NOP3	MOVLW 0x55	MOVWF INDF	MOVLW 0X00	MOVWF PORTC

Figure 4.8: Timing analysis of the processor executing program 4.1 with $D_{t2p} = 274\ ns$, showing EM pulse occurring during instruction cycle 5

Several experiments were performed and faulty data was analyzed. As shown in Figure 4.9, two experiments indicated the faulty value was 0x54, which confirmed that EM FIA successfully changed the value inside the memory.

Add...	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	ASCII
000020	55	55	55	55	55	55	55	55	55	55	55	54	55	55	55	55	UU...
000030	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	UU...
000040	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	UU...

Memory View

Add...	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	ASCII
000050	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	UU...
000060	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	UU...
000070	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	UU...

(a)

Add...	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	ASCII
000020	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	54	UU...
000030	55	55	55	55	55	55	55	55	55	55	55	55	55	54	55	55	UU...
000040	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	UU...

Memory View

Add...	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	ASCII
000050	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	UU...
000060	55	55	55	55	55	55	55	55	54	55	55	55	55	55	55	55	UU...
000070	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	55	UU...

(b)

Figure 4.9: Partial memory dumps indicating (a):fault injected at address 0x2B and (b): fault injected at addresses 0x2F, 0x3D and 0x68

The attack flipped the last bit of the memory from '1' to '0'. Further analysis of the program indicated that the fault was not a reset bit fault but was actually related to the prefetch of the instruction which follows the $MOVWF\ INDF$. As shown in Figure 4.8, the chip is also prefetching the $MOVLW\ 0x00$ instruction in parallel when the $MOVWF\ INDF$ instruction is executing. In fact, it was empirically shown that the $MOVLW\ 0x00$ instruction was replaced with the $BCF\ INDF, 0$ instruction (see Section 4.4.3). As shown in Table 4.1, both instructions have a similar opcode with a hamming distance of 2.

In some experiments, even though the fault handling loop is executed, no faulty data was found in the EEPROM. This kind of fault will be classified as a control fault where the Tgt_inst is likely replaced with a jump instruction which forces the program to execute the fault handling loop. An example is presented in Section 4.4.3 where the Tgt_inst is a $MOVLW$ instruction. The next experiments will use the CNC machine with an automated setup (see Chapter 3).

4.3 Identify the Best Location for Injecting the Fault on Chip #2

After the handheld experiment, we built the automated platform to run EM FIA. The automated platform allows us to repeatedly run the experiment under the same setting and conduct a statistical analysis on the faulty behavior. Moreover, the motorized x-y-z stage enables us to precisely move the probe over the die of the chip and find out a best location to inject the fault. The BPS202 system was used due to its remote control capability. The AES-128 attack was also mounted on this automated platform. All the experiments done on the automated platform used a backside decapsulated chip (which was referred to as chip #2). There are mainly two reasons for using a backside decapsulated chip. First, driven by the mobile phone market, more flipchip packages are applied in the last decade due to the advantage of high performance and small package size [94]. Second, if the chip is decapsulated from the front side, it is more likely to damage the chip because of the bonding wires.

Figure 4.10 demonstrates how the probe tip was protected during the experiment. A probe cover was utilized to make sure the minimum distance between the probe tip and the package of the chip was 0.25 *mm*. Moreover, the silicon die was approximately 0.25 *mm* lower than the package when the chip was decapsulated from the backside. Therefore, when the probe tip was moving over the die, the minimum z-distance from the tip to the die was approximately 0.5 *mm*.

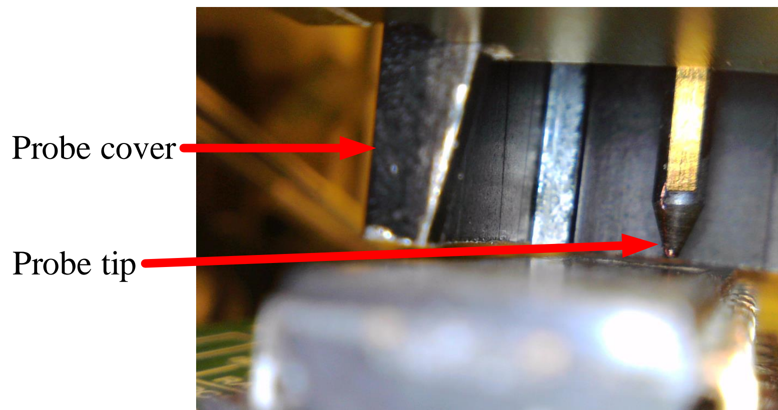


Figure 4.10: The probe tip protected by the cover over the target chip

4.3.1 Effect of the V_p and the Probe Location

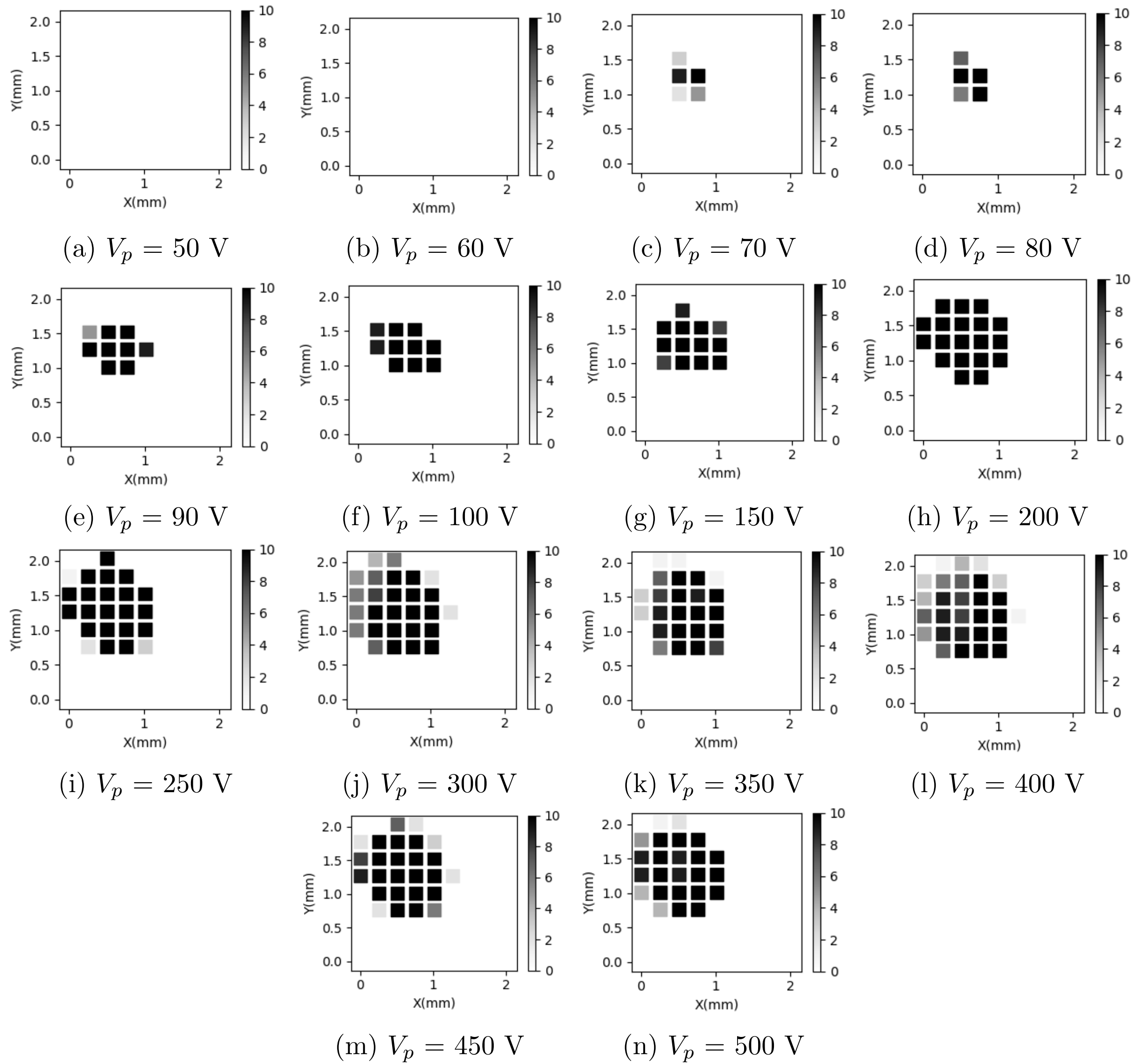
To analyze the impact of the V_p and probe location, two scans were performed to find a best point where fault is most likely to be injected. In the first scan, the probe was moved over the whole silicon die of the chip with a low resolution. In the second scan, the probe was moved in a sensitive area found by the first scan with a high resolution.

The same target program (shown in listing 4.1) was used in the scan. However, since chip #2 was used in this scan, the maximum F_{clk} was changed to approximately 56.5 MHz. Then, a 54.56 MHz clock was applied to help inject the fault. With F_{clk} set to 54.56 MHz, the delay from the trigger to the start and the end of the P_{cyl} of the $MOVLW$ $0x00$ instruction was 293.3 ns and 366.6 ns, respectively. Thus, we set the D_{t2p} as 294 ns to deliver the EM pulse while the chip was prefetching the $MOVLW$ $0x00$ instruction.

4.3.1.1 Scan Over the Die of The Chip

The first experiment we conducted on the automated platform was a scan over the die of the chip. As shown in Figure 3.2b, the probe was attached to the CNC machine by using a custom made probe holder. The size of the die of the PIC16F687 is approximately 2 $\times$ 2 mm^2. The objective of this scan was to determine a sensitive area where the fault was more easily to be injected. The CNC was set to move 0.254 mm per step. Initially, the probe tip was placed approximately over a corner of the die. To ensure that the scan covered the die area, the probe tip was moved over a grid of 9 $\times$ 9 points to form a square of 4.13 mm^2. A counter was used to record how many rounds out of a maximum of 10 rounds, had a successful fault injection at that specific xy coordinate. Moreover, for each round, the probe tip only stayed at each point for 12 s maximum trying to inject a fault. Whether the fault was injected or not, the probe tip would move to the next point after this time duration. After all the 81 points were finished, the scan would start over in the next round again until all the 10 rounds were finished. The final result is shown as shmoo plots in Figure 4.11, which corresponds to roughly the same backside view of the die in Figure 4.2.

The shmoo plots in Figure 4.11 indicated that the V_p has a significant impact on the fault injectability when it is below 200 V. No fault could be injected when the V_p is 50 V or 60 V and more faults are injected at a larger area over the die when the V_p is increased. However, when the V_p is greater than 200 V, there is no significant difference for the fault injectability. With this finding, another scan with a higher resolution was conducted with the V_p less than 200 V.



(a) $V_p = 50$ V (b) $V_p = 60$ V (c) $V_p = 70$ V (d) $V_p = 80$ V

(e) $V_p = 90$ V (f) $V_p = 100$ V (g) $V_p = 150$ V (h) $V_p = 200$ V

(i) $V_p = 250$ V (j) $V_p = 300$ V (k) $V_p = 350$ V (l) $V_p = 400$ V

(m) $V_p = 450$ V (n) $V_p = 500$ V

Figure 4.11: Shmoo plot for number of fault been injected at each coordinate over 10 rounds for the experiment scan over the die of chip #2 under different V_p

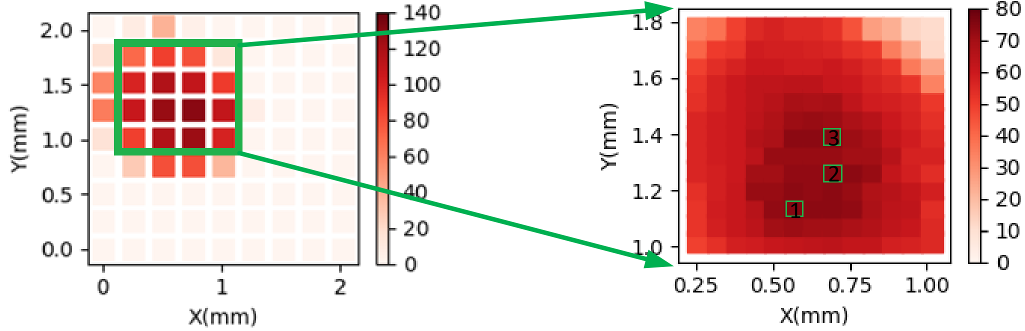


Figure 4.12: Cumulative shmoo plots over all V_p settings. Left: scan over the die; Right: scan over the sensitive area

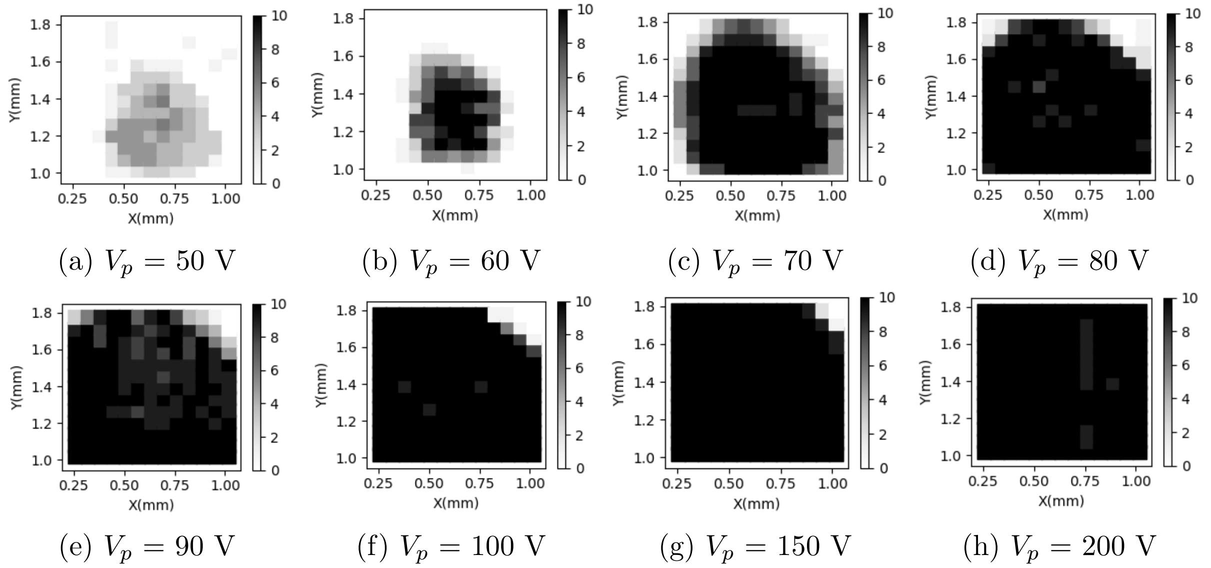
Moreover, from the left part of Figure 4.12 which plots the total number of faults injected at each point over 10 rounds for all the 14 different V_ps (which makes a maximum number of faults to be 140), a sensitive area is roughly found in the (green) square.

4.3.1.2 Scan Over the Sensitive Area

The second scan was done over the sensitive area found in the last experiment. The resolution was increased to 0.0635 mm per step to find the best point to inject the fault. Thus, the total number of points for each round was $13 \times 13 = 169$. Also, since when $V_p \geq 200\ V$, the delivered EM pulse reached its limit and could not inject significantly more faults to the chip, the V_p was limited up to 200 V in this scan. The result of this scan is shown as shmoo plots in Figure 4.13.

Similarly, to the right side of Figure 4.12, the three points with the most number of faults (75 out of 80 rounds) injected are listed with index 1, 2, 3 in Table 4.2. The final best point was chosen as the second one.

After determining the best location for injecting the fault, we also analyzed the effect of the z-distance at this best location and found that with a increased z-distance, the fault was more difficult to be injected. The result of the z-distance experiment is presented in appendix B.2. Additionally, several test programs were developed to investigate possible Flt_insts. The experimental results for these test programs are presented in the next section.



(a) $V_p = 50$ V (b) $V_p = 60$ V (c) $V_p = 70$ V (d) $V_p = 80$ V

(e) $V_p = 90$ V (f) $V_p = 100$ V (g) $V_p = 150$ V (h) $V_p = 200$ V

Figure 4.13: Shmoo plot for number of fault been injected at each coordinate over 10 rounds for the experiment scan over the sensitive area of the chip under different V_p

Table 4.2: Best probe position (xy) for largest number of faults injected using cumulative shmoo plot on right hand side of Figure 4.12

Index of the best point	x-coordinate (mm)	y-coordinate (mm)
1	0.5715	1.143
2	0.6985	1.27
3	0.6985	1.3335

4.4 Case Study: Analysis of The *Flt_inst*

To further analyze *Flt_inst*s, several different programs were designed. In this section, we will provide the details in investigating the instruction replacement fault based on different test programs. Generally, each program utilizes two loops. The first one is the normal function loop. In this loop, the program continues a write-check process until a fault occurs (similar to Figure 4.7). When the fault occurs, the program jumps into the second loop called the fault-handling loop. In the fault-handling loop, the program writes the faulty data into the EEPROM for off-chip analysis. All the experiments in this section were done with $V_p = 500\ V$, $V_{DD} = 5\ V$, $F_{clk} = 52\ MHz$.

4.4.1 Test Program One

Listing 4.2 presents part of the code for test program one. The program first writes a 0xAA to the general purpose register whose address is 0x20. Then, it writes a 0x55 to this address again. This process ensures that every bit is flipped during our writing process. If no fault is injected, the value stored at the register should be 0x55.

```
1  MOVLW 0x20
2  MOVWF PORTC ; Assert the trigger
3  NOP
4  MOVLW 0xAA ; Move literal 0xAA to W register
5  MOVWF 0x20 ; Move the data in W register to SRAM at address 0x20
6  MOVLW 0x55 ; Move literal 0x55 to W register
7  MOVWF 0x20 ; Target instruction. Move the data in W register to SRAM at
     address 0x20
8  MOVLW 0x00 ; Target instruction.
9  MOVWF PORTC ; De-assert the trigger
```

Listing 4.2: Test program one

The timing diagram for test program one is shown in Figure 4.14. In this timing diagram, the Q cycle of each instruction is also labeled. The timing diagram is used to analyze which instruction cycle is affected by the EM pulse. When D_{t2p} was set to 254 ns, the EM pulse targeted the fourth instruction cycle where the $MOVWF\ 0x20$ was prefetched. Similarly, the EM pulse was placed over the fifth instruction cycle where the $MOVLW\ 0x00$ was prefetched when D_{t2p} was 334 ns. Based on the faulty data in Table 4.3 and assume the fault was injected in the P_{cyl}, the associated *Flt_inst*s are shown in Table 4.3. The hamming distance between the *Tgt_inst* and the *Flt_inst* is "1" for both cases, as shown in Table 4.4 where "x" stands for don't care. We could find that the hamming distance between the *Tgt_inst* and the *Flt_inst* is only 1.

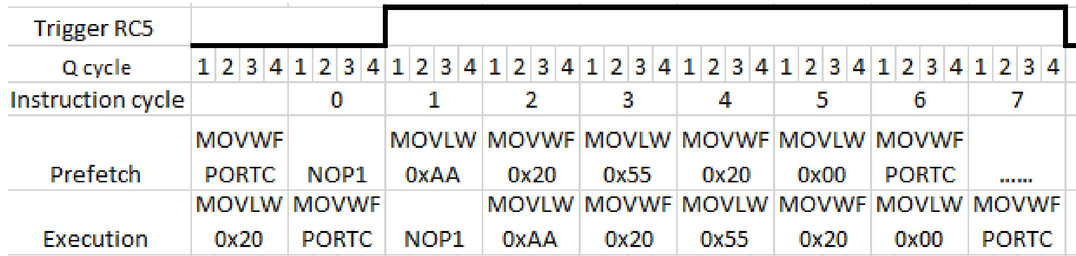


Figure 4.14: Timing diagram for test program one

Table 4.3: Possible faults injected on test program one

D_{t2p} (ns)	Faulty Value (hex)	Tgt_inst	Flt_inst
254	0xAA	$MOVWF$ 0x20	NOP
334	0x54	$MOVLW$ 0x00	BCF $INDF$, 0

Table 4.4: Hamming distance between Tgt_inst and Flt_inst of test program one

Tgt_inst	opcode	Flt_inst	opcode	Hamming distance
$MOVWF$ 0x20	0000001 0100000	NOP	0000000 xx00000	1
$MOVLW$ 0x00	1100xx0 0000000	BCF $INDF$, 0	0100000 0000000	1

4.4.2 Test Program Two

Test program two was designed with a more complex write-check loop as shown in listing 4.3. Instead of only writing a single byte to the SRAM, 0x55 and 0x03 are written to the SRAM memory whose addresses are 0x20 and 0x21, respectively. Then, the values at both addresses are added together and stored at address 0x23. In this way, when the fault occurs at either address, we could have an extra check at address 0x23. This extra check outputs additional information when the fault occurs. If no fault is injected, the expected data at these three address locations should be 0x55, 0x03 and 0x58, respectively. Initially, the data stored at all three addresses is 0xAA.

```
1  MOVLW 0x20
2  MOVWF PORTC ; Assert the trigger
3  NOP
4  NOP
5  NOP
6  MOVLW 0x55 ; Target instruction
7  MOVWF 0x20 ; Target instruction
8  MOVLW 0x03 ; Target instruction
```

```
9   MOVWF 0x21 ; Target instruction
10  NOP
11  NOP
12  MOVLW 0x00 ; Target instruction
13  MOVLW 0x00 ; Duplicated MOVLW 0x00 to ensure the trigger signal will be de
      -asserted
14  MOVWF PORTC ; De-assert the trigger
15  MOVF 0x20, W ; Move the content at0x20 to W reg
16  ADDWF 0x23, F; Add the content of W reg with content at 0x23, store the
      result back at 0x23
17  MOVF 0x21, W;Move the content at 0x21 to W reg
18  ADDWF 0x23, F;Add the content of W reg with content at 0x23, store the
      result back at 0x23
```

Listing 4.3: Test program two

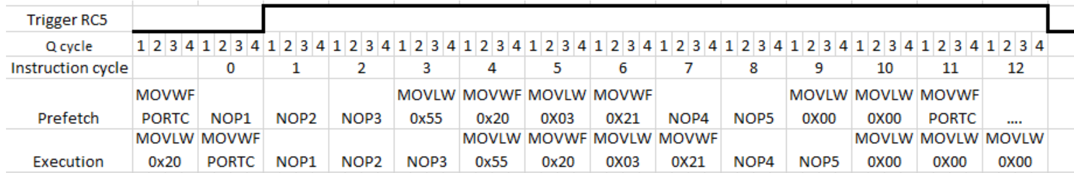


Figure 4.15: Timing diagram for test program one

The timing diagram of this program is shown in 4.15. From test program one, we understood that the fault was injected during the P_{cyl} of each instruction. Hence, the EM pulse was induced during the instruction cycle 3, 4, 5, 6, 9 to check the result. The faulty values were shown in Table 4.5. When the EM pulse targeted instruction cycle 3 and 5, the $MOVLW$ instruction was possibly replaced with the NOP instruction. Consequently, the previous value stored at W register was finally moved to the associated SRAM memory. The $MOVWF$ instruction could also be replaced with the NOP instruction if the EM pulse targeted instruction cycle 4 and 6. If the EM pulse was induced during instruction cycle 9, the $MOVLW$ $0x00$ was replaced with the BCF $INDF$, 0 instruction. This Flt_inst cleared the last bit of a register pointed by the file select register (FSR). Thus, different faulty value combinations were found when the FSR pointed to different addresses as shown in the last two rows of Table 4.5.

Table 4.6 presents the hamming distance between the Tgt_inst and the Flt_inst of test program two. When the literal value is changing, the hamming distance between the $MOVLW$ instruction and the NOP instruction also changes. Hence, in some cases, the hamming distance is increased. However, there are some opcodes that are undefined for the PIC16F687. The opcodes with bit[13:7] equals to "0000000" have 128 possible encodings.

Table 4.5: Fault injection result on test program two

D_{t2p} (ns)	Target instruction cycle	Fault Value (hex)@addr			Targeting instruction	Flt_inst
		0x20	0x21	0x23		
No EM pulse	N/A	0x55	0x03	0x58	N/A	N/A
184	3	0x20	0x03	0x23	$MOVLW$ $0x55$	NOP
254	4	0xAA	0x03	0xAD	$MOVWF$ $0x20$	NOP
334	5	0x55	0x55	0xAA	$MOVLW$ $0x03$	NOP
414	6	0x55	0xAA	0xFF	$MOVWF$ $0x21$	NOP
644 (FSR = 0x20)	9	0x54	0x03	0x57	$MOVLW$ $0x00$	BCF $INDF$, 0
644 (FSR = 0x21)	9	0x55	0x02	0x57	$MOVLW$ $0x00$	BCF $INDF$, 0

Table 4.6: Hamming distance between Tgt_inst and Flt_inst of test program two

Tgt_inst	opcode	Flt_inst	opcode	Hamming distance
$MOVLW$ $0x55$	1100xx0 1010101	NOP	0000000 xx00000	5
$MOVWF$ $0x20$	0000001 0100000	NOP	0000000 xx00000	1
$MOVLW$ $0x03$	1100xx0 0000011	NOP	0000000 xx00000	4
$MOVWF$ $0x21$	0000001 0100001	NOP	0000001 xx00000	2
$MOVLW$ $0x00$	1100xx0 0000000	BCF $INDF$, 0	0100000 0000000	1

Only 8 out of these 128 encodings are defined as shown in Table 4.7. The datasheet does not specify what happens if an undefined instruction occurs and there is no exception handler in PIC16F687 [11]. Thus, all the undefined instructions are probably executed as NOP.

Table 4.7: Instructions defined with opcode bit[13:7] = 0000000

Instruction	Description	opcode bit[6:0]
NOP	No operation	xx00000
$CLRWDT$	Clear watchdog timer	1100100
$RETFIE$	Return from interrupt	0001001
$RETURN$	Return from subroutine	0001000
$SLEEP$	Go into standby mode	1100011

Hence, Table 4.6 could be reconstructed where the opcode of the NOP instruction is replaced with an undefined instruction and the hamming distance could be even smaller. Table 4.8 presents the new hamming distance between the Tgt_inst and the Flt_inst. After considering all the undefined instruction as the NOP instruction, the new hamming distance between the $MOVLW$ instruction and the NOP instruction is reduced.

Table 4.8: Reconstruct Table 4.6 with undefined instruction executed as NOP

Tgt_inst	opcode	Flt_inst	opcode	Hamming distance
$MOVLW$ 0x55	1100xx0 1010101	NOP	0000000 1010101	2
$MOVWF$ 0x20	0000001 0100000	NOP	0000000 0100000	1
$MOVLW$ 0x03	1100xx0 0000011	NOP	0000000 0000011	2
$MOVWF$ 0x21	0000001 0100001	NOP	0000001 0100001	1
$MOVLW$ 0x00	1100xx0 0000000	$BCF\ INDF$, 0	0100000 0000000	1

4.4.3 Test Program Three

In addition to some abnormal memory operation behaviors, fault injection could also lead to a jump behavior. In this section, an example is given to show that the EM pulse could cause the program to unexpectedly jump to other addresses.

```
1   MOVLW 0x20
2   MOVWF PORTC ; Assert the trigger
3   N*NOPs
4   MOVLW 0x55
5   MOVWF 0x20
6   MOVLW 0x80    ; Target instruction
7   MOVWF PORTC ; De-assert the trigger
8   MOVF 0x20, W;
9   ADDWF 0x22, F;
10  MOVF 0x21, W;
11  ADDWF 0x22, F;
12  ......
13  MOVLW 0x20; Starting address : 0x20
14  MOVWF FSR ; Transfer data from SRAM to EEPROM;
15 LOOP    ; Starting writing to EEPROM
16  BCF STATUS, RP0
17  BCF STATUS, RP1
18  MOVF INDF, W  ; The address for this instruction is 0x80
19  ......
20  GOTO LOOP
```

Listing 4.4: Test program three

Test program three shows a case where a $MOVLW$ instruction could be replaced with a $GOTO$ instruction or a $MOVWF$ instruction. Initially, 0x21 was written to the FSR and N was set to 5. The value 0x55 was written to SRAM at address 0x20. The data at address 0x20 and address 0x21 was added together and stored at address 0x22. If no fault is injected, the expected values at these three locations are shown in the first row of Table 4.9.

Table 4.9: Fault injection result on test program three targeting $MOVLW$

Test program	D_{t2p} (ns)	Fault Value (hex)@addr			Flt_inst
		0x20	0x21	0x22	
Three	No EM pulse	0x55	0x00	0x55	N/A
Three ($N = 5$)	484	0x55	0x55	0xAA	$MOVWF\ INDF$
	484	0xFF	0x00	0x00	$GOTO$ 0x80
Three (with four more NOPs)	484	0x55	0x00	0x00	$GOTO$ 0x80
	484	0x55	0x55	0xAA	$MOVWF\ INDF$
Three (with FSR set to 0x20)	484	0x55	0x00	0x00	$GOTO$ 0x80
Three ($N = 3$)	334	0x54	0x00	0x54	$BCF\ INDF$, 0

After injecting the fault targeting the P_{cyl} of the $MOVLW$ 0x80 instruction, two groups of faulty value combinations were obtained. The first set of faulty values is shown in the third row of Table 4.9. A faulty 0x55 was written to address 0x21. Therefore, the $MOVLW$ 0x80 was likely replaced with the $MOVWF\ INDF$. The hamming distance between this group of instructions is 2 as shown in Table 4.10. To verify this, the FSR was changed to 0x20. This type of fault could not be detected because even though the $MOVLW$ 0x80 was replaced with the $MOVWF\ INDF$, the value stored at address 0x20 was not changed. Only when the $MOVLW$ 0x80 was replaced with the $GOTO$ 0x80, the fault could be detected by the program as shown in the seventh row.

The second group is shown in the fourth row of Table 4.9. From reading the EEPROM, the faulty value stored at address 0x20 was 0xFF. The default value for the EEPROM was 0xFF. Hence, it was likely that the program skipped writing to the EEPROM at address 0x20. Actually, the address of the $MOVF\ INDF$, W instruction (at line 18 of listing 4.4) is 0x80 and when the Flt_inst was the $GOTO$ 0x80, the $MOVLW$ 0x20 and the $MOVWF\ FSR$ instruction at line 13 and 14 were skipped. Hence, the EEPROM write loop started from address 0x21. Consequently, the faulty value at address 0x20 was 0xFF. The hamming distance between the $MOVLW$ 0x80 and the $GOTO$ 0x80 is only 2 as shown in Table 4.10.

Table 4.10: Hamming distance between Tgt_inst and Flt_inst of test program three

Tgt_inst	opcode	Flt_inst	opcode	Hamming distance
		$MOVWF\ INDF$	0000001 0000000	2
$MOVLW$ 0x80	1100xx1 0000000	$GOTO$ 0x80	1010001 0000000	2
		$BCF\ INDF$, 0	0100001 0000000	1

In order to verify this, four more NOPs were added to change the address of the

instruction $MOVLW$ $0x20$ to 0x80. Then, if the $MOVLW$ $0x80$ is replaced with the $GOTO$ $0x80$, it is expected that no fault occurs at address 0x20 and 0x21. However, the sum at address 0x22 should be 0x00 since all the following instructions are skipped due to the faulty $GOTO$ $0x80$. The fourth row in Table 4.9 shows that the $MOVLW$ $0x80$ was indeed replaced with the $GOTO$ $0x80$. By adding more NOPs inside the loop, it was also verified that by changing the k of the $MOVLW$ k instruction, the associated address k in the faulty instruction $GOTO$ k was changed accordingly.

When two NOPs were removed and N was set to 3, it was found that the Tgt_inst was only replaced with the $BCF\ INDF$, 0. The faulty value is shown in the last row of Table 4.9. Since the BPS system could only change the D_{t2p} by 10 ns per step, the D_{p22} (D_{p2n} with $n\ = 2$) of the P_{cyl} of the $MOVLW$ $0x80$ was changed from 16 ns to 12.15 ns. This might cause different bits to be flipped in the opcode, which results in a different Flt_inst.

From test program three, the Flt_inst could be different with the same experimental setup. Moreover, by changing D_{p22} of the P_{cyl} of the Tgt_inst, the injected fault varied. Therefore, it is important to study the statistical result of each experiment. The fault injection experiment was performed on test program three for 340 rounds for each different cases. The statistical result is shown in Table 4.11. In over 98% of the tests, the fault was successfully induced to both test programs. When N was set to 3, the only faulty instruction was the $BCF\ INDF$, 0. When N was set to 5, 42% of the Flt_inst was the $GOTO$ $0x80$ and the occurrence was slightly increased to 56% when the Flt_inst was the $MOVWF\ INDF$.

Table 4.11: Statistical result of the fault injected on test program three targeting $MOVLW$

Test program	D_{p22} (ns)	Flt_inst	Occurrence
Three (N=5)	16	$MOVWF\ INDF$	56.4%
		$GOTO$ $0x80$	42.3%
Three (N=3)	12.15	$BCF\ INDF$, 0	98.4%

4.5 The $XORWF$ Instruction on Chip #2

The $XORWF$ instruction is used to implement the XOR function which is widely used in cryptographic algorithms such as AES. Microchip released a software AES implementation in their application note [17]. By investigating this code, the $XORWF$ instruction is utilized to finish the key addition in each round of AES. Therefore, it is important to analyze the faulty behavior of this instruction. In this section, the faulty response of the

$XORWF$ instruction will be provided. After the analysis, a physically implemented AES-128 attack is discussed. All the experiments in this section were done with $V_p = 500\ V$, $V_{DD} = 5\ V$, $F_{clk} = 52\ MHz$.

The faulty response of the $XORWF$ instruction was analyzed with a test program and the experimental results are presented in appendix B.3. From the experimental results, the $XORWF\ 0x20,\ F$ instruction could be faulted and was replaced with one of four different faulty instructions, specifically NOP, $IORWF\ 0x20,\ W$, $IORWF\ 0x20,\ F$, and $XORWF\ 0x20,\ W$. Therefore, it is possible that we could inject a fault to the software implementation of AES which could generate a faulty ciphertext. With the faulty ciphertext, it may be possible to retrieve the key. In the next section, the attack on AES, targeting the $XORWF$ instruction, is presented.

4.5.1 AES Attack Targeting the $XORWF$ Instruction

In this section, the Microchip software implementation of AES-128 [17] was utilized as our target program. The attack focused on the last round operation of AES. The attack was a ciphertext-only attack. However, the attacker must be able to record both the correct and the faulty ciphertext. The notations used for presenting the attack on AES-128 are shown in Table 4.12. The program calculates the round key on the fly. Since PIC16F687 is an 8-bit microcontroller, the 16-byte key is stored from address 0x30 to 0x3F (where an address is represented as 0x3m, $m \in [0\ :\ F]$). The intermediate data is stored from address 0x20 to 0x2F. After finishing the $ARK_{10}()$, the ciphertext is output from address 0x20 to 0x2F. There are two steps to finish the $ARK_i()$ algorithm as shown in Figure 4.16. In step ①, the key in address 0x3m is moved to the W register by using the $MOVF\ 0x3m,\ W$ instruction. By using the $XORWF\ 0x2m,\ F$ instruction, the key is XORed with the intermediate state and the result is stored back to address 0x2m in step ②.

Figure 4.16: Instructions used in ARK_i of software implementation of AES, $m \in [0\ :\ F]$

The ciphertext C^m is generated and stored at address 0x2m after $ARK_{10}()$. Therefore, if the $XORWF\ 0x2m,\ F$ instruction is replaced with a NOP instruction by injecting a fault, LS^m could be output directly as the faulty ciphertext $\widehat{C^m}$. Hence, the associated key byte K_{10}^m could be easily retrieved by calculating $\widehat{C^m} \oplus C^m$. With this finding, the complete AES-128 attack will be presented in two steps.

Table 4.12: Notations for AES targeting $XORWF$ instruction

Notation	Description
LS^i	$i-th$ byte of the last state before applying the $ARK_{10}()$
K_i^{j*}	$j-th$ byte of the $i-th$ round key candidate
K_i^j	$j-th$ byte of the $i-th$ round key
K_i	The $i-th$ round key
C^i	$i-th$ byte of the ciphertext
$\widehat{C^i}$	$i-th$ byte of the faulty ciphertext
$SB_i()$	The sbox operation in $i-th$ round
$MC_i()$	The mix-column operation in $i-th$ round
$SR_i()$	The shift-row operation in $i-th$ round
$ARK_i()$	The add round key operation in $i-th$ round

4.5.1.1 Step 1: Characterize the Flt_inst

It was necessary to check the faulty response of the $XORWF$ instruction in the AES-128 program before running the attack. The attack was conducted based on the assumption that the attacker knows everything except the key. Hence, it was reasonable to artificially insert a trigger signal to help deliver the EM pulse over the $XORWF$ instruction as a proof of concept. This trigger signal was activated only during the last round. The $ARK_i()$ part for the AES-128 program is shown in listing 4.5. When the $round_counter$ is 1, the program is executing the last round and a trigger signal is activated. The key and the plaintext were set to a known value in order to calculate the 16 bytes of the last round key K_{10} and the last state LS. The experiment was conducted for 340 rounds with D_{t2p} set to 254 ns.

```
1  key_addition:
2      ...
3    MOVLW    0x1
4    SUBWF    round_counter, w
5    BTFSS    STATUS, Z ; Last round?
6    GOTO     no_trigger ; Not last round
7    BCF      STATUS, RP0 ; Last round
8    BCF      STATUS, RP1
9    MOVLW    b'00100000'
10   MOVWF    PORTC
11 no_trigger:
12   3*NOPs
13   MOVF   0x30,W        ; block[0] ^= key[0];
14   XORWF  0x20,F  ; Target instruction
15   NOP;
16   NOP;
```

```
17 MOVLW    b'00000000'
18 MOVWF    PORTC ; de−assert the trigger
```

Listing 4.5: Revised ARK_i part for AES-128

The fault injection experiment indicated that the faulty response was different from the $XORWF$ test program even though the D_{p22} of the P_{cyl} of the $XORWF$ was the same as 15.23 ns. The Flt_inst and the occurrence is shown in Table 4.13. Since the fault was induced to the P_{cyl} of the Tgt_inst, the address of the Tgt_inst might also affect the bit flipped during loading the instruction from the FLASH memory to the instruction register. The address of each instruction also changed the physical location which stored the instruction. Therefore, by moving the probe tip to a new position, it might be helpful to increase the possibility of replacing the $XORWF$ instruction with a NOP or a $MOVWF$ instruction which would output the last state LS or the last round key K_{10}, respectively as the faulty ciphertext.

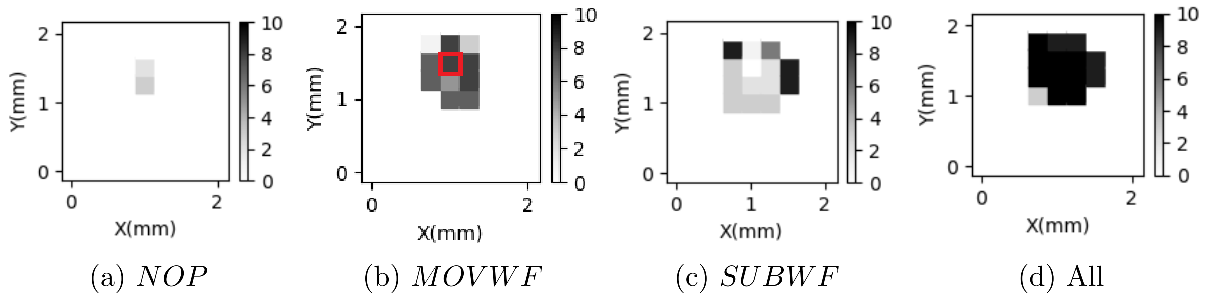
Table 4.13: Fault injection result on AES-128 code targeting the $XORWF$ instruction

Tgt_inst/Opcode	Flt_inst	Opcode	Hamming distance	Occurrence
$XORWF$ 0x20, F/	$SUBWF$ 0x20, F	0000101 0100000	1	65.8%
0001101 0100000	$MOVWF$ 0x20	0000001 0100000	2	31.4%

Based on this hypothesis, a new scan was performed with the AES-128 program where the $ARK_i()$ was modified to assert a trigger signal at the last round. The result of the scan is shown in Figure 4.17. Since it is easier to retrieve the last round key K_{10} when the Flt_inst is a $MOVWF$ or a NOP, the position for the probe was chosen as the one with a red rectangle shown in Figure 4.17b. At this position, for 8 out of 10 rounds, the Flt_inst was the $MOVWF$ instruction.

4.5.1.2 Step 2: Build The Attack Algorithm

With a location where the Tgt_inst was replaced with the desirable Flt_inst with a high probability, the attack algorithm 1-a was built. The pseudo code for the attack algorithm 1-a is shown in listing 4.6. By using random plaintexts, the last state LS is expected to be different with the same key. Only if the $XORWF$ instruction is replaced with the $MOVWF$ instruction, the faulty ciphertext would be consistent which is the associated last round key byte ${K_{10}^{i}}^*$. If the faulty ciphertext byte is not in the $Candidate_i$ list, either the current faulty ciphertext byte or all the previous faulty ciphertext bytes are resulted from a different Flt_inst other than the $MOVWF$ instruction.



(a) NOP (b) $MOVWF$ (c) $SUBWF$ (d) All

Figure 4.17: Shmoo plot for number of faults injected with Flt_inst as (a): NOP, (b): $MOVWF$, (c): $SUBWF$ and (d): all injected fault.

A single fault injection attempt is done with each byte where the ${K_{10}^{i}}^*$ is still not revealed. Whenever the ${K_{10}^{i}}^*$ is found, the associated index is removed from the index list since no more tests are required for this byte. Therefore, for the first two plaintexts, all the 16 bytes are tested. Afterwards, only the unrevealed key byte is tested with fault injection.

After all the 16 key bytes are found, the initial key is calculated by a reverse key scheduling algorithm. If this ${K_0}^*$ is the same with the key programmed to the chip, the attack is successful.

```
1  success = 0 # Total number of keys that have been successfully retrieved
2  total = 0 # Total number of keys that have been tested
3  while (1):
4      K_0 = random(key) # Randomly generate a 16 byte initial key
5      for i in range(16):
6          Candidate_i =[] # Setup an empty list for each byte of key candidate
7      index = range(16) # The index for 16 bytes key
8      while (len(index) > 0): # if there remains unknown key byte
9          P = random(plain)# Randomly generate a plaintext
10         C = Enc(P, K_0) # The first round of encryption will generate the
       correct ciphertext
11         for i in index:
12             program(trigger_i) # Program the chip where the trigger is
       targeting the i-th byte of ARK_10
13             pulse.on
14             time.sleep(12)
15             pulse.off
16             if Ĉ^i != C^i: # If a fault was injected
17                 if Ĉ^i not in Candidate_i:
18                     Candidate_i.append(Ĉ^i)
19                 else:
```

```
20                    K_10^i* = Ĉ^i
21                    index.remove(i)
22    K_0* = LastToInitial(K_10^0* to K_10^15*) # Use the 16 bytes last round key to
      calculate the initial key
23    if K_0* == K_0;
24        success = success + 1
25    total = total + 1 # Total number of keys that have been tested
```

Listing 4.6: AES-128 attack algorithm 1-a targeting $XORWF$

16 different keys were tested and the result is shown in Table 4.14. The attack required an average of around 9 plaintexts to retrieve all the key bytes. During the tests, the induced fault did not always cause a Flt_inst as $MOVWF$. In some cases, the Flt_inst was actually a $SUBWF$ or a NOP instruction where the algorithm could not output the associated K_{10}^i for the current plaintext. Hence, it increased the number of plaintexts required to get all the key bytes.

Table 4.14: AES-128 attack result with the attack algorithm 1-a targeting $XORWF$

Total test		16
Successful test		16
Number of plaintexts	Avg	8.88
	Std	2.25
Number of EM pulses	Avg	307.6
	Std	63.9

In fact, if the Flt_inst is a NOP instruction, the output is the last state LS and the last key byte could also be found. Therefore, the attack algorithm 1-b was considered where both the $\widehat{C^i}$ and the $\widehat{C^i} \oplus C^i$ are added to the associated list $Candidate_i$ (see line 18 of listing 4.6). If either the $\widehat{C^i}$ or the $\widehat{C^i} \oplus C^i$ is in the list $Candidate_i$, it is suggested as the correct key byte. Therefore, this attack algorithm considers both cases where the Flt_inst is either a $MOVWF$ or a NOP instruction. In this way, the number of plaintexts might be reduced to increase the attack efficiency. The result of this attack is shown in Table 4.15.

From Table 4.15, the number of plaintexts required before outputting the 16 bytes $K_0{}^*$ was reduced to 5.3. However, the success rate was reduced from 100% to 68.75%. The reason for the failed tests was investigated. During the $m-th$ test with the $m-th$ plaintext for the $i-th$ byte, the Flt_inst was an $INST_A$ and both $\widehat{C^i}$ and $\widehat{C^i} \oplus C^i$ were not in the list $Candidate_i$. At the $n-th$ test with the $n-th$ plaintext for the $i-th$ byte, the

Table 4.15: AES-128 attack result with the attack algorithm 1-b targeting $XORWF$

Total test		16
Successful test		11
Number of plaintexts	Avg	5.31
	Std	1.30
Number of EM pulses	Avg	222.3
	Std	80.9

Flt_inst was an $INST_B$, either $\widehat{C^i}$ or $\widehat{C^i} \oplus C^i$ was collided with the result of the $m-th$ test. A wrong key guess was output as ${K_{10}^i}^* = \widehat{C^i}$ or ${K_{10}^i}^* = \widehat{C^i} \oplus C^i$.

Since both the $\widehat{C^i}$ and $\widehat{C^i} \oplus C^i$ will be appended to the associated list $Candidate_i$ in each fault injection attempt, the list stored 2 × number of plaintexts elements. For example, for key byte 0, if 5 plaintexts has been tested but ${K_{10}^0}^*$ has not been output yet, list $Candidate_0$ have 10 elements in it. Since there are only at most 256 values for each key byte, the possibility for a wrong key byte guess increases with the length of the list.

4.6 The $DECFSZ$ Instruction on Chip #2

The $DECFSZ$ instruction is used to decrement a register and check if it reaches 0. The next instruction is skipped if the result is 0. Therefore, it is most likely used when a program needs to branch to a different address based on the result of decrementing a register. Hence, the $DECFSZ$ instruction could possibly be used in the program shown in Figure 1.2 to determine the number of trials left for the user to input the pin. An attacker could possibly inject the fault to the $DECFSZ$ instruction, which might allow the attacker to unrestrictedly input the pin. This kind of attack is referred as the bypass security check attack.

Additionally, the previous AES-128 attack targeting the $XORWF$ instruction requires retrieving the last round key byte by byte. By carefully investigating the software implementation of the AES-128, the $DECFSZ$ instruction is utilized to check if the encryption reaches the last round and if the mix-column operation should be omitted. The assembly code of this part is shown in listing 4.7. Before the last round, the round counter has been decremented to 1. During the last round, the $DECF$ *round_counter*, W instruction decrements the round counter to 0 but stores the result in the W register. If the zero-flag is set, the program executes the *GOTO last_round* instruction and skips the mix-column operation. After that, the $DECFSZ$ *round_counter*, F instruction decrements

the round counter again and stores the result back to the *round_counter* register. The *GOTO loop_encrypt* instruction is skipped and the encryption subroutine is finished with the *RETURN* instruction.

```
encrypt:
  MOVLW 0x01
  MOVWF rcon        ; RCON initialization
  CALL  key_addition     ; initial key addition
  MOVLW 0x0A     ; start the round counter for the loop
  MOVWF round_counter    ;
loop_encrypt:
  CALL  substitution_S     ; S-box layer
  CALL  enc_shift_row  ; Shift-row layer
  DECF  round_counter,W ; Last round?
  BTFSC STATUS,Z  ; Check the zero-flag
  GOTO  last_round  ; Yes, no mixcolumn
  CALL  mix_column; No, perform the mix-column
last_round:
  CALL  enc_key_schedule  ; Key scheduling
  CALL  key_addition     ; key addition
  DECFSZ  round_counter,F  ; Target instruction, last round?
  GOTO  loop_encrypt  ; No, conduct the next round operations
  RETURN
```

Listing 4.7: Assembly code for AES-128 encryption subroutine [17]

If the $DECFSZ$ *round_counter*, F is replaced with a NOP instruction, the AES-128 executes an additional round without mix-column operation. Assume this additional round is the 11-th round and the generated faulty ciphertext is $\widehat{C}$. Then we have

$$ARK_{11}(SR_{11}(SB_{11}(C))) = \widehat{C} \tag{4.1}$$

$$K_{11} = \widehat{C} \oplus SR_{11}(SB_{11}(C)) \tag{4.2}$$

Therefore, with one single fault injection, the whole 16 bytes key K_{11} could be retrieved. Consequently, the K_0 could be recovered using a reverse key scheduling algorithm. Compared with the AES-128 attack in Section 4.5.1 where the key byte is retrieved one by one, the new attack would likely require fewer EM pulses to get the whole key.

In this section, we present the analysis and the attacks by targeting the $DECFSZ$ instruction. First, a test program was built to understand the Flt_insts. The experimental results are shown in appendix B.4. The possible Flt_insts for the test program are NOP, $MOVWF$, $INCF\ 0x20,\ F$. Then, an assembly program was designed to mimic the behavior of the code shown in Figure 1.2. A proof of concept attack was implemented to

show that EM FIA could allow the attacker to guess the pin an unlimited number of times. Finally, the AES-128 attack is presented with a two-step methodology.

4.6.1 Bypass security checks attack

The assembly program shown in listing 4.8 was designed to work as the target of the attack. Initially, 0x09 is stored at address 0x20. Assume the attacker inputs a guess of the pin once and the pin is always wrong in each iteration of the $INPUT_PIN$ loop. The attacker could at most try a input guess ten times and the program jumps into a $LOCKED_LOOP$. Each time the attacker inputs the pin, an additional counter would increment by one. Therefore, if no fault is injected, the counter should store 0x0A. With this design, the number of EM pulses delivered could be used to represent how many times the attacker inputs a pin.

```
1  INPUT_PIN:
2   MOVLW b'00100000'
3   MOVWF PORTC ; Assert the trigger
4   NOP
5   NOP
6   NOP
7   NOP ;(Assume the attacker input pin within these four NOPs)
8   DECFSZ 0x20, F  ; Target instruction
9   GOTO NOT_ZERO
10  GOTO LOCKED_LOOP
11  NOT_ZERO:
12  ...  ; De-assert the trigger here
13  GOTO INPUT_PIN
```

Listing 4.8: The assembly program to mimic the code shown in 1.2

The experiment was performed for 500 tests with D_{t2p} set as 254 ns (D_{p22} of the P_{cyl} of the $DECFSZ$ instruction was 15.23 ns). In each test, the EM pulse was delivered for approximately 2.5 seconds (with an approximate 13 ms error since the control script was implemented in python). Since the trigger signal was asserted approximately every 120 us, the maximum number of EM pulses delivered over 2.5 seconds should be 20833 +/- 108.

The result of the attack is shown in Table 4.16 where N_{EM} denotes the number of EM pulses delivered during a single round. There are two main findings:

1. The attack was not successful for 46.4% percent of the tests. During these tests, either no fault could be injected within the first 10 rounds or the faulty instruction

did not lead to a significant increase of the data stored at address 0x20. For example, if the Flt_inst was a NOP, the attacker could only guess the pin once more.

2. When N_{EM} was $\geq$ 20833, it indicated that the attacker could guess the pin an unlimited number of times within this 2.5 seconds. These tests were considered as successful attacks. Hence, the attack was successful with a 31.2% rate. In fact, if the Flt_inst is a $MOVWF$, it sets the data at address 0x20 to be 0x20. The program has another 32 rounds to output the trigger signal (The attacker then has another 32 opportunities to input a pin guess). Within these 32 rounds, if the Flt_inst is a $MOVWF$ for one of the round, it refreshes the data stored at address 0x20 again. Consequently, the attacker could input the pin guess for an unlimited number of times.

Table 4.16: Experimental result of the bypass security check test

	Percentage	Average	Standard deviation
$N_{EM} \geq 20833$	31.2%	20934	28.7
$N_{EM} \leq 15$	46.4%	11	1.1
$16 \leq N_{EM} \leq 20832$	22.4%	1162	1849

4.6.2 The AES Attack Targeting the $DECFSZ$ Instruction

In this section, a round addition attack was performed by targeting the $DECFSZ$ instruction used in the software implementation of AES-128. The attack was a ciphertext-only attack since only the ciphertext/faulty ciphertext were required. Also, the attack was based on the assumption that the attacker knows everything except the key. A trigger signal was inserted to help deliver the EM pulse to the associated Tgt_inst. The encryption subroutine was revised as shown in listing 4.9. The trigger signal was only activated during the last round when the $round_counter$ was 1. Similar to the attack described in Section 4.5.1, the attack was implemented in two steps.

```
1 encrypt:
2   MOVLW 0x01
3   MOVWF rcon        ; RCON initialization
4   CALL  key_addition    ; initial key addition
5   MOVLW 0x0A     ; start the round counter for the loop
6   MOVWF round_counter    ;
7 loop_encrypt:
8   CALL  substitution_S     ;
```

```
 9   CALL   enc_shift_row
10   DECF   round_counter,W
11   BTFSC STATUS,Z
12   GOTO   last_round
13   CALL   mix_column
14 last_round:
15   CALL   enc_key_schedule
16   CALL   key_addition     ; key addition
17   MOVLW    0x01
18   SUBWF    round_counter, w ;
19   BTFSS    STATUS, Z      ; Last round?
20   GOTO     no_trigger ; No, trigger is not activated
21   BCF      STATUS, RP0
22   BCF      STATUS, RP1
23   MOVLW    b'00100000'
24   MOVWF    PORTC ; Activate the trigger
25 no_trigger:
26      4*NOPs
27   DECFSZ   round_counter,F ; Target instruction
28   GOTO   loop_encrypt
29   RETURN
```

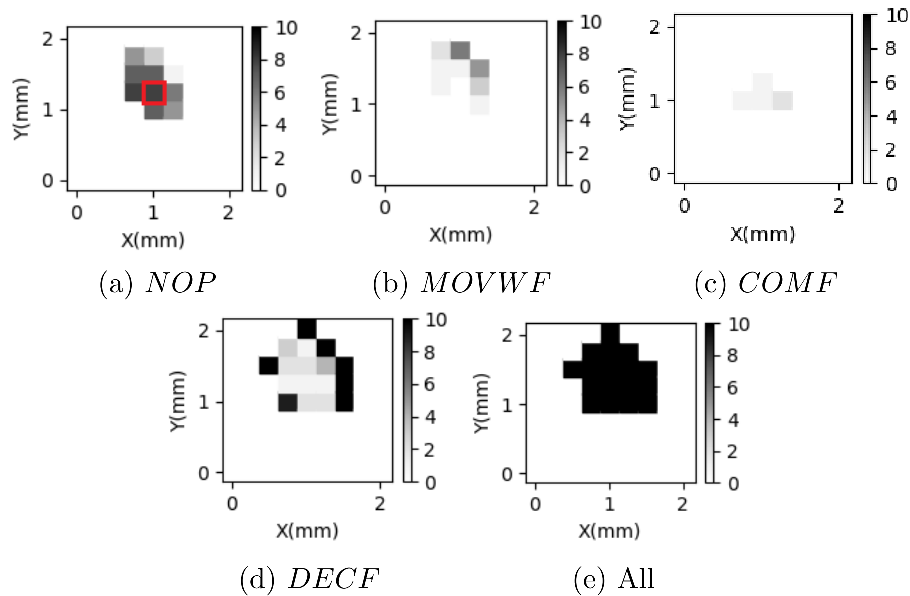
Listing 4.9: The revised encryption subroutine with a trigger signal

4.6.2.1 Step 1: Characterize the Flt_inst

The first step of the attack is to identify a probe location where the NOP instruction is most likely to be the Flt_inst. The scan was performed exactly the same as described in Section 4.5.1.1 with the associated faulty instructions changed. The D_{t2p} was set to 254 ns and the D_{p22} of the P_{cyl} of the $DECFSZ$ was 15.23 ns. The Flt_insts by targeting the $DECFSZ$ with test program five (see appendix B.4) and program in listing 4.9 were different even though the D_{p22} was the same. The associated Flt_insts were NOP, $MOVWF$, $DECF$ and $COMF$. The result of the scan is shown in Figure 4.18. Since the NOP instruction is the desired Flt_inst, the probe was moved to the location indicated by the red rectangle in Figure 4.18a. At this location, the Flt_inst was the NOP instruction for 8 out of 10 rounds.

4.6.2.2 Step 2: Build the Attack Algorithm

After finding the best location, the attack algorithm 2 was built as shown in listing 4.10. In this attack algorithm, the plaintext is fixed before the current attack is successful or



(a) *NOP* (b) *MOVWF* (c) *COMF*

(d) *DECF* (e) All

Figure 4.18: Shmoo plot for the number of faults injected with *Flt_inst* as (a): *NOP*, (b): *MOVWF*, (c): *COMF* and (d): *DECF* (e): all injected fault.

the number of attack attempts is over 50. Whenever the fault is injected, the possible ${K_{11}}^*$ is calculated and a reverse key scheduling algorithm is used to compute the initial key candidate ${K_0}^*$. If the number of attack attempts for the current key is over 50, this attack will be identified as failed and the attack will move to the next randomly generated key K_0.

```
1  success = 0 # Total number of keys that have been successfully retrieved
2  total = 0 # Total number of keys that have been tested
3  while (1):
4      K_0 = random(key) # Randomly generate a 16 byte key
5      succ_flag = 0
6      P = random(plain)# Randomly generate a plaintext
7      C = Enc(P, K) # The first round of encryption will generate the correct
       ciphertext
8      attack_attempt_cnt = 0 # Number of attacks had been tried
9      while (succ_flag == 0) and (attack_attempt_cnt < 50): # if the attack
       hasn't been successful
10         program() # Program the chip
11         pulse.on
12         time.sleep(12)
13         pulse.off
```

```
14          attack_attempt_cnt = attack_attempt_cnt + 1
15          if Ĉ^i != C^i: # If fault was injected
16              K11* = SR(SB(C^i)) ⊕ Ĉ^i
17          K0* = AdditionToInitial(K11*) #
18          if K0* == K0;
19              success = success + 1
20              succ_flag = 1
21      total = total + 1 # Total number of keys
```

Listing 4.10: The AES-128 attack algorithm 2 targeting $DECFSZ$

Table 4.17: AES-128 attack result with the attack algorithm 2 targeting $DECFSZ$

Total test		100
Successful test		100
Number of attack attempts	Avg	1.48
	Std	0.86
Number of EM pulses	Avg	7.05
	Std	6.06

The result of the attack is shown in Table 4.17. The attack was performed on 100 different keys and all the keys were successfully revealed. Moreover, the average number of EM pulses before finding the key was slightly over 7. Therefore, the attack is significantly more efficient than the attack based on algorithm 1-a and 1-b where the $XORWF$ was the Tgt_inst. The average number of attack attempts was 1.48. This indicated that around two out of three attack attempts, the Flt_inst was the NOP instruction.

4.7 Extend The AES-128 Attack To Chip #3

To make sure the attack is general, robust and reproducible, the attack was performed on chip #3 (which was also decapsulated from the backside). Due to process variation, chip #3 could run with a higher clock frequency at 61 MHz. Chip #3 was placed on the board after removing chip #2. Since the relative position between the probe tip and the silicon die could only be estimated (see Section 3.3), the faulty instructions are possibly different due to this positioning error and low scanning resolution during characterization of the Flt_inst.

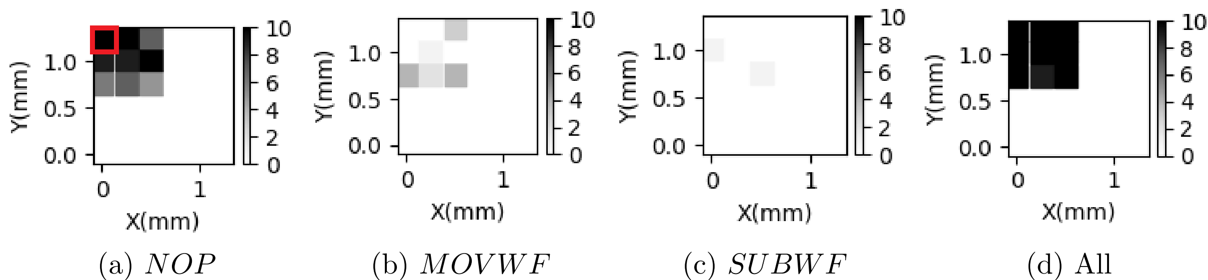
4.7.1 The AES-128 Attack Targeting the $XORWF$

The F_{clk} was set as 58.521 MHz and the D_{t2p} was set to 224 ns. Thus, with test program in listing 4.5, D_{p22} of the P_{cyl} of the $XORWF$ was still 15.23 ns.

4.7.1.1 Step 1: Characterize the Flt_inst

To identify the location where the Flt_inst is most likely to be a $MOVWF$ instruction or a NOP instruction, a similar scan was conducted again. However, from the result of the scan on chip #2, the sensitive area where the fault could be injected was roughly over the flash memory on the chip (though this is just an estimate). Therefore, instead of scanning over a total of 81 points, the scan on chip #3 was done with only 36 locations to increase the scan efficiency. These 36 locations formed a 1.6 mm^2 area which approximately covered all the flash memory. The result of the scan is shown in Figure 4.19.

The best attack position is indicated with a red rectangle in Figure 4.19a. At this location, the Flt_inst was always the NOP instruction over 10 rounds. Therefore, the output faulty ciphertext was the last state LS. Hence, the attack algorithm 1-c, which is similar to the attack algorithm 1-a shown in listing 4.6, was proposed. Instead of the $\widehat{C^i}$, the $\widehat{C^i} \oplus C^i$ is appended to the $Candidate_i$ list since this is most likely to be the last round key.



(a) NOP (b) $MOVWF$ (c) $SUBWF$ (d) All

Figure 4.19: Shmoo plot of chip #3 for number of faults injected with Flt_inst as (a): NOP, (b): $MOVWF$, (c): $SUBWF$ and (d): all injected fault.

4.7.1.2 Build the attack algorithm

The attack result with the attack algorithm 1-c is shown in Table 4.18. Compared with the result shown in Table 4.14, the average number of plaintexts required was reduced. At the

probe location on chip #3, the Flt_inst was the NOP instruction for 10 out of 10 rounds whereas for the probe location on chip #2, the Flt_inst was the $MOVWF$ instruction for only 8 out of 10 rounds. Therefore, it was more likely to append a byte which was not the last round key to the $Candidate_i$ list in the first attack. The average number of EM pulses was also reduced since less plaintexts were required.

Table 4.18: AES-128 attack result with the attack algorithm 1-c targeting $XORWF$ on chip #3

Total test		22
Successful test		22
Number of plaintexts	Avg	4.23
	Std	1.38
Number of EM pulses	Avg	129.8
	Std	19.6

The attack algorithm 1-b was also tried on chip #3. The result is shown in Table 4.19. There was only one test where the initial key was failed to be retrieved whereas 5 out of 16 tests were failed for the experiment on chip #2 with the attack algorithm 1-b. Moreover, the average number of plaintexts required was only reduced for 5% compared with the attack with the attack algorithm 1-c. Since the possibility of getting a Flt_inst as NOP was high, the attack efficiency will not be significantly increased by considering that the Flt_inst was either a NOP instruction or a $MOVWF$ instruction.

Table 4.19: AES-128 attack result with the attack algorithm 1-b targeting $XORWF$ on chip #3

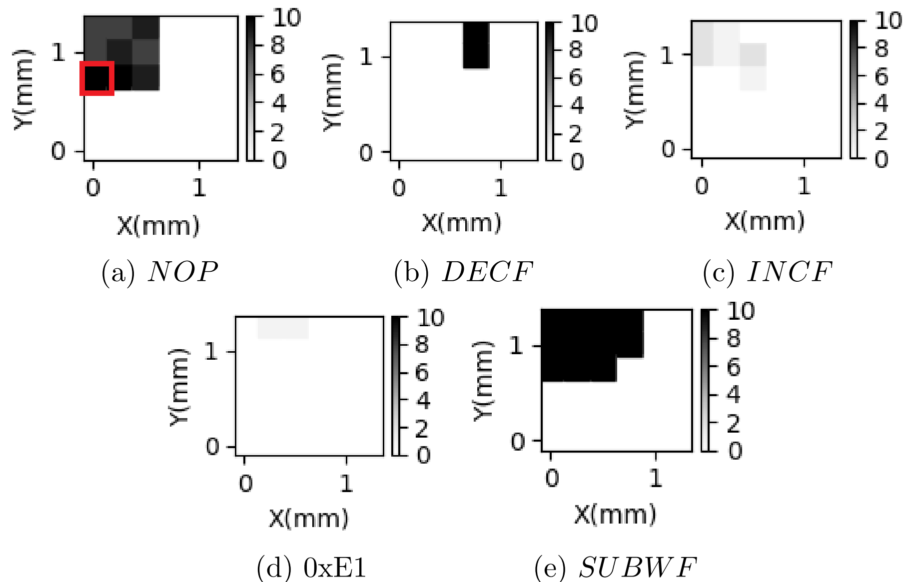
Total test		31
Successful test		30
Number of plaintexts	Avg	4.03
	Std	1.38
Number of EM pulses	Avg	124.7
	Std	24.5

4.7.2 The AES-128 Attack Targeting the $DECFSZ$

The D_{t2p} was also set to 224 ns and F_{clk} was set to 58.521 MHz. The assembly code of the AES-128 encryption subroutine was modified such that a trigger was embedded. The program is shown in listing 4.9. The attack was also based on the two-step approach.

4.7.2.1 Step 1: Characterize the *Flt_inst*

A scan was performed exactly the same as described in Section 4.5.1.1 with the associated faulty instructions changed. Similarly, the scan on chip #3 was only done on 36 locations which formed a square with approximately 1.6 mm^2 area. The result of the scan is demonstrated in Figure 4.20. One of the injected faults resulted in a faulty data 0xE1 in the round counter register. However, we did not find a single instruction which could lead to this result. Therefore, the *Flt_inst* was not found for this faulty data. The best location for performing the attack is indicated with a red rectangle as shown in Figure 4.20a. At this location, the fault was injected in every round and the only *Flt_inst* was the *NOP* instruction.



(a) *NOP* (b) *DECF* (c) *INCF*

(d) 0xE1 (e) *SUBWF*

Figure 4.20: Shmoo plot for number of faults injected with *Flt_inst* as (a): *NOP*, (b): *DECF*, (c): *INCF* and (d): 0xE1 fault (e): all injected fault.

4.7.2.2 Step 2: Build the Attack Algorithm

The attack algorithm was the same as the attack algorithm 2 as shown in listing 4.10. The result of the attack is shown in Table 4.20. It took an average of 8.77 EM pulses to retrieve the key.

Table 4.20: AES-128 attack result with the attack algorithm 2 targeting $DECFSZ$ on chip #3

Total test		120
Successful test		120
Number of attack attempts	Avg	1.64
	Std	1.05
Number of EM pulses	Avg	8.77
	Std	6.89

4.8 Countermeasure

After identifying the Flt_inst for some instructions, it is now possible to propose practical fault tolerant countermeasures and empirically verify them. Four instructions were chosen to study possible countermeasures. These four instructions are $MOVWF$, $MOVLW$, $XORWF$, $DECFSZ$. These instructions cover basic byte-oriented memory operation, literal operation, arithmetic instruction and control instructions. Moreover, one possible Flt_inst when injecting the fault to the $MOVLW$ is the $GOTO$ instruction. Therefore, the jump instruction was also taken into consideration. Since we had no access to the hardware, all the proposed countermeasures were in software by adding extra instructions. The added instructions for the countermeasure are underlined in associated listings.

In each countermeasure designed for a specific instruction, the experiment was performed on chip #2 to verify the efficiency of the countermeasure using the same setup ($F_{clk} = 52\ MHz$ and $V_{DD} = 5\ V$). V_p was set to 500 V to ensure the proposed countermeasure was effective. First, a fault was injected to the associated program without the countermeasure. After adding the countermeasure, the efficiency of the countermeasure was verified by running the program 100 rounds with the same setting. Also, we assume that the fault could not be injected more than once within 13 instructions, which is 13 *4 * 1000/52 = 1 us. This is as discussed in Section 3.3.

4.8.1 Countermeasure for the $MOVWF$ Instruction

The $MOVWF$ instruction is similar to the idempotent instruction introduced in [91]. Since the only Flt_inst for $MOVWF$ empirically observed was NOP, a duplication of the $MOVWF$ instruction would be sufficient. Moreover, duplication of the $MOVWF$ does not change the functionality of the program. The test code is similar to the code shown in listing 4.2 with a duplicated $MOVWF$ $0x20$ instruction. When the duplicated

$MOVWF\ 0x20$ instruction was not added, the fault was injected at address 0x20 when D_{t2p} was set to 254 ns. The faulty data at address 0x20 was 0xAA since the $MOVWF\ 0x20$ instruction was replaced with NOP and the data stored at address 0x20 was unchanged. After adding the duplicated $MOVWF\ INDF$ to the program, 0x55 was correctly stored to address 0x20. For all the 100 rounds, no faulty data was found. Therefore, the proposed countermeasure provides a fault tolerant protection over the $MOVWF$ instruction. The overhead is an additional instruction, which takes 14 bits extra FLASH memory. The latency of the program is also increased by an extra instruction cycle.

4.8.2 Countermeasure for the $MOVLW$ Instruction

In test program three shown in listing 4.4, when N was set to 5, the $MOVLW\ 0x80$ instruction was replaced with the $MOVWF\ INDF$ or the $GOTO\ 0x80$. Both Flt_insts need to be taken into consideration to design the countermeasure. The countermeasure is shown in listing 4.11. The initial value stored at address 0x21 is 0x00 and FSR is set to 0x21.

The countermeasure provides two protections. First, after executing the $MOVLW\ 0x80$ instruction, the previous data stored at address 0x21 is written back to itself again as shown at line 9 and 10. Moreover, since the $MOVLW\ 0x80$ instruction could also be replaced with the $GOTO\ 0x80$ instruction, another $GOTO\ Feedback$ instruction is inserted. The address of this instruction is 0x80. Therefore, if the $MOVLW\ 0x80$ instruction is replaced with the $GOTO\ 0x80$ instruction, the program will jump back to the $MOVLW\ 0x80$ instruction again.

```
1    MOVLW 0x20
2    MOVWF PORTC
3     5*NOPs
4    MOVLW 0x55
5    MOVWF 0x20
6  Feedback:
7    MOVLW 0x80     ; Target instruction
8    MOVWF PORTC
9    MOVLW 0x00
10     MOVWF INDF; Store the initial value back to the File register pointed by
       FSR.
11   MOVF 0x20, W;
12   ADDWF 0x22, F;
13   MOVF 0x21, W;
14   ADDWF 0x22, F;
15    ......
```

```
16   MOVLW 0x20; Starting address : 0x20
17   MOVWF FSR ; Transfer data from SRAM to EEPROM;
18 LOOP    ; Starting writing to EEPROM
19   BCF STATUS, RP0
20   BCF STATUS, RP1
21   GOTO Continue
22  GOTO Feedback ;The address for this instruction is 0x80
23 Continue:
24   MOVF INDF, W
25    ......
26   GOTO LOOP
```

Listing 4.11: Countermeasure for MOVLW instruction

With D_{t2p} set to 484 ns, the experiment was run for 100 rounds and no fault was found in the program. Therefore, this countermeasure successfully thwarted the attack targeting the $MOVLW$ $0x80$ instruction. However, the software designer needs to be careful since the countermeasure is literal dependent. The penalty for this countermeasure is four additional instructions for each $MOVLW$ instruction, which take additional 4*14 bits FLASH memory. The latency is increased by two additional instruction cycles due to the added $MOVLW$ $0x00$ and $MOVWF$ $INDF$ instruction at line 9, 10 in listing 4.11.

To defeat this countermeasure, the attacker has to inject two faults to the Tgt_inst and either the instruction at line 9, 10 or 22, which is impossible since it was assumed the attacker could not inject two faults within 13 instructions.

4.8.3 Countermeasure for the $XORWF$ Instruction

The $XORWF$ instruction is similar to the separable instructions like ADD $R1$, $R1$, $R2$ as described in [91]. However, in PIC16F687, the W register is the only register that can be used to store a temporary variable. Thus, instead of using extra registers to provide the protection, extra data memory is utilized in this case.

The countermeasure was verified using the AES-128 code as shown in listing 4.5. The new code with the countermeasure is shown in listing 4.12. The added instructions for the countermeasures are underlined.

```
1 key_addition:
2      ...
3   MOVLW    0x1
4   SUBWF    round_counter, w
5   BTFSS    STATUS, Z ; Last round?
```

```
6    GOTO     no_trigger ; Not last round
7    BCF      STATUS, RP0 ; Last round
8    BCF      STATUS, RP1
9    MOVLW    b'00100000'
10   MOVWF    PORTC
11 no_trigger:
12   3*NOPs
13   MOVF   0x30,W; W = key[0];
14   XORWF 0x20,F; Target Instruction. Block[0] xor key[0] and stored back to
       address 0x20
15   XORWF 0x60,F;    Duplicate the key_addition using second memory location
16   2*NOPs
17   MOVLW    0x00;
18   MOVWF    PORTC ; de-assert trigger
19   MOVF 0x20, W
20   SUBWF 0x60, W ; check if data at address 0x20 = data at address 0x60
21   BTFSS STATUS, Z; If Zero flag was set, skip the next instruction
22   GOTO   MAIN   ;Restart the encryption
23       .....
24       RETURN
25 store_var:
26       MOVF 0x20, W
27       MOVWF 0x60
28       ...
29       RETURN
30 encrypt:
31   MOVLW 0x01
32   MOVWF rcon        ; RCON initialization
33   CALL      store_var   ;
34   CALL   key_addition     ; initial key addition
35   MOVLW ROUNDS        ; start the round counter for the loop
36   MOVWF round_counter
37 loop_encrypt:
38   CALL   substitution_S     ; substitution layer
39   CALL   enc_shift_row    ; shift_row layer
40   CALL      store_var         ; Copy the intermediate state
41   DECF   round_counter,w
42   BTFSC STATUS,Z
43   GOTO   last_round
44   CALL   mix_column   ; mix_column layer
45   CALL      store_var    ; Copy the intermediate state
46 last_round:
47   CALL   enc_key_schedule
48   CALL   key_addition     ; key addition
49   DECFSZ  round_counter,f
```

```
50  GOTO    loop_encrypt
51  RETURN
```

Listing 4.12: Countermeasure for protecting the XORWF instruction

The idea of this countermeasure is to do the key addition twice, but with two memory locations. The memory locations from address 0x60 to 0x6F are used to store the intermediate state inside the AES-128 encryption. Therefore, during the encryption, the intermediate value must be copied each time before executing the *key_addition* subroutine. The *store_var* subroutine is used to copy the intermediate state to the memory addressed from 0x60 to 0x6F.

The experiment was conducted for 100 rounds and no faulty ciphertext was generated. Therefore, this countermeasure provides a fault tolerant protection to the $XORWF$ instruction. However, it requires additional memory to store the intermediate state before each $ARK()$ layer. Also, it requires checking every byte of the intermediate state after the $ARK()$. The data storage overhead is another 16 bytes data memory. The code storage overhead is 41*14 bits more FLASH memory. Each *store_var* subroutine takes 34 instruction cycles. For a complete AES program, the $CALL$ *store_var* instruction is executed 20 times. Hence, the latency overhead for a complete AES is 34 * 20 + 20 * 2 ($CALL$ *store_var* instructions) + 5 * 11 (other added instructions inside the *key_addition* subroutine) = 775 instruction cycles.

This countermeasure could fail if the attacker is able to inject faults to both the Tgt_inst and either the $XORWF\ 0x60, F$ (at line 15) or the $GOTO\ MAIN$ (at line 22). However, this is not feasible since faults could not be injected within 13 instructions.

4.8.4 Countermeasure for the $DECFSZ$ Instruction

The $DECFSZ$ instruction was utilized to conduct a powerful attack on AES. The countermeasure for protecting AES over the fault injected to the $DECFSZ$ instruction is shown in listing 4.13. The second round counter register is added as shown at line 8. The constant stored at the second round counter is 0x0B instead of 0x0A. Both round counters are decremented for each round. Then, during the last round, before executing the $DECFSZ$ *round_counter*$, F$ instruction, both round counters store 0x01. If the target instruction is faulted, the *GOTO loop_encrypt* is executed while it should be skipped. The second round counter will decrement again and force the program to jump back to the $RETURN$ instruction preventing the program from running more rounds.

```
1  encrypt:
```

```
2   MOVLW 0x01
3   MOVWF rcon        ; RCON initialization
4   CALL  key_addition     ; initial key addition
5   MOVLW 0x0A     ; start the round counter for the loop
6   MOVWF round_counter    ;
7   MOVLW    0x0B
8   MOVWF    round_counter_sec    ; Store 0x0B to the second round counter
9 loop_encrypt:
10     DECF    round_counter_sec , F
11   BTFSC   STATUS, Z
12   GOTO    BACK_RET
13   CALL  substitution_S
14   CALL  enc_shift_row
15   DECF  round_counter ,W
16   BTFSC STATUS,Z
17   GOTO  last_round
18   CALL  mix_column
19 last_round:
20   CALL  enc_key_schedule
21   CALL  key_addition     ; key addition
22   MOVLW    0x01
23   SUBWF    round_counter , w
24   BTFSS    STATUS, Z
25   GOTO     no_trigger
26   BCF      STATUS, RP0
27   BCF      STATUS, RP1
28   MOVLW    b'00100000'
29   MOVWF    PORTC
30 no_trigger:
31     4*NOPs
32   DECFSZ   round_counter ,F        ; Target instruction
33   GOTO  loop_encrypt
34 BACK_RET:
35   RETURN
```

Listing 4.13: Countermeasure for protecting the DECFSZ instruction

The experiment was performed with D_{t2p} set to 254 ns for 100 rounds. No faulty ciphertext was output. Therefore, this countermeasure successfully thwarted the attack. The overhead for this countermeasure is an additional single byte data memory for storing the second round counter and another five more instructions. These five instructions take 5*14 bits FLASH memory. The latency overhead is 2 + 1*10 +2*10 ($BTFSC$ takes two instruction cycles if it skips the next instruction) = 32 instruction cycles.

This countermeasure might be bypassed if the attacker could skip the Tgt_inst and

then skip the *GOTO BACK_RET* instruction at line 12. This is not feasible within our assumption.

4.8.5 Countermeasure to Protect the AES

The single countermeasure proposed for the $XORWF$ or the $DECFSZ$ could not totally prevent the AES key from being retrieved by the attacker. For example, if the $DECFSZ$ instruction is protected by using the countermeasure proposed in Section 4.8.4, the key could be retrieved by using the attack algorithm proposed in Section 4.5.1 where the $XORWF$ instruction was faulted, and vice versa. Therefore, it is important to combine both countermeasures together so that the previous attacks described in Section 4.5.1 and 4.6.2 could be prevented. Hence, both countermeasures for the $XORWF$ and the $DECFSZ$ were applied to AES, exactly as shown in sections 4.8.3 and 4.8.4, respectively. Triggers for both types of faults were inserted. The code for this countermeasure is shown in appendix B.5.

The verification experiment for this countermeasure was performed for 100 rounds. No faulty ciphertext was generated. Hence, it provides a solid protection to the AES-128 program from the attacks proposed in Section 4.5.1 and 4.6.2. This countermeasure utilizes 17 bytes more data memory to store the intermediate state and the round counter. Moreover, it takes another 46 instructions, which utilize 46*14 bits FLASH memory. The latency overhead for a complete AES program is another 775+32 = 807 instruction cycles.

4.9 Summary and Comparison

In this chapter, FIA experiments on PIC16F687 were introduced with detailed analysis on the effect of the induced fault. The EM pulse empirically influenced the prefetch cycle of the Tgt_inst and under a successful fault injection, the instruction was replaced with a Flt_inst. Table 4.21 summarizes all the instructions that had been tested and the associated possible Tgt_insts from different test programs.

In previous research, the methodologies were mostly focused on the experimental setup where the researcher listed out the equipment used, the controllability of the parameters of the generated pulse, or the underlying silicon geometries [7, 4, 32].

With the Flt_insts revealed, two AES-128 attacks were physically implemented using a two-step methodology. After identifying the Flt_inst, the implemented attacks are

Table 4.21: The instructions that had been tested with all the observed Flt_insts in different test programs

Tgt_inst	Flt_inst
$MOVLW$ $0x80$	$MOVWF$ $INDF$ $GOTO$ $0x80$ BCF $INDF$, 0
$MOVWF$ $ADDR$	NOP
$XORWF$ $ADDR$, F	$XORWF$ $ADDR$, W $IORWF$ $ADDR$, W $IORWF$ $ADDR$, F NOP $MOVWF$ $ADDR$ $SUBWF$ $ADDR$, W
$IORWF$ $ADDR$, F	$IORWF$ $ADDR$, W
$SUBWF$ $ADDR$, W	$MOVWF$ $ADDR$ NOP
$DECFSZ$ $ADDR$, F	$DECF$ $ADDR$, F $INCF$ $ADDR$, F $COMF$ $ADDR$, F $MOVWF$ $ADDR$ NOP

straightforward and do not require complex analysis of the propagation of the fault in the cryptographic algorithm compared with the attacks discussed in [24, 82, 83, 5]. For example, Giraud et al. proposed a DFA which required a single bit fault on the last round of AES [24], which was not feasible in our case with either BPS201 or BPS202. With our experimental results, the engineering effort in characterizing the fault pays off from the perspective of finding a simpler DFA algorithm. Moreover, the attack requires a small number of EM pulses to retrieve the whole key, which shows that the cryptographic devices running without countermeasures are significantly vulnerable to the attack. For the AES attack targeting the $XORWF$ instruction, the plaintext must be changed after the iteration for all the key bytes which are not revealed yet. When attacking the AES targeting the $DECFSZ$ instruction, we do not need to have control over the plaintext. However, we need to record the plaintext so that we could verify if the retrieved key was correct.

Moreover, the attack methodology was demonstrated with chip #3, which indicates that the proposed attack is reliable and reproducible. Since it appeared that the chip was more sensitive to the fault over the flash memory area, the characterization phase could be done with a scan over a smaller area, which improves the efficiency.

Currently EM FIA could retrieve the K_{10} or K_{11} for AES-128 by faulting the $XORWF$

or the $DECFSZ$ instruction, respectively. With K_{10} or K_{11} revealed, the initial round key could be calculated. However, the attack is not sufficient to retrieve the initial key if AES-192 or AES-256 is utilized. Assume AES-256 is now running on the chip, the attacker needs to get both K_{13} and K_{14} to calculate the initial round key. If there is an additional fifteenth round, K_{14} and K_{15} are needed to calculate the initial key. However, the attacker could target the $XORWF$ instruction first and retrieve K_{14}. Then, K_{15} can be retrieved by targeting the $DECFSZ$ instruction. With two attacks combined, the attack can still break the security of AES-256 and reveal the initial key. A similar attack methodology also applies to AES-192.

Countermeasures were also designed and empirically verified based on the Flt_insts for the $MOVWF$, $MOVLW$, $XORWF$ and $DECFSZ$ instruction, which covers the byte-oriented instruction, the literal control instruction, the jump instruction, and the arithmetic instruction. The countermeasures consider all possible statistical faulty instructions. Hence, they provide robust protections over the target instructions. The countermeasures designed for the $XORWF$ and $DECFSZ$ instruction were also empirically verified with a real AES implementation as demonstrated in Section 4.8.3, 4.8.4 and 4.8.5.

4.9.1 Comparison with Previous Research

There were also few previous research papers which utilized the PIC16F as the target or applied EM FIA to physically retrieve the key of AES. Skorobogatov applied laser FIA to the PIC16F84 and PIC16F628 and modified the memory contents successfully [41]. This attack did not target any specific instruction but rather delivered the laser beam to the memory directly and modified the data contents.

Oswald applied the voltage glitching attack to the PIC16F687 microcontroller [95]. Unlike our research, the microcontroller was running with a nominal clock speed (8 MHz) and the supply voltage was set to 4 V. Their target instruction was the $GOTO$ instruction which was skipped under fault injection. As mentioned by Oswald, their experiments were used to demonstrate the fault injectability instead of performing a full attack towards a cryptographic algorithm. Hence, unlike our research where we analyzed the possible Flt_insts of each target instruction, they did not further analyze the associated faulty instruction.

The presented EM FIA attack targeting the $DECFSZ$ instruction described in Section 4.6.2 was also implemented using a laser FIA approach in [9]. The chip under attack was also PIC16F687. Unlike this research, the laser beam is able to skip multiple consecutive instructions with different laser pulse energy settings.

Dehbaoui et al. implemented another AES-128 attack in 2012 [74]. However, they did not perform the attack in assembly code level and did not find the corresponding faulty instruction. Their attack was based on a random byte fault model. Compared with our self-designed attack algorithms, their DFA algorithm was designed by Piret et al.[96]. The device under attack was an 8-bit AVR microcontroller. The microcontroller was clocked at 3.57 MHz. A trigger signal was activated at the beginning of the 9-th round. By sweeping through the 9-th round with 10 ns per step, they found the correct timing in inducing single byte fault to a specific byte. This single byte fault led to four faulty bytes in the ciphertext due to the mix-column operation and could leak information for four key bytes. The fault had to be injected at four different columns such that all the 16 key bytes could be retrieved. Additionally, for each injected byte fault, three sets of non-linear equation were set up and the attacker had to iterate through all the possible sub-key bytes. Compared with our attack which targeted the $XORWF$ instruction, we did not need to iterate through all the sub-key bytes and no non-linear equation was required to be solved.

Another attack which successfully broke the AES-128 was proposed in [49]. The attacker was able to inject faults to the incrementing round counter (the instruction was not specified) by using the EM pulse and introduced an additional round. The AES-128 was running on a microcontroller with an ARM-Cortex M3 core. The chip was clocked at 24 MHz, unlike our overclocking (however the rising time of their system was only 2 ns). The attacker was able to sweep through the last round of AES by 100 ps per step (unlike our 10 ns resolution), and they found a time window where the fault was injected to the round counter without triggering the HardFault exception. The desired result was that the round counter was not incremented to 10. However, unlike our research, the faulty value in the round counter was not analyzed, nor were any faulty instructions. Even if the round counter was set to 9, the attacker still needed to exhaustively search each key byte within their DFA. Then, each key byte would have two different candidate bytes. Hence, another exhaustive search needed to be done over 2^{16} different combinations. Consequently, the total search space was $2^8 \times 2^{16}$. This is unlike the proposed AES attack targeting the $DECFSZ$ instruction, where no additional exhaustive search was required.

Unlike the previous research using the Forward/backward body-bias injection, the probe must be in contact with the substrate [69, 70, 71]. Their successful attack to the CRT based RSA running on a secure microcontroller implemented with a 90 nm technology did not analyze the faulty instructions.

Moro et al. [91] designed a replacement sequence for different categories of instructions to mitigate the fault injection attack, however, unlike this research, only the instruction skip fault model was examined and the countermeasures were not verified empirically. Moro et al. [61] empirically verified some countermeasures designed for specific instructions,

however, countermeasures for a complete cryptographic algorithm were not provided.

The next chapter will provide research results on the LPC1114 microcontroller.

Chapter 5

Experimental results on LPC1114

This chapter begins with a brief introduction to the LPC1114 microcontroller, which contains an ARM Cortex-M0 core. We will present the EM FIA experimental results targeting the LPC1114.

Similarly to Chapter 4, results of a handheld experiment to check the fault injectability and clock frequency are reported. In addition, the automated platform results which find the best location for detailed analysis are presented.

Due to the damage of the chip, the fault injection experiments were done on two LPC1114 chips. Also, the experiments performed on LPC1114 were based on the BPS201 system since the BPS202 was damaged and we were not able to inject faults into LPC1114 with the 0.5 mm diameter probe tip. On chip #1, the address fault was the only fault that could be injected while both the address fault and the data fault could be injected on chip #2. Moreover, only chip #1 was placed on the automated platform and the BPS201 was utilized. All the experiments on chip #2 were performed with a handheld setup due to the limitations of the equipment as described in Section 3.3.

5.1 Introduction to the LPC1114

LPC1114 is a 32 bit microcontroller with an ARM Cortex-M0 core and manufactured by NXP with 32 kB on-chip flash programming memory [97]. The nominal maximum frequency is 50 MHz and the nominal supply voltage range is from 1.8 V to 3.6 V. The microcontroller is designed with a Von Neumann architecture where the program and the data share the same memory space unlike the Harvard architecture of the PIC16F687.

Thus, it is a good comparative platform for researching EM FIA. The Cortex-M0 implements the ARMv6-M architecture which supports the 16-bit Thumb instruction set and a small portion of 32-bit instructions from the Thumb-2 [98]. The instructions that had been tested during our experiments are shown in Table 5.1 along with the associated encoding. The LPC1114 was also decapsulated from the backside as shown in Figure 5.1. The copper shield was removed to ensure the EM pulse would not be blocked.

Table 5.1: Target instruction description and the associated encoding for LPC1114

Instruction	Description	Encoding (16 bits)
$LDR\ <Rt>,\ [PC,\ \#<imme>]$	Calculate an address from the PC and an immediate offset, then load a word from memory and write it to a register	01001 + Rt + imme
$LDR\ <Rt>,\ [<Rn>,\ \#<imme>]$	Calculate the address from a base register and an offset register value,	01101 + imme + Rn + Rt
$LDMIA\ <Rn>!,\ <registers>$	Load multiple registers from consecutive memory locations using an address from a base register	11001 + Rn + register_list
$CMP\ <Rn>,\ <Rm>$	Compare register and update the flag	0100001010 + Rm + Rn
$CMP\ <Rn>, \#<imm8>$	Compare immediate and update the flag	00101 + Rn + imm8

The LPC1114 utilizes a three-stage pipeline where two instructions are fetched in parallel in every other clock cycle as shown in Figure 5.2. Similar to the PIC16F687's pipeline flow, the processor may have to flush the pipeline if there is a branch instruction executing. The LPC1114 is different from the PIC16F687 in terms of the memory access instructions such as LDR, STR. The LPC1114 has an extra pipeline stall stage since these instructions take two cycles to be executed. For a single load instruction, the Advanced High-performance Bus (AHB) retrieves the data from the memory during the execution stage. The pipeline is stalled for one cycle for transferring the data to the destination register [99]. This is an important property for analyzing the timing when injecting the fault to the LPC1114.

Another important feature of the LPC1114 is the Nested Vectored Interrupt Controller (NVIC) which manages the system exceptions and peripheral interrupts [97]. When there is a system exception or an interrupt request, the NVIC checks the interrupt vector table

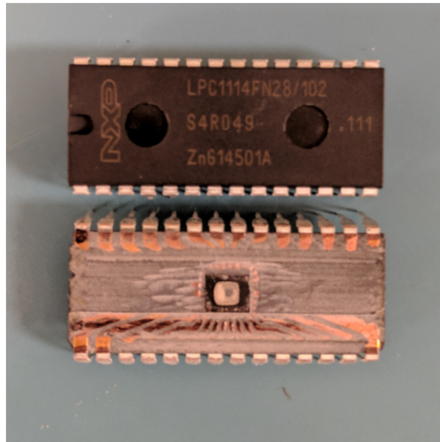


Figure 5.1: Original chip and backside decapsulated chip

based on the exception number or interrupt request number. The interrupt vector table stores the corresponding address for the exception handler or the interrupt service routine.

The HardFault exception plays a crucial role in our experiment. The exception number is 3 which indicates that it is the third highest priority exception. When a HardFault exception is triggered, the processor takes the exception and pushes eight registers R0, R1, R2, R3, R12, LR, PC, xPSR onto the stack. The register information will be used to analyze the reason of the HardFault [100]. By default, the HardFault handler is a infinite loop. It could be configured to output the register information from the stack.

There are two startup codes used to initialize the microcontroller in the Keil uVision5 [101]. One of them is written in C programming language while the other one is written in assembly. Both codes must be compatible with the selected device. The C code is typically used to configure the pins, clock frequency, peripherals, etc. The assembly code is mostly used to deal with the exception handlers and the interrupt service routine. Only the Reset handler or Non-maskable interrupt could preempt the fixed priority HardFault handler.

The trigger signal is generated on the PIO0_2 pin which is configured as general purpose output. A value was written to the GPIO0DATA register addresses from 0x50000000 to 0x50003FFC to assert or de-assert the trigger. The bits [13:2] of the address of the GPIO0DATA register are used to create a mask for writing or reading operations for the 12 GPIO pins. The write process to the i-th GPIO pin is only valid when the (i+2)-th bit of the address of the GPIO0DATA register is 1. The UART module was utilized to send out the faulty data for off chip analysis.

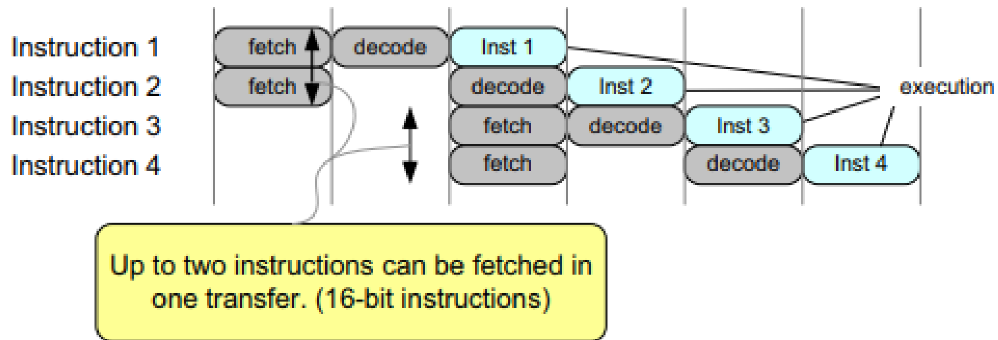


Figure 5.2: Three stage pipeline flow for ARM Cortex-M0 [12]

5.2 Experiment on Chip #1

In this section, the fault injection result on chip #1 will be introduced. The experiments done on chip #1 were divided into two parts. The first part was based on a handheld experiment where we aimed at injecting the first fault with a specific frequency. The second part was based on the CNC machine where an automated platform was built. A detailed timing analysis was done on the injected address fault. Unfortunately, before we could run more experiments, the chip was damaged possibly due to an unexpected short circuit.

5.2.1 Handheld Experiment on Chip #1

In this section, we will present the EM FIA result targeting the LPC1114 with a handheld experimental setup. Figure 5.3 shows the orientation of the probe tip. Note that this probe tip is the same as the one we used in Figure 4.6 even though they look different. The ferrite core is shorter because it was damaged. The copper wire was soldered back to the ferrite core after the damage.

To start with EM FIA, a test program was designed to investigate how a fault could be injected to the LPC1114. The test program is presented in listing 5.1. The program loads the same data value (or literal) to two different registers and then checks if the data values stored in the registers are equal. The two LDR instructions in the fifth and sixth line are $LDR\ PC - relative$ instructions. The assembler stores the constant value inside a specific address range in the code section referred as the literal pool. Then, it generates the offset between the PC and the address at which the constant is stored.

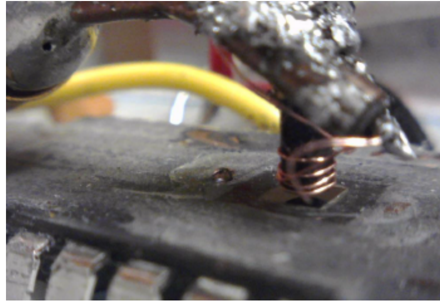


Figure 5.3: BPS201 probe tip over the die of the LPC1114

The program will branch into the $_FAULT$ subroutine if $R3 \neq R4$, which indicates that a fault occurs. The $_FAULT$ subroutine is an infinite loop and a GPIO output will be driven high to turn on an LED. A variable N is applied to make sure the EM pulse could successfully be delivered during the fetch/decode/execute of the target instruction even when the minimum D_{t2p} is used under different clock frequencies. This variable N actually works as an adjustable delay from the trigger to the target clock cycle.

```
1  _main
2    Assert trigger;
3      N*NOPs;
4    LDR R3, =(0xFEDCBA98);
5    LDR R4, =(0xFEDCBA98);
6    CMP R3, R4;
7    BNE _FAULT;
8    De-assert trigger;
9    B _main;
```

Listing 5.1: Test program one for handheld experiment targeting LPC1114

5.2.1.1 Experiment with $F_{clk} = 48\ Mhz$ and $V_{DD} = 3.3\ V$

Initially, the processor was running at a nominal 48 MHz F_{clk} and V_{DD} was set to 3.3 V. N was set to 13. As shown in Figure 5.4a (code 1-a), the delay from the trigger to the end of the execution stage of the $LDR\ R3$ instruction (in cycle 14) is 14*1000/48 = 291.7 ns while the D_{t2p} was set to 284 ns (EM pulse injected in cycle 14). However, even when we set the V_p to the maximum, no fault could be injected. Moreover, we adjusted the D_{t2p} by 10 ns per step to scan through all the clock cycles where LDR, CMP or BNE instructions were fetched, decoded or executed, but no fault was injected.

Cycles	0	1	2-10	11	12	13	14	15	16	17	18
Trigger PIO0_2											
Fetch	NOP2 NOP3		Fetch NOPs		LDR R3 LDR R4		CMP R3, R4 BNE _FAULT	STALL		STALL	Others
Decode	NOP1	NOP2	Decode NOPs	NOP12	NOP13	LDR R3	LDR R4	STALL	CMP R3, R4	STALL	BNE _FAULT
Execute	trigger	NOP1	Execute NOPs	NOP11	NOP12	NPO13	LDR R3	STALL	LDR R4	STALL	CMP R3, R4

(a) Test code 1-a, $N = 13$

Cycles	0	1	2-12	13	14	15	16	17	18	19	20
Trigger PIO0_2											
Fetch	NOP2 NOP3		Fetch NOPs		LDR R3 LDR R4		CMP R3, R4 BNE _FAULT	STALL		STALL	Others
Decode	NOP1	NOP2	Decode NOPs	NOP14	NOP15	LDR R3	LDR R4	STALL	CMP R3, R4	STALL	BNE _FAULT
Execute	trigger	NOP1	Execute NOPs	NOP13	NOP14	NOP15	LDR R3	STALL	LDR R4	STALL	CMP R3, R4

(b) Test code 1-b, $N = 15$

Cycles	0	1	2-22	23	24	25	26	27	28	29	30	31
Trigger PIO0_2												
Fetch	NOP2 NOP3		Fetch NOPs		NOP26 LDR R3		LDR R4 CMP R3, R4		STALL	BNE _FAULT LDR R0	STALL	
Decode	NOP1	NOP2	Decode NOPs	NOP24	NOP25	NOP26	LDR R3	LDR R4	STALL	CMP R3, R4	STALL	BNE _FAULT
Execute	trigger	NOP1	Execute NOPs	NOP23	NOP24	NOP25	NOP26	LDR R3	STALL	LDR R4	STALL	CMP R3, R4

(c) Test code 1-c, $N = 26$

Figure 5.4: Timing diagram for test program 1-a, 1-b and 1-c

Then, we wanted to check if the fault could be injected during the fetch stage of the two LDR instructions. N was set to 15 and the associated timing diagram is shown in Figure 5.4b (code 1-b). The delay from the trigger to the end of the fetch stage of two LDR instructions is 291.7 ns (end of cycle 14) and the D_{t2p} was set to 284 ns. However, no fault could be injected with the maximum V_p.

We also scanned through all the possible delay settings from 274 ns up to 504 ns with 10 ns per step using a trial and error method for both test programs. However, we still did not inject any faults under the 48 MHz clock frequency and 3.3 V supply voltage even though the V_p was set to 180 V which is the maximum pulse voltage for BPS201 system.

In summary, when the chip is running with a nominal clock frequency and nominal supply voltage, a fault could not be injected with the BPS201 system even though we set the V_p to its maximum value. Hence, the chip was overclocked (see Section 3.1) to investigate the fault injectability.

5.2.1.2 Experiment with $F_{clk} = 104.4Mhz$ and $V_{DD} = 3.3V$

Since we could not inject the fault with a nominal clock frequency and supply voltage, we overclocked the chip to inject the fault more easily. First, we found that the chip was still running correctly when F_{clk} was boosted to 107.2 MHz. For up to 30 minutes, the chip could correctly output the trigger signal at 107.2 MHz. If we kept increasing the frequency to 108 MHz, the chip stopped outputting the trigger signal and had to be reset to be functional. Then, we reduced the F_{clk} to 104.4 MHz. Further experiments showed that with overclocking technique (see Section 3.1), a fault could be injected and the program branched into the permanent $_FAULT$ loop. We fixed the F_{clk} to 104.4 MHz where the chip executed the program correctly without any faults. The number of NOPs was increased to 26 as shown in Figure 5.4c (code 1-c). The delay from the trigger to the end of the stall stage of the $LDR\ R4$ instruction was 287.4 ns and D_{t2p} was set to 284 ns. The program successfully jumped into the $_FAULT$ loop and the LED was turned on successfully which indicated that the fault was injected.

The UART was embedded inside the $_FAULT$ loop. Therefore, if a fault is injected, the faulty value will be sent out by the UART to the PC.

To analyze the faulty behavior, a simulation was performed first by using the debug function in the Keil uvision5 [101]. The disassembly window shows how each instruction is encoded and where it is stored in the memory. The result is shown in Table 5.2. The $LDR\ R4, = (0xFEDCBA98)$ instruction is a $LDR,\ PC-relative$ with an offset of 0x30. The DCW is a directive that allocates a half word of memory.

Table 5.2: Simulation result for test code 1-c

Address (in hexadecimal)	Encoding(in hexadecimal)	Instruction
0x00000A14	0x4C0C	$LDR\ R4, [PC,\ \#0x30]$
0x00000A16	0x42A3	$CMP\ R3,\ R4$
0x00000A18	0xD19A	$BNE\ 0x00000950$
0x00000A1A	0x480C	$LDR\ R0, [PC,\ \#0x30]$
......	......	Other instructions
0x00000A48	BA98	$DCW\ 0xBA98$
0x00000A4A	FEDC	$DCW\ 0xFEDC$

From Table 5.3, the faulty value for test code 1-c was 0x480CD19A and it is equivalent to the encoding of the $CMP\ R3,\ R4$ and $BNE\ 0x00000950$ (0x00000950 was the address of the $_FAULT$ loop) instruction. Therefore, it is likely that the offset was not correctly added to the PC or the two set bits in the offset were reset when the fault was injected. Hence, the correct data was not loaded from the literal pool. Consequently, the next two instructions were loaded into register R4. In fact, the $BNE\ _FAULT$ and $LDR\ R0, [PC,\ \#0x30]$ were fetched together from Figure 5.4c. Thus, it is assumed that the induced fault was caused due to fetching the data from a faulted address. Therefore, this fault was categorized as the address fault.

The handheld experiment empirically verified the proposed methodology for how to inject a fault by using overclocking technique. However, more experiments needed to be done to analyze the faulty behavior. Thus, the automated platform experiment was designed to run more in depth analysis to investigate the reason why the load instructions fetched the next two instructions from the memory.

5.2.2 Automated Platform Experiment on chip #1

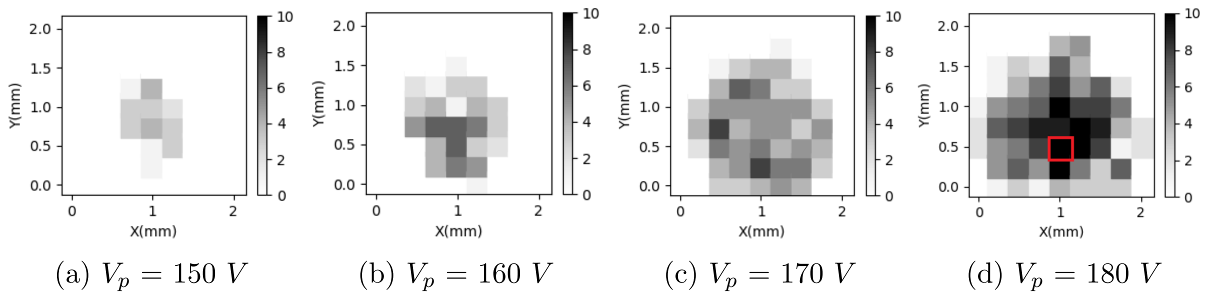
After the initial experiment with the handheld setup, an automated platform was built. Note that BPS201 was utilized due to the damage of the probe of the BPS202. The automated platform first aimed at finding out a best location where the fault could be injected more easily. Then, we developed other test programs to investigate how the faulty value was sent out from the UART at this best location.

5.2.2.1 Scan Over the Die of The LPC1114

To find out the best position for injecting the fault, we also utilized the CNC machine to work as the motorized x-y-z stage to build the platform and run the scan over the die of

the chip. Similarly, we analyzed the impact of the V_p by running the scan several times with different V_ps. Test program 1-c was utilized for this scan. The supply voltage was set to 3.3 V and the clock frequency was set to 104.4 MHz.

Similar to the experiments performed on the PIC16F687, we run a scan over the die of the chip to find out the best coordinate for injecting the fault. The die size of the LPC1114 is around 2 $\times$ 2 mm^2. The resolution of the scan was set to 0.254 mm and the probe tip was initially placed close to the left bottom corner of the die. The probe tip was moving over a square with 9 $\times$ 9 points in total, which covered an area of 4.13 mm^2.



(a) $V_p = 150\ V$ (b) $V_p = 160\ V$ (c) $V_p = 170\ V$ (d) $V_p = 180\ V$

Figure 5.5: Shmoo plot for the number of faults injected at each point over 10 rounds for the scan over the die of the LPC1114 under different V_ps

The shmoo plots are shown in Figure 5.5. When the V_p is reduced, the fault is injected at fewer points. Thus, we select the red rectangle point shown in Figure 5.5d as the best point. The associated x and y coordinate for this point was 1.016 mm and 0.508 mm, respectively.

After the best coordinate was found, several other different test programs were designed to investigate the faulty behavior. In our experiment, two different faults were found. The first fault was injected since the $LDR\ PC-relative$ instruction loaded data from a wrong address. The second kind of fault was a HardFault injected to the circuit. Both faults will be further analyzed in the following sections.

5.2.2.2 Analysis for The Fault Targeting the $LDR\ PC-relative$ Instruction

We needed to identify which instruction was actually being faulted and how the fault occurred during the experiment using test code 1-c. We designed three more test programs to better analyze the fault occurred when targeting the $LDR\ PC-relative$ instruction. The associated timing diagrams are shown in Figure 5.6a, 5.6b and 5.6c. For all these

cases, faults were induced successfully and the faulty values are listed in Table 5.3. The faulty value was consistent for each case when it was output by the UART. The faulty behaviors were further analyzed below. The frequency was still fixed at 104.4 MHz and the supply voltage was 3.3 V.

5.2.2.3 Analysis on the Effect of Code Sequence

As shown in Figure 5.4c, when the $LDR\ R3$ instruction is being decoded, the $CMP\ R3,\ R4$ instruction is being fetched. When the $LDR\ R4$ instruction is being executed, the processor is also decoding the $CMP\ R3,\ R4$. In order to make the analysis simpler, test program 1-d was generated to help eliminate the impact of other instructions which were decoding or executing in parallel with the target instruction. Sixteen NOPs were added after the two LDR instructions and six NOPs were added between the $CMP\ R3,\ R4$ and $BNE\ _FAULT$ instruction. Figure 5.6a shows the timing for test code 1-d (also shown in listing 6.2). From the second row of Table 5.3, the faulty value was 0xBF004C17. Additionally, the fault was injected with the same D_{t2p} of 284 ns, when the EM pulse was delivered over cycle 30 in Figure 5.4c. Thus, the $CMP\ R3,\ R4$ and $BNE\ _FAULT$ instructions were not the reason for the injected fault since they were far away from the EM pulse due to the many NOPs added. The simulation result is shown in Table 5.4. The faulty value indicated that the $LDR\ R4$ instruction loaded the opcode of itself and the next NOP into register R4.

Table 5.3: Summary of the fault injection result by using UART to read the faulty output with D_{t2p} set as 274, 284, 294, 304 ns

Test code	Faulty value in R4 (R5 in the last row) (received from UART)	Associated instructions
1-c	0x480CD19A	$LDR\ R0, [PC,\ \#0x30],\ BNE\ 0x00000950$
1-d	0xBF004C17	$NOP,\ LDR\ R4, [PC,\ \#0x5C]$
1-e	0x20004C17	$MOVS\ R0, \#0,\ LDR\ R4, [PC,\ \#0x5C]$
1-f	0xBF004D17	$NOP,\ LDR\ R5, [PC,\ \#0x5C]$

Test code 1-e (Figure 5.6b) was almost the same as test code 1-d except the first NOP after the $LDR\ R4$ was replaced with a $MOVS\ R0$ instruction. The faulty value was changed to 0x20004C17 which indicated that the $MOVS\ R0$ instruction was loaded into the register R4. We also analyzed if the fault could be injected when the destination register was changed from R4 to R5 as shown in test code 1-f in Figure 5.6c. The faulty value was changed to 0xBF004D17 which demonstrated that the fault could also be injected with a different destination register

Cycles	0	1	2-22	23	24	25	26	27	28	29	30	31
Trigger PIO0_2												
Fetch	NOP2 NOP3		Fetch NOPs		NOP26 LDR R3		LDR R4 NOP27		STALL	NOP28 NOP29	STALL	
Decode	NOP1	NOP2	Decode NOPs	NOP24	NOP25	NOP26	LDR R3	LDR R4	STALL	NOP27	STALL	NOP28
Execute	trigger	NOP1	Execute NOPs	NOP23	NOP24	NOP25	NOP26	LDR R3	STALL	LDR R4	STALL	NOP27

(a) Test code 1-d, $N = 26$, more NOPs added after the $LDR, R4$ instruction

Cycles	0	1	2-22	23	24	25	26	27	28	29	30	31
Trigger PIO0_2												
Fetch	NOP2 NOP3		Fetch NOPs		NOP26 LDR R3		LDR R4 MOVS R0, #0x0		STALL	NOP27 NOP28	STALL	
Decode	NOP1	NOP2	Decode NOPs	NOP24	NOP25	NOP26	LDR R3	LDR R4	STALL	MOVS R0, #0x0	STALL	NOP27
Execute	trigger	NOP1	Execute NOPs	NOP23	NOP24	NOP25	NOP26	LDR R3	STALL	LDR R4	STALL	MOVS R0, #0x0

(b) Test code 1-e, $N = 26$, change the first NOP after $LDR\ R4$ to $MOVS\ R0$

Cycles	0	1	2-22	23	24	25	26	27	28	29	30	31
Trigger PIO0_2												
Fetch	NOP2 NOP3		Fetch NOPs		NOP26 LDR R3		LDR R5 NOP27		STALL	NOP28 NOP29	STALL	
Decode	NOP1	NOP2	Decode NOPs	NOP24	NOP25	NOP26	LDR R3	LDR R5	STALL	NOP27	STALL	NOP28
Execute	trigger	NOP1	Execute NOPs	NOP23	NOP24	NOP25	NOP26	LDR R3	STALL	LDR R5	STALL	NOP27

(c) Test code 1-f, $N = 26$, change target register to $R5$

Figure 5.6: Timing diagrams of test program 5.1 where EM pulse is injected in cycle 29, 30, 31 and 32

Table 5.4: Simulation result for test code 1-d, 1-e and 1-f

Address (in hexadecimal)	Encoding(in hexadecimal)	Instruction
0x00000A14	0x4C17 (1-d,e) 0x4D17 (1-f)	$LDR\ R4, [PC,\ \#0x5C]$ (1-d,e) $LDR\ R5, [PC,\ \#0x5C]$ (1-f)
0x00000A16	0xBF00 (1-d,f) 0x2000 (1-e)	$NOP27$ (1-d,f) $MOVS\ R0$ (1-e)
0x00000A18	0xBF00	$NOP28$
0x00000A1A	0xBF00	$NOP29$
......	......	Other instructions
0x00000A74	BA98	$DCW\ 0xBA98$
0x00000A76	FEDC	$DCW\ 0xFEDC$

In summary, the $LDR\ PC - relative$ instruction could be faulted. The induced fault caused the processor to fetch data from a wrong address. Here are the key findings in this section:

1. The fault is injected to the $LDR\ PC - relative$ instruction.

2. The faulty $LDR\ PC - relative$ could load the value from the same faulty address independent of the instruction stored at that address and the target register of the $LDR\ PC - relative$ instruction.

5.2.2.4 Analyze the Timing and Address

After analyzing the faulty behavior on some different programs, there are two remaining questions to be resolved. First question is the timing issue. During which pipeline stage, is the $LDR\ PC - relative$ instruction faulted? Second question is called the address issue. Comparing test program 1-d with 1-c, why were the faulty values output from different addresses in the memory?

For test program 1-c, 1-d, 1-e, 1-f, the delay from asserting the trigger signal to the fetch, decode and execute cycle of the $LDR\ R4/R5$ instruction is the same. Moreover, fault could be injected with the same range of D_{t2p}s as shown in Table 5.3. Therefore, for a specific D_{t2p} and a specific pipeline stage of $LDR\ R4$ instruction, the delay between the pulse and this edge is the same. Hence, we will only analyze test program 1-d for the timing and also develop some additional test programs to determine the details of the address fault.

Table 5.5: Delay from trigger to the four cycles for $LDR\ R4/R5$ instruction

Pipeline stage	Start (ns)	End (ns)	Cycle (in Figure 5.6a)
Fetch	239.5	249.0	26
Decode	249.0	258.6	27
Execute	268.2	277.8	29
Pipeline stall	277.8	287.4	30

Table 5.5 presents the delay from the trigger to the four clock cycles where $LDR\ R4/R5$ was fetched, decoded, executed and the pipeline stall stage. Also, from Table 5.3, the fault was injected when D_{t2p} ranged from 274 ns to 304 ns. Other D_{t2p}s ranging from 314 ns to 344 ns, no function faults could be injected. For the discussion of these faults which

are injected in cycle 31 and 32 (equivalent D_{t2p} 294 ns and 304 ns), see detailed analysis in Section 5.3 with experimental results on chip #2.

To better understand the address issue, some more test programs were developed. The objective was to study the relationship between the address where the literal was stored and the address where the faulty value was stored. For example, from Table 5.3 and Table 5.4, the address for the literal was 0x00000A74 and the address where the faulty value was stored was 0x00000A14. The test programs insert different number of NOPs to increase the address where the literal was stored. Then, we checked where the output faulty value was stored. The address for the literal in different test programs ranged from 0x00000A48 to 0x00000A80. We could not add more NOPs afterwards. Otherwise, the branch instruction inside the code will have an out of range error. Moreover, we could not reduce the address of the literal to be below 0x00000A48 since we need to make sure the delay between the trigger to the target $LDR\ R4$ instruction was the same. The literal was placed in a word-aligned literal pool. Thus, the increment for the address was 4.

Equation 5.1 illustrates the relationship between the two addresses. The EM pulse may have revised the address on the address bus and finally causes the fault. The arrow shows the original address on the left and the faulted address on the right side. The range of values of M is integers from 4 to 8, which all get faulted to '1'.

$$0x00000AMN \Longrightarrow 0x00000A1N;\ 0x48\ \leqslant 0xMN\ \leqslant 0x80.\ N\ \in [0,\ 4\ ,8\ ,C] \tag{5.1}$$

In summary, with the timing analysis on the injected fault, the $LDR\ PC-relative$ is likely faulted during the execution stage where the target address is sampled by the AHB to fetch the data from the literal pool and the faulty address is summarized in equation 5.1.

Other than the $LDR\ PC-relative$ instruction, the $LDR\ \ <Rt>,\ [<Rn>,\ \#<imme>]$ and $LDMIA\ \ <Rn>!,\ <registers>$ instructions were also utilized as our fault injection targets. However, we were unable to inject faults to them initially. Before we could determine the reason why faults were not injected, chip #1 was damaged together with the first probe tip. However, further experiments on the second chip with the second probe tip indicated that faults could be successfully injected into the targeted $LDR\ \ <Rt>,\ [<Rn>,\ \#<imme>]$ and $LDMIA\ \ <Rn>!,\ <registers>$ instructions. The experimental results on chip #1 where faults failed to be injected are summarized in the Appendix C for reference since they were not fully investigated. With the second probe tip and chip #2, the address fault was successfully induced along with the data fault. In the next section, experimental results on chip #2 are presented.

5.3 Experiment on Chip #2

This section presents the results of experiments on chip #2, using a handheld setup, focused on various types of load instructions of the LPC1114. All experiments were performed with $F_{clk} = 100MHz$ and $V_{DD} = 3.3V$. The reason for using handheld setup was discussed in Section 3.3.

5.3.1 Experiments Targeting the $LDR\ PC - relative$ Instruction

The experiment on the second chip started with targeting test program 1-d, which focused on the $LDR\ PC - relative$ instruction. The timing diagram of test program 1-d is illustrated again in Figure 5.7a.

Cycles	0	1	2-22	23	24	25	26	27	28	29	30	31
Trigger PIO0_2												
Fetch	NOP2 NOP3		Fetch NOPs		NOP26 LDR R3		LDR R4 NOP27		STALL	NOP28 NOP29	STALL	
Decode	NOP1	NOP2	Decode NOPs	NOP24	NOP25	NOP26	LDR R3	LDR R4		NOP27		NOP28
Execute	trigger	NOP1	Execute NOPs	NOP23	NOP24	NOP25	NOP26	LDR R3		LDR R4		NOP27

(a) Test code 1-d, $N = 26$, more NOPs added after the $LDR, R4$ instruction

Cycles	0	1	2	3	4	5-25	26	27	28	29	30	31
Trigger PIO0_2												
Fetch	NOP1 NOP2		STALL	NOP3 NOP4		Fetch NOPs		LDR R4 NOP27		NOP28 NOP29	STALL	
Decode	LDR R3	NOP1		NOP2	NOP3	Decode NOPs	NOP25	NOP26	LDR R4	NOP27		NOP28
Execute	Assert trigger	LDR R3		NOP1	NOP2	Execute NOPs	NOP24	NOP25	NOP26	LDR R4		NOP27

(b) Test code 1-g,N= 26, $LDR\ R3$ was placed directly after asserting the trigger

Figure 5.7: Timing diagrams of test program 1-d and 1-g

The experiment was conducted with 100 rounds for each D_{t2p}. However, when D_{t2p} was set to 274 ns (cycle 28). We found that the fault could be injected to either R3 or R4 whereas for the previous experiment on chip #1 with the same D_{t2p} and V_p, the fault could only be injected to R4. The faulty values in R3 with D_{t2p} set to 274 ns (cycle 28) were shown in Table 5.6 where the column of count indicates the number of occurrence for this faulty value. The only faulty value for R4 was BF004C17. When D_{t2p} was set to 274 ns, it was likely that the EM pulse could affect the execution stage (cycle 27) of the $LDR\ R3$ instruction due to the 14 ns jitter (see Figure 3.5d). With a increasing D_{t2p}, the fault could only be induced to R4 and the associated faulty values were similar to test

program 1-g (to be further detailed below). The analysis of the fault induced to register R4 will be provided after presenting the results on test program 1-g.

Table 5.6: Faulty values in R3 with EM pulse injected in cycle 28

Faulty value	Count	Faulty value	Count	Faulty value	Count
BF4C4C17	1	FEDC3C17	1	FEDCB817	4
BF404C17	1	FEDCB818	3	FEDCB813	3
FEDCBC17	2	FE5C3C17	3	FEDCB811	2
BF004C17	3	FEDCB898	2	N/A	N/A

In order to make sure the fault was only injected to R4 and make the analysis simpler, test program 1-g was designed. The difference between test program 1-d and 1-g is that the $LDR\ R3$ instruction was placed directly after asserting the trigger as shown in Figure 5.7b where the $LDR\ R3$ instruction was executed in cycle 1. In this way, the $LDR\ R3$ instruction will not be affected by the EM pulse. As expected, all the faulty values we received indicated that no fault occurred in register R3. The faulty value was shown along with the associated count in Table 5.8. Also, test program 1-h was designed with some redundant NOPs removed so that the offset of the $LDR\ R6, [PC,\ \#0x4C]$ instruction was reduced. Based on equation 5.1, the address of 0xFEDCBA98 was reduced to 0xA64 to ensure that the address faulty value would still come out from address 0x00000A14. Moreover, the destination register was changed from R4 to R6, which changed the opcode of the target $LDR\ PC - relative$ instruction from 0x4C17 to 0x4E13. The simulation results for test program 1-g and 1-h are shown in Table 5.7. The fault injection results for test program 1-g and 1-h are presented in Table 5.8 and 5.9, respectively.

Table 5.7: Simulation result for test code 1-g and 1-h

Address (in hexadecimal)	Encoding(in hexadecimal)	Instruction
0x00000A14	0x4C17 (1-g) 0x4E13 (1-h)	$LDR\ R4, [PC,\ \#0x5C]$ (1-g) $LDR\ R6, [PC,\ \#0x4C]$ (1-h)
0x00000A16	0xBF00	NOP Other instructions
0x00000A64 (1-h) 0x00000A74 (1-g)	BA98	$DCW\ 0xBA98$
0x00000A66 (1-h) 0x00000A76 (1-g)	FEDC	$DCW\ 0xFEDC$

There are mainly four findings for test code 1-g and 1-h:

Table 5.8: Summary of the faulty value for test program 1-g with different delay settings (correct value should be FEDCBA98)

D_{t2p} (ns) /(Cycle #)	Faulty value in R4 (Hex) and their count					
	Address fault		Data fault		Combined fault	
	Fault value	Count	Fault value	Count	Fault value	Count
274 (28)	No	N/A	No	N/A	No	N/A
284 (29)	BF004C17	8	No	N/A	No	N/A
294 (30)	BF004C17	23	No	N/A	No	N/A
304 (31)	BF004C17	22	FEDCB898	4	No	N/A
314 (32)	BF004C17	12	FEDCB818	2	BF484C17	2
			FEDCB813	2	FEDCB817	1
324 (33)	BF004C17	1	FEDCB818	4	FEDCBC17	2
			FEDCB898	3	FEDC3C17	2
			FEDCB813	3	FEDCB817	2
			FEDCB810	3	FE5C7C17	1
			FEDCB811	2		
334 (34)	No	N/A	FEDCB898	3	FEDC3C17	2
			FEDCB811	2	FEDCBC17	1
			FEDCB818	2		
344 (35)	No	N/A	FEDCB898	13	No	N/A
			FEDCB818	1		
354 (36)	No	N/A	No	N/A	No	N/A

Table 5.9: Summary of the faulty value for test program 1-h with different delay settings (correct value should be FEDCBA98)

D_{t2p} (ns) /(Cycle #)	Faulty value in R6 (Hex) and their count					
	Address fault		Data fault		Combined fault	
	Fault value	Count	Fault value	Count	Fault value	Count
274 (28)	No	N/A	No	N/A	No	N/A
284 (29)	BF004E13	26	No	N/A	No	N/A
294 (30)	BF004E13	18	No	N/A	No	N/A
304 (31)	BF004E13	25	FEDCBA18	2	No	N/A
314 (32)	BF004E13	13	FEDCBA13	5	FEDC3E13	1
			FEDCBA18	2		
324 (33)	BF004E13	4	FEDCBA13	1	FEDC3E13	1
					BF404E13	1
334 (34)	No	N/A	FEDCBA11	5	FEDCBE13	1
			FEDCBA13	4		
			FEDCBA10	1		
344 (35)	No	N/A	FEDCBA18	7	No	N/A
			FEDCBA13	6		
			FEDCBA11	5		
			FEDCBA10	2		
354 (36)	No	N/A	No	N/A	No	N/A

1. Instead of only finding the address fault, another different kind of fault was found. For example, when D_{t2p} was set to 344 ns, we did not find BF004C17 as the faulty value. However, another faulty value FEDCB898 was found. The hamming distance between this faulty value and the correct data which was supposed to be loaded into register R4 is 1. Hence, this kind of fault is defined as the data fault. This kind of faulty value was also found with the experimental results on test program 1-h as shown in Table 5.9.

2. There were some other faulty values which were likely caused by affecting both the execution and the pipeline stall stage. For example, when D_{t2p} was set to 334 ns, one possibly faulty value was BF4C4C17. The hamming distance between this faulty value and the faulty value from the address fault (which was BF004C17) was 2. Therefore, the EM pulse probably induced the fault in both the execution and the pipeline stall stage. Consequently, the AHB returned BF004C17 and it was faulted again during writing to register R4. Thus, this kind of fault was defined as the combined fault. Moreover, the combined fault occurred only when both data fault and address fault were assumed to be injected. Similar combined faulty value BF404E13 was also observed in Table 5.9.

3. Another different kind of faulty value was in form as FEDC + 4C17 where the first four hex digits were likely from the correct address while the remaining four hex digits were from the wrong address. The ARM v6-M architecture supports halfword aligned memory access [98]. Hence, the EM pulse might be affecting the selection signal of the multiplexer of the AHB [102], resulting in a halfword address fault. The faulty value FEDC + 4C17 was likely faulted again during the pipeline stall stage, which caused a final combined fault. For example, faulty value FEDCB817, FEDCBC17, FE5C7C17 were all classified as the combined fault. Also, if we consider these faulty values as data fault, FEDCBA98 should be the data on the bus after the execution stage (cycle 29) and many bits needs to be flipped to get these faulty values as shown in Table 5.10. Since the rising time of the pulse generated by BPS201 is 13 ns, the EM pulse might be able to affect both the execution stage and the pipeline stall stage, which finally caused the combined fault. Similar faulty values such as 0xFEDC3E13 was also found for test code 1-h in Table 5.9.

Table 5.10: Hamming distance between the final faulty value and the value on the data bus after the execution stage

Final faulty value	data on the bus after the execution stage	
	FEDC + 4C17	FEDC + BA98
FEDCBC17	4	7
FEDCB817	5	6
FE5C7C17	3	10
FEDC3C17	3	8

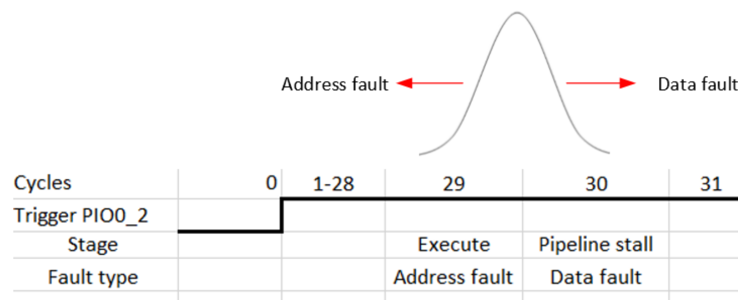


Figure 5.8: Fault distribution illustration for test program 1-g

4. When D_{t2p} was set to 284 ns, only the address fault was injected to register R4. With increasing D_{t2p}, it was more likely to inject the data fault before D_{t2p} reached

354 ns. Similarly, the address fault was more easily to be injected with reducing D_{t2p} as shown in Figure 5.8 assuming the jitter for the EM pulse follows a normal distribution. It may be that the jitter was larger than 14 ns, however, this would not explain the faults being injected over a much wider time region, that of 7 cycles (284 ns - 344 ns). Since the ARM Cortex-M0 focuses on low power, it may be that the succeeding NOP instructions did not change the control signals to reduce the switching within the architecture. In effect, the fault could still be injected several cycles after the $LDR\ R4/R6$ had completed since the control signals were each cycle relatching the same data into R4/R6. It is interesting that consistent to all tables 5.8 and 5.9, the faults could no longer be injected after 5 NOPs which followed the $LDR\ R4/R6$ instruction. This may be due to the instruction buffer containing 3 words of instructions. Also, from Figure 5.7b, the $LDR\ R4$ was fetched together with the 27-th NOP instruction. Hence, after completing 6 instructions ($LDR\ R4$ + 5 NOPs), it contains the next 6 instructions and the control signals were removed.

5.3.2 Targeting the $LDR\ < Rt >, [< Rn >, \# < imme >]$ Instruction

The $LDR\ < Rt >, [< Rn >, \# < imme >]$ instruction has similar functionality compared with the $LDR\ PC - relative$ instruction. Instead of calculating the address using $PC + offset$, the base address is now stored in a different register Rn. The final address is calculated from the base register plus the immediate offset. Therefore, test program two was designed to check the fault injection result on the $LDR\ < Rt >, [< Rn >, \# < imme >]$ instruction. Test program 2-a is shown in listing 5.2.

```
1      Initialization;
2  _main
3    Assert trigger;
4    LDR R2, [R0, #0x4];
5      N*NOPs;
6      LDR R3, [R1, #0x8]; Target instruction
7      NOPs;
8    CMP R2, R3;
9    NOPs;
10   BNE _FAULT;
11   De-assert trigger;
12   LDR R3, =(0x01234567)
13   B _main;
```

Listing 5.2: Test program 2-a targeting the LDR immediate offset instruction

The memory map is shown in Figure 5.9a. The memory from 0x10000000 to 0x10001000 (SRAM region) is filled with data starting from 0x00000000 and incremented by one every word. Thus, if a faulty value is the same as one of the values stored in these addresses, it would be easy to find the associated faulty address. Then, 0xFEDCBA98 is written to address 0x10000000 and 0x5A5A5A5A is written to both address 0x10000004 and 0x10000010. The base address stored in register R0 is 0x10000000 and in register R1 is 0x10000008. Thus, after executing the $LDR\ R2,\ [R0,\ \#0x4]$ and $LDR\ R3,\ [R1,\ \#0x8]$ instruction, the value stored at register R2 and R3 should be 0x5A5A5A5A.

```
0x10000000: 98 BA DC FE
0x10000004: 5A 5A 5A 5A
0x10000008: 02 00 00 00
0x1000000C: 03 00 00 00
0x10000010: 5A 5A 5A 5A
0x10000014: 05 00 00 00
0x10000018: 06 00 00 00
0x1000001C: 07 00 00 00
```

(a) Test program 2-a

```
0x10000000: 98 BA DC FE
0x10000004: 5A 5A 5A 5A
0x10000008: 69 45 23 01
0x1000000C: 6A 45 23 01
0x10000010: 5A 5A 5A 5A
0x10000014: 6C 45 23 01
0x10000018: 6D 45 23 01
0x1000001C: 6E 45 23 01
```

(b) Test program 2-b

Figure 5.9: Data value stored in the memory with the $LDR\ <Rt>,\ [<Rn>,\ \#<imme>]$ instruction

The timing diagram for both test program 2-a and 2-b is shown in Figure 5.10. Compared with test program 1-g (Figure 5.7b), the destination register of the target instruction is changed from R4 to R3 and the $LDR\ PC-relative$ instruction is changed to $LDR\ <Rt>,\ [<Rn>,\ \#<imme>]$ instruction.

Cycles	0	1	2	3	4	5-25	26	27	28	29	30	31
Trigger PIO0_2												
Fetch	NOP1 NOP2		STALL	NOP3 NOP4		Fetch NOPs		LDR R3 NOP27		NOP28 NOP29	STALL	
Decode	LDR R2	NOP1		NOP2	NOP3	Decode NOPs	NOP25	NOP26	LDR R3	NOP27		NOP28
Execute	Assert trigger	LDR R2		NOP1	NOP2	Execute NOPs	NOP24	NOP25	NOP26	LDR R3		NOP27

Figure 5.10: Timing diagram for test program 2-a where N=26

The experimental results are shown in Table 5.11. No pure data fault was discovered. Faulty values 0x00000000 and 0xFEDCBA98 were defined as the address fault for the following reasons:

1. The value 0xFEDCBA98 was only stored at address 0x10000000 and 0x00000A78 (literal pool). Hence, it was likely that the destination address of the instruction

Table 5.11: Summary of the faulty value for test program 2-a with different delay settings (correct value should be 5A5A5A5A for both registers)

D_{t2p} (ns) /(Cycle #)	Faulty value (Hex) in Register Rx and their count					
	R3				R2	
	Address fault		Combined fault		Address fault	
	Fault value	Count	Fault value	Count	Fault value	Count
274 (28)	No	N/A	No	N/A	No	N/A
284 (29)	00000000	10	No	N/A	No	N/A
	FEDCBA98	3				
294 (30)	00000000	21	No	N/A	No	N/A
	FEDCBA98	3				
304 (31)	00000000	15	FEDCBA18	2	No	N/A
	FEDCBA98	11				
314 (32)	00000000	8	No	N/A	00000000	1
	FEDCBA98	2				
324 (33)	No	N/A	No	N/A	00000000	2
334 (34)	No	N/A	No	N/A	No	N/A

$LDR\ R3,\ [R1,\ \#0x8]$ was calculated as $[R1] - \#0x8$ while it was supposed to be $[R1]+\#0x8$. Consequently, the AHB returned the value stored at address 0x10000000 instead of the value stored at address 0x10000010.

2. The fault value 0x0000000 should not be the pure data fault because the hamming distance between 0x5A5A5A5A and 0x0000000 is 16. We might also consider it as the combined fault. For example, if the AHB returned the value from a wrong address, say 0x10000008, and it returned 0x00000002 from the memory but was faulted again during the pipeline stall stage, the faulty value would possibly be 0x00000000. To further verify this, test program 2-b was designed where we stored different initial values to SRAM to make the hamming distance between the initial value and 0x00000000 much larger, the memory map is shown in Figure 5.9b. However, 0x00000000 was still the faulty value we found. Thus, 0x0000000 is an address fault. The combined fault was also observed with faulty value as 0xFEDCBA18 when D_{t2p} was set to 304 ns.

Additionally, the fault also occurred in register R2. However, the EM pulse was unable to affect the $LDR\ R2,\ [R0,\ \#0x4]$ instruction. If the EM pulse changed the destination register from R3 to R2 in the pipeline stall stage and induced the address fault in the execution stage, we may expect this faulty value in register R2 but in the meantime, R3 should also be faulted with storing the initial data. However, no fault occurred in both

R2 and R3 at the same time. The reason for the fault injected to R2 might be the control signals on the datapath were faulted.

5.3.3 Targeting the $LDMIA < Rn >!, < registers >$ Instruction

$LDMIA < Rn >!, < registers >$ is another different kind of LDR instruction that is of interest. This instruction loads multiple words into a list of registers from consecutive memory locations with the base address stored in register Rn. Test program three shown in listing 5.3 was designed to investigate the possible faulty response of the $LDMIA < Rn >!, < registers >$ instruction. The program initially stored 0xFEDCBA98 to memory locations whose addresses are 0x10001000, 0x10001004 and 0x10001008. After that, the program applied a $LDMIA$ instruction to load the data stored in these three locations to register R1, R3 and R4, respectively at line 11 as shown in listing 5.3. Since no LDR instruction was added before the target instruction, 28 NOPs were utilized whereas in test program 1 and 2, only 26 NOPs were added before the target instruction (LDR instructions takes two clock cycles to execute). Thus, the delay between trigger to the execution stage of the target instruction was the same (cycle 29 is the execution stage and cycle 30 is the first stall stage as shown in Figure 5.11).

```
1  _main
2    LDR R0, =(0x10001000);
3    LDR R1, =(0xFEDCBA98);
4    LDR R3, =(0xFEDCBA98);
5    LDR R4, =(0xFEDCBA98);
6    STMIA R0!, {R1,R3, R4}; Store 0xFEDCBA98 to address from 0x10001000 to 0
       x10001008
7    LDR R0, =(0x10001000);
8     Assert trigger;
9       28*NOPs;
10
11   LDMIA R0!, {R1,R3, R4}; Target instruction
12      NOPs;
13   CMP R1,R3;
14   BNE FAULT;
15   CMP R3,R4;
16   BNE FAULT;
17   De-assert trigger;
18    Clear the value in R1, R3, R4;
19   B _main;
```

Listing 5.3: Test program three targeting the $LDMIA$ instruction

Cycles		1	2-25	26	27	28	29	30	31	32	33
Trigger PIO0_2											
Fetch	NOP2 NOP3		Fetch NOPs	NOP28 LDMIA		NOP29 NOP30		STALL 1	STALL 2	STALL 3	NOP31 NOP32
Decode	NOP1	NOP2	Decode NOPs	NOP27	NOP28	LDMIA	NOP29				NOP30
Execute	Assert trigger	NOP1	Execute NOPs	NOP26	NOP27	NOP28	LDMIA				NOP29

Figure 5.11: Timing diagram for test program three

The timing diagram of test program three is shown in Figure 5.11. Since the $LDMIA$ instruction in test program three had three destination registers (R1, R3, R4), it took 4 cycles to finish execution where the last three cycles were all in pipeline stall stage [103]. Therefore, the address for reading data from the memory was likely sampled at cycle 29, 30, 31 and the data was written back to R1, R3, R4 at cycle 30, 31, 32, respectively.

Table 5.12: Faulty value in R1 with test program three

D_{t2p} (ns) /(Cycle #)	Faulty value in R1 (Hex) and their count					
	Address fault		Data fault		Combined fault	
	Fault value	Count	Fault value	Count	Fault value	Count
274 (28)	00000000	8	No	N/A	000000C5	8
284 (29)	00000000	14	FEDCB298	1	000000C5	14
294 (30)	No	N/A	FEDCB298	5	No	N/A
			FEDCB088	1		
304 (31)	No	N/A	FEDCB098	6	No	N/A
			FEDCB088	1		
314 (32)to 354 (36)	No	N/A	No	N/A	No	N/A

Table 5.13: Faulty value in R3 with test program three

D_{t2p} (ns) /(Cycle #)	Faulty value in R3 (Hex) and their count					
	Address fault		Data fault		Combined fault	
	Fault value	Count	Fault value	Count	Fault value	Count
274 (28)	No	N/A	No	N/A	No	N/A
284 (29)	No	N/A	No	N/A	000000C5	7
294 (30)	00000000	7	No	N/A	000000C5	1
304 (31)	00000000	42	No	N/A	No	N/A
314 (32)to 354 (36)	No	N/A	No	N/A	No	N/A

The fault injection results are shown in Table 5.12 and 5.13. Based on these results, here are two findings:

1. When D_{t2p} was set to 274 ns, both the address fault and the combined fault could be injected to R1 while no fault occurred in R3. The faulty value 0x000000C5 was

considered as the combined fault because this word was not stored in any memory locations. Moreover, it should not be a pure data fault since the hamming distance between 0xFEDCBA98 and 0x000000C5 is 22. Hence, it was possibly that an address fault was injected first and the associated faulty value was 0x00000000. Then, during writing back to the destination register, four bits were flipped which resulted in a 0x000000C5.

2. It was expected that the fault could be injected to R1 first. With increasing D_{t2p}, fault should be injected to R3 and R4. More faults were injected to register R3 as shown in Figure 5.12. However, the fault was never injected to R4 even the D_{t2p} was increased to 354 ns.

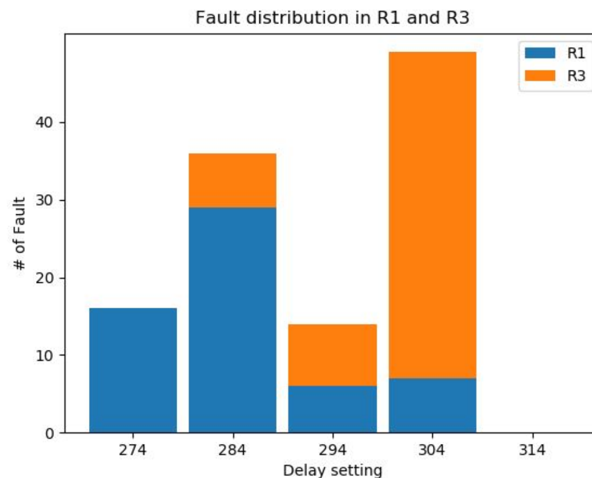


Figure 5.12: Fault distribution in R1 and R3

5.3.3.1 Targeting the CMP Instructions

In modern microprocessors, the secure boot process is widely applied to generate a chain of trust [25]. Each level of the bootloader will decrypt the next level image and verify its signature such that no malicious software could be run by the processor. Under this circumstance, the CMP instruction is of special interest since it is the most likely used instruction to verify the signatures.

```
1 _main
2   LDR R3, =(0xFEDCBA98);
```

```
 3  LDR R4, =(0xFEDCBA98);
 4   Assert trigger;
 5    N*NOPs;
 6
 7  CMP R3, R4;
 8  BNE _FAULT;
 9  De-assert trigger;
10  B _main;
```

Listing 5.4: Test program four targeting the CMP instruction

Test program four was designed to check if the fault could be injected to the CMP instruction. As shown in listing 5.4, we first load the same value to R3 and R4. Then, we use the CMP instruction to compare the value in register R3 and R4. If the CMP instruction is corrupted, the processor will jump to the $_FAULT$ loop where the UART will send out the faulty data. However, by changing N from 26 to 28 and trying all the possible D_{t2p}s from 274 ns to 404 ns with 10 ns per step, no faulty data was sent out by the UART.

Another different type of CMP instruction is the compare immediate instruction shown in the last row of Table 5.1. Instead of comparing the values stored in two different register, the $CMP\ \ < Rn > \# < imm8 >$ compares the value in register Rn to a immediate. The program is the same as listing 5.4 except that we replaced the $CMP\ R3,\ R4$ with the $CMP\ R3,\ 0x98$ and we load 0x00000098 to R3 register initially. The reason is the permitted value for the immediate is from 0 to 255. We also changed N from 26 to 28 and scanned through all possible delay settings from 274 ns to 404 ns. However, the experimental result shows that no faulty data was sent out by UART.

In fact, the processor did not branch to the $_FAULT$ loop every time for all the test programs when an abnormal behavior happened. In some cases, the processor just stopped outputting the trigger signal and also did not sent out any output information from the UART. The processor was likely halted in these cases. By checking the processor manual, the halted phenomenon probably occurred when the processor jumped to the HardFault handler. In the next section, the HardFault will be investigated.

5.4 HardFault

Fault exceptions are triggered when the processor detects a fault. In ARM Cortex-M0, the HardFault exception is the only exception type that handles a fault. The HardFault occurs in various situations. For example, when the processor tries to execute the $BKPT$

instruction with the debug function disabled, the HardFault exception will be triggered [104]. Also, the HardFault exception is triggered when an undefined instruction is executed.

During our fault injection experiment, the fault injected to the $LDR\ PC - relative$ instruction did not always lead to a faulty value sent out by the UART. Under these cases, the induced fault halted the processor.

In fact, the processor was likely executing the HardFault handler which is a infinite loop by default as described in Section 5.1. In order to identify what happened when the EM pulse halted the processor, we revised the HardFault handler based on the design described in [104]. The new HardFault handler will output the eight registers which are pushed onto the stack through UART when the processor takes an exception.

To test if the function of the HardFault handler was correct, we artificially inserted a HardFault by adding a $BKPT$ instruction. If the microcontroller is not running in the debug mode, the $BKPT$ instruction will trigger a HardFault exception. A simulation was performed first and the associated eight registers were recorded. Then, we programmed the chip and compared the output register information from the UART with the simulation result. It showed that all the register information was the same as the simulation, which indicated that the HardFault handler had the correct functionality. The code is shown in listing 5.5.

```
1  _main
2    Assert trigger;
3      26*NOPs;
4
5    LDR R4, =(0xFEDCBA98);
6    3*NOPs
7    LDR R3, =(0xFEDCBA98);
8    3*NOPs
9    CMP R3, R4;
10   3*NOPs ; The address of the third NOP is 0x000009FC
11   BKPT #0x08; Used to artificially trigger a HardFault. Removed later for
       FIA experiment
12   BNE _FAULT;
13   De-assert trigger;
14   B _main;
```

Listing 5.5: Test program five for HardFault handler

Cycles	0	1	2-22	23	24	25	26	27	28	29	30	31	32	33	34
Trigger PIO0_2															
Fetch	NOP2 NOP3		Fetch NOPs		NOP26 LDR R4		NOP27 NOP28			NOP29 LDR R3		NOP30 NOP31			NOP32 CMP R3, R4
Decode	NOP1	NOP2	Decode NOPs	NOP24	NOP25	NOP26	LDR R4	NOP27	STALL	NOP28	NOP29	LDR R3	NOP30	STALL	NOP31
Execute	trigger	NOP1	Execute NOPs	NOP23	NOP24	NOP25	NOP26	LDR R4		NOP27	NOP28	NOP29	LDR R3		NOP30

Figure 5.13: Timing diagram for code in listing 5.5

5.4.1 Complexity of the HardFault

After verifying that the HardFault handler was working properly, the $BKPT$ #0x08 instruction in line 11 was removed and the EM pulse was utilized to inject the HardFault. The timing analysis for this code is shown in Figure 5.13 with the $BKPT$ instruction removed. However, the faulty value sent out by the HardFault handler when the fault was injected by the EM pulse was complex. Moreover, the output register information was also strange. For example, Table 5.14 shows two groups of the output register information using the same code as shown in listing 5.5. These faulty values were collected on chip #2 with $F_{clk} = 100\ MHz$ and D_{t2p} was set to 294 ns.

In the first group, the PC was 0x0000094E, which pointed to the $BX\ LR$ instruction in the delay subroutine (This delay subroutine is used to ensure two consecutive EM pulses could be generated correctly). In the delay subroutine, the R0 is initially set to 0x00000600. Then, R0 is decremented until it reaches 0 when the $BX\ LR$ instruction is executed to return to the main loop. Therefore, the HardFault possibly occurred at executing the delay subroutine. In our code, both after we assert or de-assert the trigger signal, this delay subroutine is called. However, there are 300 clock cycles between trigger and decrementing R0 to 0x57C. Therefore, the EM pulse was unable to affect the delay subroutine. In the second group, the Link register stored 0x0000044B which should be the return address of a subroutine. However, the instruction at this address is used to setup the UART and is actually inside the startup code. There are no subroutine calls inside the startup code.

In summary, the HardFault handler outputs unexplainable information most of the time. Only when the experiment was done on chip #1 with $F_{clk} = 104.4\ MHz$ and D_{t2p} was set to 274 ns, did the processor output stable information from the handler which could be used for analysis. Therefore, the analysis on the HardFault was only carried out with this experimental setup.

Table 5.14: Two groups of register information output from UART on injecting HardFault to chip #2

Register	First group	Second group
R0	0x0000057C	0x10000000
R1	0x00000004	0x00000004
R2	0x00000008	0x00000004
R3	0x00000000	0x000000C8
R12	0x00000000	0x00000000
Link register	0x00000A11	0x0000044B
PC	0x0000094E	0x000009FC
Program Status Register (PSR)	0x21000000	0x01000000

5.4.2 HardFault on Chip #1 with $F_{clk} = 104.4\ MHz$, $V_{DD} = 3.3\ V$, $D_{t2p} = 274\ ns$.

The experiment on chip #1 was done with $F_{clk} = 104.4\ MHz$ and $V_{DD} = 3.3\ V$. The D_{t2p} was set to 274 ns. The output register information is shown in Table 5.15. These faulty values were consistent. The most significant bit of the PSR, which is the negative flag, was set to 1. It indicated that the result of an operation was negative. However, there were no negative operations in our code. Another finding was that the microcontroller took the HardFault exception when PC was 0x000009FC. The instruction at this address was the 32-nd NOP which is shown as $NOP32$ in timing diagram 5.13. Thus, the processor was likely fetching the $NOP32$, decoding $NOP31$ and executing $NOP30$ at this time when HardFault occurred assuming no fault was injected to the PC.

Table 5.15: Register information from simulation and received from UART after injecting the fault with EM pulse

Register	From UART	From simulation
R0	0xFC000000	0x50003FFC
R1	0x00000FC5	0x00000004
R2	0x00000004	0x00000004
R3	0x00000000	0xFEDABC98
R12	0x00000000	0x00000000
Link register	0x00000A21	0x00000A21
PC	0x000009FC	0x000009FC
PSR	0xA1000000	0x21000000

To understand the register information, we ran another simulation and set the program

counter to 0x000009FC. The simulation result shows the correct register values before the fault occurs when the PC is 0x000009FC. From Table 5.15, we could find that the data in R0, R1, R3 and the PSR had been changed when the fault occurred. The faulty value in R0 and R1 looked highly related to the value previous loaded into R0.

To investigate the relationship between the faulty value and the correct value, we changed the value stored at R0 before the fault occurred. The value in R0 was the address for the GPIO0DATA register. Therefore, we changed the address of the GPIO0DATA register but made sure bit 4 of the address was 1 so that the trigger signal could be asserted and de-asserted correctly. Table 5.16 presents parts of our testing result. Note that, only these three register values were changing when the address of the GPIO0DATA register was changed.

Table 5.16: Register information output by the HardFault handler after injecting the fault with the EM pulse (All values are in hexadecimal)

GPIO0DATA address (bit[15:0])	R0 (bit[31:24])	R1 (bit[15:0])	PSR (bit[31:24])
0010	10	0004	01
0090	90	0004	81
001C	1C	0005	01
00F0	F0	0004	81
03BC	BC	00C5	A1
07BC	BC	01C5	A1
1C5C	5C	0705	A1
3FFC	FC	0FC5	A1

We found the following rules concerning the faulty value in R0 and R1, assuming the GPIO0DATA address is encoded as $addr\,[31:0]$:

1. $R0\,[31:24] = addr\,[7:0]$. $R0\,[23:0]$ are always '0's.
2. $R1\,[15:4] =$ "00" $+\, addr\,[15:8] +$ "00". $R1\,[3:0] = 0x4$ if $addr\,[3:0] = 0x0$ or $0x8$. Otherwise, $R1\,[3:0] = 0x5$. "+" means concatenation.
3. If $addr\,[15:8] \neq 0x00$, bit[29] of the PSR is always set to 1. This bit is the carry/borrow sign. Bit[24] of the PSR is always 1 which indicated that the processor was running in Thumb state.

The faulty value was consistent and we could identify a specific relationship between the faulty value and the original value. It is similar to a shifting behavior on register R0 and register R1. These behaviors might be due to the remaining instructions in the pipeline or the bus error where the AHB returned wrong data.

5.5 Countermeasure

As discussed in Section 2.4, countermeasures could be divided into the fault tolerant countermeasure and the fault detection countermeasure. The fault detection countermeasure for the LDR instruction is exactly the same as what we did in our test program to detect a fault [61]. Therefore, only the fault tolerant countermeasure is considered in this section.

In the following sections, the duplication countermeasure proposed previously in [91] will first be examined. Then, we propose our own fault tolerant countermeasure verified with empirical results.

Recall test program one which is shown in listing 5.1 and assume the $LDR\ R4$ instruction is the attack target which therefore needs to be protected by the countermeasure. Moreover, all the verification was done on chip #2 with $V_p = 180\ V$, $F_{clk} = 100\ MHz$ and $V_{DD} = 3.3\ V$. These countermeasures presented in this section were designed based on the assumption that the attacker could only induce a single fault within 20 instructions and the countermeasure is only focused on the target instruction. These 20 instructions take 200 ns to execute, which is far less than the 1 us minimum period of the Riscure system as described in section 3.3.

5.5.1 Duplication Countermeasure

The duplication countermeasure was verified based on test code 1-d with the $LDR\ R4$ instruction duplicated as described by Moro et al.[91]. If the duplication countermeasure can thwart EM FIA, the program should never branch to the $_FAULT$ loop. The simulation result for this part of code is shown in Table 5.17. The first instruction after the duplicated $LDR\ R4$ was changed to a $MOVS\ R0,\ \#0$ since the address at which 0xFEDCBA98 was stored was changed to 0x78. According to equation 5.1, if the duplication countermeasure could not protect $R4$ from being faulted, the opcode of the instruction at address 0xA18 and 0xA1A will be output by the UART.

The fault injection experiment was performed for 100 rounds per D_{t2p}. The result is shown in Table 5.18. From this table, the fault could still be injected with slightly increased D_{t2p}. Also, the induced address fault followed equation 5.1. Compared with the result for test program 1-g shown in Table 5.8 where the address fault could be injected with D_{t2p} set to 284 ns, the fault started to be injected when D_{t2p} was set to 294 ns. This is likely because the fault was injected to the second $LDR\ R4$ instruction.

In summary, the duplication countermeasure for LDR instruction provides fault tolerant protection only if the target instruction is skipped or executed as NOP. Otherwise, the

Table 5.17: Simulation result for test code cm-1

Address (in hexadecimal)	Encoding(in hexadecimal)	Instruction
0x00000A14	0x4C18	$LDR\ R4, [PC,\ \#0x60]$
0x00000A16	0x4C18	$LDR\ R4, [PC,\ \#0x60]$
0x00000A18	0x2000	$MOVS\ R0,\ \#0$
0x00000A1A	0xBF00	NOP
......	......	Other instructions
0x00000A78	BA98	$DCW\ 0xBA98$
0x00000A7A	FEDC	$DCW\ 0xFEDC$

Table 5.18: Faulty value in R4 with duplication countermeasure

D_{t2p} (ns)	Faulty value	Count
274	No function fault	N/A
284	No function fault	N/A
294	BF002000	5
304	BF002000	8
314	BF002000	23
324	BF002000	23
334	No function fault	N/A
344	No function fault	N/A

attacker could inject the fault to the duplicated instruction which will not be corrected. As long as the attacker could inject the fault to the last $LDR\ R4$ instruction, no matter how many $LDR\ R4$ instructions are added, the fault is still able to be injected. Therefore, the duplication countermeasure does not work. Hence, another fault tolerant countermeasure was designed and will be discussed in the next section.

5.5.2 Fault Tolerant Countermeasure

The fault tolerant countermeasure not only aims at detecting the induced fault but also correcting it. Considering that the attacker could inject a function fault to any $LDR\ PC-relative$ instruction, a dual protection mechanism was applied in the fault tolerant countermeasure. The test program CM-1 shown in listing 5.6 illustrates how this countermeasure was designed. Instead of simply adding duplicated $LDR\ R4$ instruction, compare instructions were also added. Since only a single fault could be injected within 20 instructions, the attacker could only inject a fault to one of the $LDR\ R4$ instructions. Hence, if the fault is injected to the first $LDR\ R4$ instruction, the second one could be executed without the fault. If the attacker aims at injecting the fault to the second $LDR\ R4$ instruction,

the first one will be fault free and the program will not execute the second one. Thus, this program could correct the fault induced to the $LDR\ R4$ instruction.

```
1  _main
2    Assert trigger;
3      26*NOPs;
4      LDR R3, =(0xFEDCBA98);
5    LDR R4, =(0xFEDCBA98); Target instruction
6    CMP R3, R4; Check if R3 = R4;
7    BEQ _continue; Yes;
8    LDR R4, =(0xFEDCBA98); No.Execute the second LDR instruction
9    CMP R3, R4; Check if R3 = R4;
10   BNE _FAULT;
11 _continue
12     De-assert trigger;
13 ... Other instructions;
```

Listing 5.6: Test program CM-1

To verify this countermeasure, the experiment was conducted with D_{t2p} ranging from 274 ns to 414 ns and no function fault could be induced. Therefore, this countermeasure successfully corrected the induced fault.

5.5.3 The HardFault Handler As a Countermeasure

The previous two countermeasures don't provide a protection for an induced HardFault. An induced HardFault will force the processor jumps into the HardFault handler. As mentioned in Section 1.2.2, FIA consists of the fault injection and the fault exploitation phase. If the induced fault is not exploitable, the secret information could be protected. Therefore, the HardFault handler integrated in the Cortex-M0 is a possible built in countermeasure. By default, the HardFault handler is just an infinite loop which does not output anything to the attacker. The designer could add different functions to the HardFault handler. For example, the HandFault handler could be used to erase the cryptographic key. However, not every fault injected to the chip results in a HardFault. Therefore, the processor designer needs to carefully design under which condition, should the HardFault exception be triggered to cover more types of injected faults.

5.6 Summary

In this chapter, we analyzed the fault injected to LPC1114 which embeds an ARM Cortex-M0 processor. In summary, two different kinds of faults could be consistently injected. The first type of fault is defined as the function fault. In the function fault, HardFault exception is not triggered and the processor is still running in thread mode. Generally, the function fault is more exploitable compared with the HardFault. The function fault consists of the address fault and the data fault. The address fault is likely caused during the execution stage of the LDR instruction where the AHB samples the address and fetches the data from the associated memory location. The data fault is probably caused during the pipeline stall stage where the AHB writes back the data to the destination register. However, the function fault injected with different types of LDR instructions varies. With the address fault injected to the $LDR < Rt >, [< Rn >, \# < imme >]$, the destination register might be loaded with all '0's. Hence, this fault could be used to further develop DFA algorithms to break some cryptographic algorithms such as AES.

Another fault was the HardFault. With the revised HardFault handler, we successfully retrieved the faulty information from the stack. The complexity of the HardFault was analyzed. However, due to the lack of the detailed implementation of Cortex-M0, it is difficult to fully explore the induced HardFault. Table 5.19 lists all the instructions we have tested on LPC1114 and the fault injection result.

Table 5.19: Instructions that have been tested on LPC1114

Instruction	Fault injection result
$LDR < Rt >, [PC, \# < imme >]$	Function fault, HardFault
$LDR < Rt >, [< Rn >, \# < imme >]$	Function fault, HardFault
$LDMIA < Rn >!, < registers >$	Function fault, HardFault
$CMP < Rn >, < Rm >$	HardFault
$CMP < Rn >, \# < imm8 >$	HardFault

5.6.1 Comparison with Previous Research

Table 5.20 lists five research papers which had faults injected to different ARM processors. Paper [32] gave a general introduction on injecting faults to an ARM Cortex-A9 processor with hand made probe tip. However, they were unable to corrupt their target counter program but only observed graphics corruption or system crash. They did not provide a specific target instruction.

As detailed in Section 2.2.1.1, Korak et al. applied both clock glitch and power glitch attack on LPC1114 with a Cortex-M0 core [63]. The chip was running at 24 MHz (the maximum nominal clock frequency is 50 Mhz). The fault could be injected by glitching the clock with underpowering the processor to 1.2 V (the nominal supply voltage range is 1.8 V to 3.6 V). The fault could be injected in the fetch or execute stage. For the arithmetic instruction and branch instruction, the fault injected to the fetch stage prevented the target instruction from being executed. For LDR instruction, it caused the loaded value to be all '0's when attacking the fetch stage. This is unlike our experimental results where a function fault could not be injected by targeting the fetch stage of the LDR instruction. They further discovered that when the glitch was inserted during the "second execution cycle" of the LDR instruction, it caused the $LDR\ Rt\ [Rn]$ instruction to be executed as a $MOVS\ Rt, Rn$. This is unlike our experiment, where it was more likely to induce a data fault where several bits of the loaded data were flipped. Moreover, the "second execution cycle" is likely the pipeline stall stage (as discussed in Section 5.1) but the author did not analyze what the processor was doing in each cycle.

EM FIA was utilized to inject the fault to an ARM Cortex-M3 core [7]. The clock frequency was 56 MHz. The supply voltage was not specified by the author. They first analyzed the fault induced to a single load instruction and found that the loaded value was incorrect. However, the type of the load instruction was not specified. Moreover, they did not explicitly specify which pipeline stage was their target. In their second experiment, the target instruction was still a load instruction (likely a $LDR\ PC - relative$). They set the EM pulse to sweep over a 20 ns window with 200 ps per step. In each step, they performed a scan over the chip. They observed three different types of faults: Interrupt/exception triggered, crash of the microcontroller, and a faulty value loaded into the destination register. However, they did not specify where this 20 ns time window was located. Hence, there was no timing analysis performed. Similarly, in their third experiment where an additive loop was utilized as the target, they swept through a 4 us window (this additive loop took around 3.5 us to finish) with 200 ps per step. However, they just simply listed their result without further timing analysis. When they analyzed the effect of the pulse voltage, the target instruction was the $LDR\ PC - relative$ instruction which was also utilized in our experiment. However, they only found with a higher pulse voltage, the hamming weight of the loaded value was increased.

Another research paper proposed a method to bypass the secure boot of an ARM processor [6]. However, the processor core was not specified while the architecture was likely the ARMv8-M. The attack algorithm was introduced in Section 2.3.2. The main idea of the attack was to use voltage glitch to force the destination register of LDR or $LDMIA$ instruction to be changed to the PC. Here we mainly discuss their attack result.

Table 5.20: Research papers with FIA result on ARM

Paper	Core	Target instruction/program	Fault type	Fault injection technique
[32]	Cortex-A9	Counter program	Graphics error/System crash	EM
[63]	Cortex-M0	$ADDS$, LDR, STR, BEQ	Instruction skip, $LDR \rightarrow MOV$	Voltage/Clock glitch
[7]	Cortex-M3	LDR	Load faulty data	EM
[6]	N/A	LDR, $LDMIA$	Change destination register to PC	Voltage glitch
Our thesis	Cortex-M0	$LDR\ PC-relative$, $LDR\ <Rt>, [<Rn>, \#<imme>]$, $LDMIA$	Data fault, address fault, combined fault	EM

The attack was implemented by using voltage glitch. The chip was clocked at 800 MHz and the supply voltage was set to 1.1 V (0.1 V under the nominal value). To make the fault injection more effective, the capacitors were also removed from the PCB. They used two test programs to mimic the behavior of copying data from external memory to the volatile internal memory. The attack was performed by different combinations of glitch length (randomly chosen between 700 ns to 1000 ns), glitch voltage (randomly chosen between -1.4 V to -1.0 V) and glitch delay (randomly chosen between 30 us to 35 us).

When LDR instruction was utilized as the target, there was only 1 out of 10000 combinations of glitch length and glitch voltage led to a successful attack. The success rate for glitching the $LDMIA$ instruction was higher. There were 27 out of 10000 combinations of glitch length and glitch voltage led to a successful attack. The author further hypothesized that the $LDMIA$ instruction was easier to be faulted because it required less bits to be flipped to change the destination register to PC. Compared with our research work, we were unable to flip bits in the opcode which resulted in another valid instruction. Some bits might be flipped, resulting in an illegal instruction and triggered the HardFault exception.

We also proposed countermeasures to thwart EM FIA targeting the $LDR\ PC-relative$ instruction. Unlike our proposed countermeasure, the instruction duplication countermeasure for $LDR\ PC-relative$ instruction designed in [91] did not provide sufficient protection in our experiments. Therefore, understanding the fault injection mechanism is important in designing efficient countermeasures.

The next chapter will introduce the charge-based fault model with empirical verification on both the PIC16F687 and LPC1114.

Chapter 6

Charge-based Fault Model

In this chapter, the proposed charge-based fault model is introduced. The definition of the charge-based fault model is presented in Section 6.1 along with a methodology of how to empirically verify the model. In the following two sections, we demonstrate our empirical results on verifying the fault model on the PIC16F687 and the LPC1114.

6.1 Charge-Based Fault Model

As discussed in Section 2.2.2.1, the delay fault model [7] and the sampling fault model [4] were previously researched.

Since the delay fault model has already been illustrated to be not practical in EM FIA [4], we mainly focus on checking whether the sampling fault model was practical with our experiments.

In our experiments, the fault model parameters that affect the fault injectability are combinations of the supply voltage, the clock frequency, and the position of the EM pulse within a specific clock cycle. Based on this finding, we proposed the charge-based fault model.

Critical charge is defined as the minimum amount of charge collected by a circuit node which finally causes an “upset” [105]. In the digital circuit, ‘1’ or ‘0’ at a node represents the capacitor at this node is charged or discharged. With a higher clock frequency, the time to charge/discharge the capacitor is reduced. Additionally, with a lower supply voltage, the current to charge/discharge the capacitor is reduced. Therefore, the capacitor at this

node may not be fully charged/discharged with a faster clock or smaller supply voltage. Based on these findings, we summarized the charge-Based fault model as follows:

Definition. ***Charge-Based Fault Model:*** *The EM pulse could increase/decrease the charge (either via noise or power-ground network) accumulated at a specific node. This change of the charge results in the change of the state at this node. If this change of charge persists when a clock edge arrives and it is greater than the critical charge, the fault is injected. To reduce the critical charge, one can either increase the clock frequency or decrease the supply voltage.*

Figure 6.1 illustrates how the fault is injected based on the charge-based fault model. When the EM pulse (indicated by the red arrow) is injected, it modifies the accumulated charge at the capacitor before the next flip flop. Assume the initial charge at this node is Q_i, when the EM pulse is induced, Q_i is decreased or increased. If this node is not fully charged/discharged, the EM pulse might be able to force a '0'/'1'(when it should be '1'/'0') to be latched into the next flip flop and finally causes the fault.

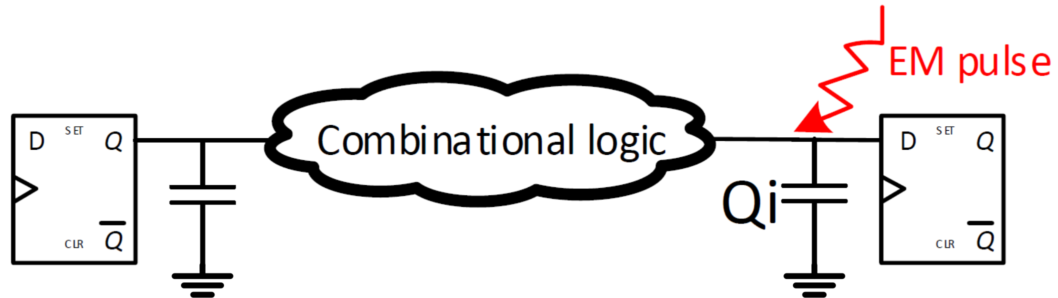


Figure 6.1: Charge based fault model

In the following sections, we will present empirical validation of this charge-based fault model on both the PIC16F687 and the LPC1114 microcontroller. Figure 6.2 illustrates our methodology for verifying the charge-based fault model. The methodology is proposed as follows:

1. Assume a fault is injected at a specific D_{t2p} with a frequency F_{clk} and a supply voltage V_{DD}.
2. We find all possible edges that might cause this fault based on the timing and also the data sheet of the chip.

3. We calculate the distance from the pulse to the edge. For example, in Figure 6.2, the distance from the pulse to edge 1 is $D_1 - D_{t2p}$ where $D_1 = N \div F_{clk}$.

4. We increase the delay setting from D_{t2p} to $\widehat{D_{t2p}}$.

5. Calculate the new clock frequency as $\widehat{F_{clk}} = N \div (\widehat{D_{t2p}} + D_1 - D_{t2p})$. Check if the fault could be injected. In this way, the delay from the pulse to edge 1 remains the same.

6. If the fault could not be injected, gradually reduce the supply voltage until the fault could be injected again. Record this $\widehat{V_{DD}}$.

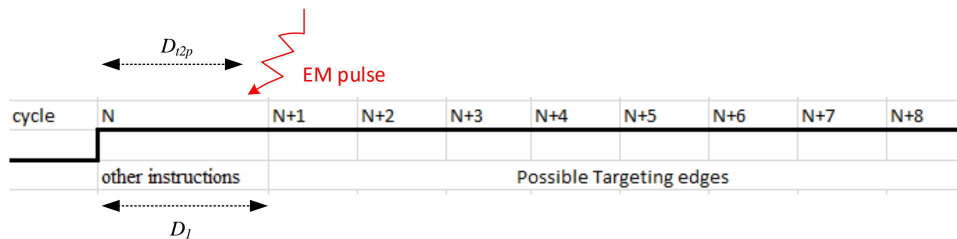


Figure 6.2: Example for empirical verification of the charge-based fault model

6.2 Empirical Verification on PIC16F687

To empirically check the fault model on PIC16F687, we utilized the program in listing 6.1. The probe was located at the best location as shown in Table 4.2.

```
1   MOVLW 0x20
2   MOVWF FSR ;Use indirect addressing mode.
3   LOOP2
4   BCF STATUS, RP1
5   BCF STATUS, RP0
6   MOVLW 0x20
7   MOVWF PORTC    ; Assert the trigger.
8   NOP
9   NOP
10  NOP
11  MOVLW 0x55 ; Move literal to W register before writing it to SRAM
```

```
12      MOVWF 0x20
13      MOVLW 0x01   ; Target MOVLW instruction
14      MOVWF 0x21
15      NOP
16      MOVLW 0x00
17      MOVWF PORTC ; De-assert the trigger
18      CALL Delay1
19      BTFSS FSR,7
20      GOTO LOOP2
```

Listing 6.1: Program used for empirically verify the charge-based fault model on PIC16F687

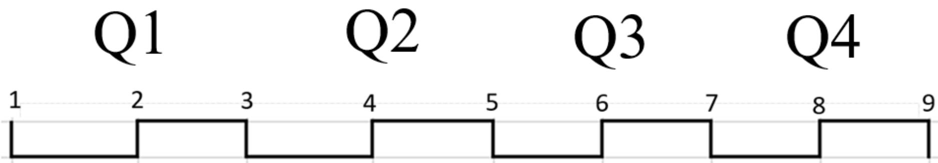


Figure 6.3: The nine clock edges in the prefetch cycle of the target instruction for PIC16F687

The initial frequency was set at 52 MHz and the supply voltage was set at 5 V. A fault could be injected with a 304 ns delay setting. The fault was injected because the $MOVLW$ $0x01$ instruction was replaced with the NOP instruction. Finally, the faulty value stored at address 0x21 was 0x55. The system utilized in this experiment was the BPS202 and V_p was set to 500 V.

Since in PIC16F687, one instruction cycle consists of four clock cycles and we could not confirm which clock edge should be the correct target and the data sheet of the PIC16F687 uses both rising edge and falling edge of the clock as the triggering edge in their figures, we considered all the nine edges within this instruction cycle where the $MOVLW$ $0x01$ instruction was fetched to make our verification to be more robust. These nine edges are shown in Figure 6.3. The first edge might belong to the previous instruction cycle. However, we still considered it as one of our possible edges.

Table 6.1 presents the experimental result for verifying the charge-based fault model on PIC16F687. A negative value indicates that the EM pulse was induced after the associated edge. Initially, a fault was able to be injected with $F_{clk} = 52\ MHz$ and $V_{DD} = 5\ V$ with $D_{t2p} = 304\ ns$. Based on this, we calculated the distance between the pulse to all the nine possible edges. Then, the delay setting was increased to 344 ns and we calculated nine different clock frequencies which could ensure the distance from the pulse to the associated

Table 6.1: Experiment result for the charge-based fault model verification on PIC16F687, F_{clk} in MHz, delay in ns and V_{DD} in V

F_{clk}	D_{t2p}	Distance between EM pulse to the Nth edge									FIA/5 V	FIA/V_{DD}
		1	2	3	4	5	6	7	8	9		
52	304	3.7	13.3	22.9	32.5	42.2	52.8	61.4	71.0	80.6	Yes	Yes/5
46.01	344	**3.7**	–	–	–	–	–	–	–	–	No	Yes/4.2
46.18	344	2.5	**13.3**	–	–	–	–	–	–	–	No	Yes/4.2
46.33	344	1.4	–	**22.9**	–	–	–	–	–	–	No	Yes/4.4
46.48	344	0.2	–	–	**32.2**	–	–	–	–	–	No	Yes/4.4
46.61	344	−0.7	–	–	–	**42.2**	–	–	–	–	No	Yes/4.4
46.74	344	−1.7	–	–	–	–	**52.8**	–	–	–	No	Yes/4.7
46.87	344	−2.6	–	–	–	–	–	**61.4**	–	–	No	Yes/4.7
46.99	344	−3.5	–	–	–	–	–	–	**71.0**	–	No	Yes/4.8
47.10	344	−4.3	–	–	–	–	–	–	–	**80.6**	No	Yes/4.9

edge was the same. However, we found that, after we reduced the clock frequency, no fault could be injected after 5 minutes (approximately 2.5 million EM pulses injected within this time duration). Therefore, it illustrated the first part of the charge-based fault model. The reduced frequency ensured that there was more time to charge/discharge the capacitor which finally made the critical charge increased. In this case, two bits in the opcode need to be reset to result in the faulty instruction as NOP. Hence, the capacitor accumulated more charges when the clock frequency was reduced, which made it to be more difficult to be discharged to '0' with the EM pulse.

In the meanwhile, if we reduce the supply voltage to a certain level, the fault could be injected again. Additionally, we had run the chip for over 30 minutes to ensure the correct functionality with the reduced supply voltage. Thus, it demonstrated the second part of the charge-based fault model where the reduced supply voltage decreased the charge current and reduced the critical charge. We could also tell from the Table that, lower supply voltage is required for lower frequency to inject the fault. Additionally, the injected fault was the same as we had initially.

We also need to consider the 10 ns jitter introduced by the BPS202 system. Hence, the distance between the pulse to the associated edge might be changing all over the time. However, this does not change the feature that the actual delay between the trigger to the pulse was increased by 10 ns every time when we increase the D_{t2p} by 10 ns. Moreover, based on the analysis of the Flt_inst in Chapter 4, it is reasonable to assume that the distance from the pulse to the associated clock edge is the same if two fault injection experiments targeting the same test program and the same Tgt_inst have the same Flt_inst. For example, consider the first row in Table 6.1, the D_{t2p} was set as 304 ns. If the jitter

was taken into consideration, the actual D_{t2p} should be 304 + x ns and $x \in [-5, 5]$ when the fault was injected. The number of clock cycles from the trigger to the second edge was 4*4+0.5 = 16.5 cycles. Therefore, we could calculate the delay from the pulse to the second edge as 1000/52*16.5 - (304 + x) = 13.3 - x ns. After we increased the delay setting to 344 ns, to ensure the distance between the pulse and the second edge was the same, the delay from the trigger to the second edge could be calculated as 344 + x + 13.3 - x = 357.3 ns. Hence, the new frequency could be calculated as 16.5 * 1000/357.3 = 46.18 MHz. Therefore, the jitter does not affect the verification of the fault model.

6.3 Empirical Verification on LPC1114

The program used to empirically verify the charge-based fault model was the program 1-d shown in Chapter 5. For simplicity, it is shown in listing 6.2. The experiment was performed with chip #1 on the automated platform with the best location indicated in red rectangle in Figure 5.5. Initially, we set F_{clk} to 104.4 MHz and V_{DD} to 3.3 V. When D_{t2p} was set to 284 ns, the fault was able to be injected. Then, the faulty value was sent out through UART. The timing diagram is presented in Figure 6.4.

```
1  _main
2    Assert trigger;
3      26*NOPs;
4
5    LDR R3, =(0xFEDCBA98);
6    LDR R4, =(0xFEDCBA98);
7    16*NOPs;
8    CMP R3, R4;
9    6*NOPs;
10   BNE _FAULT;
11   De-assert trigger;
12   B _main;
```

Listing 6.2: Test program 1-d used to empirically verify the charge-based fault model on LPC1114

Based on our analysis in Chapter 5, the fault was injected during the execution stage. However, to provide a complete validation for our charge-based fault model. We considered all the edges at the end of the clock cycle from cycle 26 to cycle 31. The experimental result is shown in Table 6.2. The Nth edge in Table 6.2 refers to the end of the $(N+25)$th cycle in Figure 6.4. For example, the fourth edge in Table 6.2 is the end of the 29th cycle in Figure 6.4 where the address is sampled by the AHB bus. The fault injection system

Edges							1	2	3	4	5	6
Cycles	0	1	2-22	23	24	25	26	27	28	29	30	31
Trigger PIO0_2												
Fetch	NOP2 NOP3		Fetch NOPs		NOP26 LDR R3		LDR R4 NOP27		STALL	NOP28 NOP29	STALL	
Decode	NOP1	NOP2	Decode NOPs	NOP24	NOP25	NOP26	LDR R3	LDR R4		NOP27		NOP28
Execute	trigger	NOP1	Execute NOPs	NOP23	NOP24	NOP25	NOP26	LDR R3		LDR R4		NOP27

Figure 6.4: Test code 1-d used for empirical verification of the charge-based fault model

applied was the BPS201 and V_p was fixed at 180 V. From Table 6.2, the EM pulse was induced in cycle 30. Considering the 14 ns jitter brought by the BPS201 system, the pulse could be affecting either the end of cycle 29 or the end of cycle 30.

Table 6.2: Experimental result for the charge-based fault model verification on LPC1114, F_{clk} in MHz, delay in ns and V_{DD} in V

F_{clk}	D_{t2p}	Distance between EM pulse to the Nth edge						FIA/3.3 V	FIA/V_{DD}
		1	2	3	4	5	6		
104.4	284	-35.0	-25.4	-15.8	-6.2	3.4	12.9	Yes	Yes/3.3
100.37	294	**-35.0**	–	–	–	4.9	–	No	Yes/2.06
100.51	294	–	**-25.4**	–	–	4.5	–	No	Yes/2.07
100.65	294	–	–	**-15.8**	–	4.1	–	No	Yes/2.08
100.77	294	–	–	–	**-6.2**	3.7	–	No	Yes/2.09
100.89	294	–	–	–	–	**3.4**	–	No	Yes/2.1
101.00	294	–	–	–	–	3.0	**12.9**	No	Yes/2.11

From Table 6.2, we could find similar properties as we discussed in Section 6.2. When the clock frequency was reduced, the fault could no longer be injected after 5 minutes even though we kept the distance between the pulse and the associated edge the same. Then, the supply voltage was reduced to check if the same fault could be injected again. Also, the chip could run fault free for 30 minutes with each reduced supply voltage and the associated clock frequency when the EM pulse injection was not applied. Only by reducing the supply voltage, the fault could be injected again. Moreover, it requires a lower supply voltage to inject the fault if the clock frequency is smaller. The fault type was the same as the result of the first row.

6.4 Summary

In this chapter, we proposed the charge-based fault model. This fault model utilizes the critical charge concept from the single event upset. The delay fault model suggests that the induced EM pulse introduces a delay [7] and the sampling fault model states that the induced EM pulse causes a perturbation of the data [4]. In our charge-based fault model, the induced EM pulse could charge/discharge the capacitor at a specific node. The charge/discharge must be greater than the critical charge to induce the fault. The charge-based fault model is related to the circuit frequency and supply voltage.

Moreover, we empirically verified our fault model on two different microcontrollers. However, due to the lack of tools to check the real time status of the register value in the microcontroller, it is impossible to directly validate the fault model. For example, for PIC16F687, we were unable to directly check if the bits in the instruction were flipped but only observe the final faulty value at the associated memory address to infer the correct faulty instruction.

It was obvious that the sampling fault model did not match our results since when the distance from the pulse to every possible edge was kept the same, with a reduced frequency, the fault could no longer be injected. When we reduce the supply voltage, the fault could be injected again. This is unlike previous research where Ordas et al. stated that for a sampling fault model, the probability of injecting the fault over the whole clock cycle was independent of the clock frequency [4]. In our fault model, the clock frequency plays a vital role in the fault injectability.

There were some other research papers which also suggested that using a lower supply voltage could help either make the induced fault more reproducible [90], or make the induced fault to be more effective [6]. However, these two FIA experiments were performed with clock glitch.

Chapter 7

Conclusions

This chapter concludes the thesis. First, a brief summary of this dissertation is presented in Section 7.1 along with the discussion about the difference of the fault injected to the PIC16F687 and the LPC1114. The limitations are also investigated. The contributions are summarized in Section 7.2. The possible future research directions are presented in Section 7.3.

7.1 Summary, Discussion and Limitation

Since the first attack proposed in [51], FIA becomes a severe threat to embedded devices and hardware with the development of many fault injection techniques. Among these techniques, EM FIA provides a moderate spatial resolution, low potential of damaging the device and a simpler setup. However, there remains a lack of understanding on the injected fault and the associated countermeasures.

One of the main challenges of EM FIA is to completely characterize the injected fault. Most previous research did not provide a detailed analysis of the induced fault, resulting in insufficient countermeasures to thwart EM FIA. This challenge motivated the in-depth research in EM FIA targeting embedded devices.

In this thesis, we presented our experimental results on two microcontrollers. As discussed in Chapter 5, the faults injected to the PIC16F687 and the LPC1114 are different. Compared with the PIC16F687, the induced fault on the LPC1114 is more difficult to explore. The target instruction is replaced with a faulty instruction on the PIC16F687. However, the same kind of fault was not injected with the LPC1114. The instructions

from the PIC16F687 share similar encoding, thus the hamming distance is small between similar instructions. For instance, the hamming distance between $XORWF$ and $IORWF$ is only one. Moreover, the hamming distance between NOP and other instructions is also small. This may contribute to the reason that the instruction replacement fault is more likely to be injected on the PIC16F687. In contrast, the ARMv6-M architecture has a different Instruction Set Architecture (ISA) and will HardFault on illegal instructions and other exceptions. We did not observe the instruction replacement fault in our experiments, but observed many HardFaults. Similar results were found by previous research [61].

Previous research also includes attacks which utilized system failures or design flaws such as "Row-hammer" which could corrupt the data in adjacent rows of DRAM memory [106], "Meltdown" which utilizes the out of order execution of the processor to access the kernel memory [107], and "Spectre" which exploits the branch prediction to access arbitrary memory [108]. However, these attacks require loading the attacker's own software onto the target chip. On the contrary, for embedded microcontrollers, the attacker may not have the ability to load their own software onto the microcontroller. Hence, these software-based attacks are outside the scope of this dissertation.

There remain some limitations in this thesis. The major limitations are summarized below:

1. The faulty instruction identified for the PIC16F687 could not be directly verified due to the lack of direct access to the instruction register. However, this restriction also applied to previous research on FIA [7, 63, 59]. Similarly, faults injected to the LPC1114 and the charge-based fault model were only verified empirically within the limits of our equipment.

2. Some analysis for the faults injected to the LPC1114 ended without a complete verification due to the damaged equipment. For example, the explanation for not being able to inject the fault to register R4 for the $LDMIA\ R0!,\ R1, R3, R4$ is unknown and was not validated empirically. However in general, due to the lack of observability of the fault induced from the circuit level, the induced fault might be complex and extremely hard to explain, which is similar to previous research [76].

The limitations for the equipment were previously discussed in Section 3.3. Due to these limitations, the attacks discussed in Chapter 4 were based on the assumption that the attacker knows the software and also the timing where the EM pulse should be delivered. Assume now we have a perfect EM fault injection system with a high timing resolution for adjusting the D_{t2p} (such as 100 ps accuracy utilized by Dehbaoui et al. [49]). With the

passive SCA, it may be feasible to identify the last round of AES. The artificially inserted trigger signal could be removed and the correct timing for delivering the EM pulse could be identified by scanning through the last round by adjusting the D_{t2p} with 100 ps per step. A similar approach was utilized in [49] (see Section 4.9.1). Even if we have no access to the software or if the AES code is obfuscated, the timing for delivering the EM pulse could still be identified by analyzing the associated faulty output. For example, given the new attack from Section 4.6.2 where a single EM pulse reveals the entire AES key if the associated Flt_inst is the NOP instruction, the attacker could scan the EM pulse over each invocation of the obfuscated AES or unknown code, checking the possible revealed AES key derived from the faulty ciphertext (by running AES on the possible key with same plaintext) until the correct AES key is found (which matches the correct ciphertext). Alternatively, if the obfuscated code is implemented along with fault tolerant countermeasures, the EM FIA system may have more difficulty trying to generate EM pulses to fault the target instruction as well as additional EM pulses to fault the countermeasures (assume the attacker has access to multiple EM FIA systems). Thus, countermeasures in addition to obfuscation are likely important.

7.2 Contributions

In-depth research of EM FIA on two different embedded microcontrollers was presented in this thesis. The contributions for this research work are summarized below.

The first contribution is a complete instruction-centric EM FIA methodology for closed-source microcontrollers (see Section 4.9 for detailed description). Previous research did not suggest an in-depth methodology for characterizing the microcontroller at the instruction level followed by proposing instruction specific countermeasures for the exploitable fault. Some previous research analyzed the underlying silicon geometries [7, 59] without focusing on the faulty behavior of the associated instruction. This methodology is important for evaluating the security vulnerabilities for closed-source microcontrollers since it develops specific countermeasures for any possible attacks focused on the associated instructions.

Second, we provide for the first time a statistical characterization of faulty instructions (see Chapter 4). We show the probability of different faulty instructions and suggest that it is likely related to the address of the target instruction and also the delay between the EM pulse to a specific clock edge which is not pointed out in the previous research. Moreover, previous research did not statistically analyze all the possible faulty instructions [95, 59]. Identifying all the possible faulty instructions is critical for designing the associated instruction specific countermeasures. For example, the $MOVLW$ instruction might be replaced

with a $GOTO$ or two other instructions (see Section 4.4.3). If this faulty instruction is not taken into consideration, the added countermeasure might be skipped and the fault is still injected. Unlike previous research, the countermeasures include protection from all statistically possible faults which may occur.

The third contribution is a new instruction specific attack on AES for a microcontroller. With the faulty instruction revealed, we proposed two AES attacks with a two-step methodology targeting the $XORWF$ instruction (see Section 4.5.1.2 for attack algorithm 1-a and 1-b and 4.7.1.2 for 1-c) and the $DECFSZ$ instruction (see Section 4.6.2.2 for attack algorithm 2). Unlike AES attacks with EM FIA proposed in [74, 49], our attacks are less complex since we do not need to solve any non-linear equations or perform any exhaustive search. This new attack highlights that microcontrollers running without countermeasures are susceptible to EM FIA with very few number of EM pulses required.

Fourth, we provide a more robust understanding of the fault with detailed timing analysis combined with the action of the microcontroller when the fault is injected (see Chapter 5). With this understanding, a fault tolerant countermeasure for the LDR instruction is provided for the first time. This countermeasure is unlike previous research where only fault detection is provided [61].

The fifth contribution is the charge-based fault model. Unlike the fault models proposed in previous research [7, 4], the charge-based fault model suggests that the induced EM pulse could change the charge accumulated at a circuit node. It also considers the clock frequency and supply voltage which could affect the fault injectability. We could not inject a fault with the fault models proposed in previous research [7, 4]. This charge-based fault model is important in providing a guideline on how to inject the fault with EM pulse on some embedded microcontrollers.

7.3 Future Research Directions

The experiments done on PIC16F687 showed that cryptographic algorithms were vulnerable under FIA. However, the trigger signal was artificially added to deliver the EM pulse at the correct timing in the AES-128 attack proposed in Chapter 4. Therefore, one further work direction is to remove the trigger but use passive SCA to help find the correct timing to deliver the EM pulse. Researchers from Riscure had already successfully used this methodology to find out when to inject the fault [25]. However, if the delay between the pulse to the end of Q2 cycle is changed, the faulty instruction might also be changed. Thus, the difficulty of running a successful attack might be increased significantly.

Public key cryptography is widely used in authentication. While an attack on the symmetric key cipher was done on the PIC16F687, another research work that could be done with PIC16F687 is to perform an attack targeting a public key cipher. For example, the RSA attack demonstrated in [51] could be applied with our results on the PIC16F687. For example, likely the ALU instructions in PIC16F687 will be used in calculating M_p or M_q. Hence, we can fault these instructions and get a faulty M_p or M_q. This should also be feasible with inducing the observed faults on the LPC1114.

The processor with an ARM core is widely used in different markets [109]. Therefore, the security inside these cores are critical. With the current result on the LPC1114, another research direction is to conduct a real attack. It is possible to run a DFA attack with other cryptographic algorithms running on the LPC1114. Moreover, EM FIA on other similar but more advanced ARM cores is also a promising research direction. The ARM Trustzone technology utilizes an additional bit in the address bus to identify a secure/non-secure world request [110]. The address fault induced to the Cortex-M0 might be applied to other ARM processors to attack the Trustzone technology where the attacker may gain access to the secure world from a non-secure world request.

Future research could also focus on countermeasures in hardware. For both chips utilized in our experiment, there is no access to the hardware. RISC-V is an open source ISA with many giant tech companies as its members such as Google, NVIDIA, NXP,etc [111]. Therefore, it may be possible to research adding hardware countermeasures to processors implementing the RISC-V ISA on an FPGA emulation platform. For example, a new pipeline stage was added to allow the processor to encrypt and decrypt the instructions before decoding them [112]. Moreover, a simulation was done on attacking the hidden registers in RISC-V processor with FIA [113]. However, the security analysis on RISC-V was still limited. Therefore, physically injecting faults to the RISC-V based processor and analyzing its faulty behavior are of interest [114]. Future research would involve statistically characterizing faults on other embedded microcontrollers/microprocessors in addition to exploiting faults in attacks and countermeasures.

References

[1] M. Joye and M. Tunstall, *Fault analysis in cryptography*, vol. 7. Springer, 2012.

[2] T. Hummel, "Exploring effects of electromagnetic fault injection on a 32-bit high speed embedded device microprocessor," Master's thesis, University of Twente, 2014.

[3] C. H. Gebotys, "Em, lasers and methodologies," tech. rep., University of Waterloo, 2016.

[4] S. Ordas, L. Guillaume-Sage, and P. Maurine, "Electromagnetic fault injection: the curse of flip-flops," *Journal of Cryptographic Engineering*, pp. 1–15, 2016.

[5] C. H. Kim and J.-J. Quisquater, "New differential fault analysis on aes key schedule: Two faults are enough," in *International Conference on Smart Card Research and Advanced Applications*, pp. 48–60, Springer, 2008.

[6] N. Timmers, A. Spruyt, and M. Witteman, "Controlling pc on arm using fault injection," in *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pp. 25–35, IEEE, 2016.

[7] N. Moro, A. Dehbaoui, K. Heydemann, B. Robisson, and E. Encrenaz, "Electromagnetic fault injection: towards a fault model on a 32-bit microcontroller," in *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2013 Workshop on*, pp. 77–88, IEEE, 2013.

[8] "User manual ici probe ic injection probe." https://www.langer-emv.de/en/product/ic-side-channel-analysis/94/ici-01-l-eft-set-ic-em-pulse-injection-langer-pulse/821.

[9] K. Amin, C. Gebotys, M. Faraj, and H. Liao, "Analysis of dynamic laser injection and quiescent photon emissions on an embedded processor," in *International Conference on Physical Assurance and Inspection of Electronics*, 2019.

[10] M. T. Inc, "Picmicro mid-range mcu family reference manual." http://ww1.microchip.com/downloads/en/devicedoc/33023a.pdf, 1997. original document from Microchip Technology Inc.

[11] M. T. Inc, "Pic16f631/677/685/687/689/690 data sheet." http://ww1.microchip.com/downloads/en/DeviceDoc/50002227C.pdf, 2007. original document from Microchip Technology Inc.

[12] "Arm cortex™-m programming guide to memory barrier instructions." http://infocenter.arm.com/help/topic/com.arm.doc.dai0321a/DAI0321A_programming_guide_memory_barriers_for_m_profile.pdf, 2012. original document from ARM.

[13] L. Zussa, A. Dehbaoui, K. Tobich, J.-M. Dutertre, P. Maurine, L. Guillaume-Sage, J. Clediere, and A. Tria, "Efficiency of a glitch detector against electromagnetic fault injection," in *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, pp. 1–6, IEEE, 2014.

[14] J. Breier, S. Bhasin, and W. He, "An electromagnetic fault injection sensor using hogge phase-detector," in *2017 18th International Symposium on Quality Electronic Design (ISQED)*, pp. 307–312, IEEE, 2017.

[15] C. Deshpande, B. Yuce, L. Nazhandali, and P. Schaumont, "Employing dual-complementary flip-flops to detect emfi attacks," in *2017 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*, pp. 109–114, IEEE, 2017.

[16] D. El-Baze, J.-B. Rigaud, and P. Maurine, "An embedded digital sensor against em and bb fault injection," in *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pp. 78–86, IEEE, 2016.

[17] M. T. Inc, "Advanced encryption standard using the pic16xxx." http://ww1.microchip.com/downloads/en/Appnotes/00821a.pdf, 2002. original document from Microchip Technology Inc.

[18] Y. Zhou and D. Feng, "Side-channel attacks: Ten years after its publication and the impacts on cryptographic module security testing.," *IACR Cryptology ePrint Archive*, vol. 2005, p. 388, 2005.

[19] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Annual International Cryptology Conference*, pp. 388–397, Springer, 1999.

[20] Z. Chen and Y. Zhou, "Dual-rail random switching logic: a countermeasure to reduce side channel leakage," in *International Workshop on Cryptographic Hardware and Embedded Systems*, pp. 242–254, Springer, 2006.

[21] S. Mangard, T. Popp, and B. M. Gammel, "Side-channel leakage of masked cmos gates," in *Cryptographers' Track at the RSA Conference*, pp. 351–365, Springer, 2005.

[22] M. Bucci, R. Luzzi, M. Guglielmo, and A. Trifiletti, "A countermeasure against differential power analysis based on random delay insertion," in *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on*, pp. 3547–3550, IEEE, 2005.

[23] F. Bao, R. Deng, Y. Han, A. Jeng, A. Narasimhalu, and T. Ngair, "Breaking public key cryptosystems on tamper resistant devices in the presence of transient faults," in *Security Protocols*, pp. 115–124, Springer, 1998.

[24] C. Giraud, "Dfa on aes," in *International Conference on Advanced Encryption Standard*, pp. 27–41, Springer, 2004.

[25] N. Timmers and A. Spruyt, "Bypassing secure boot using fault injection." https://www.riscure.com/publication/bypassing-secure-boot-using-fault-injection/, 2016.

[26] A. Barenghi, L. Breveglieri, I. Koren, and D. Naccache, "Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures," *Proceedings of the IEEE*, vol. 100, no. 11, pp. 3056–3076, 2012.

[27] I. LABS, "Security requirement for embedded devices - what is really needed?." http://www.iconlabs.com/prod/security-requirements-embedded-devices---what-really-needed, 2017.

[28] C. H. Gebotys, *Security in embedded devices*. Springer Science & Business Media, 2009.

[29] A. Juels, "Rfid security and privacy: A research survey," *IEEE journal on selected areas in communications*, vol. 24, no. 2, pp. 381–394, 2006.

[30] S. Ravi, A. Raghunathan, P. Kocher, and S. Hattangady, "Security in embedded systems: Design challenges," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 3, no. 3, pp. 461–491, 2004.

[31] B. Jun, "Protecting consumer electronics." https://www.rambus.com/wp-content/uploads/2015/08/RSA2008_ProtectingCE.pdf, 2008.

[32] R. Velegalati, R. Van Spyk, and J. van Woudenberg, "Electro magnetic fault injection in practice," in *International Cryptographic Module Conference*, 2013.

[33] K. Jeong, Y. Lee, J. Sung, and S. Hong, "Security analysis of hmac/nmac by using fault injection," *Journal of Applied Mathematics*, vol. 2013, 2013.

[34] S. Mangard, "A simple power-analysis (spa) attack on implementations of the aes key expansion," in *International Conference on Information Security and Cryptology*, pp. 343–358, Springer, 2002.

[35] J.-J. Quisquater and D. Samyde, "Electromagnetic analysis (ema): Measures and counter-measures for smart cards," *Smart Card Programming and Security*, pp. 200–210, 2001.

[36] Z. Martinasek, V. Zeman, and K. Trasy, "Simple electromagnetic analysis in cryptography," *International Journal of Advances in Telecommunications, Electrotechnics, Signals and Systems*, vol. 1, no. 1, pp. 13–19, 2012.

[37] M. Backes, M. Dürmuth, S. Gerling, M. Pinkal, and C. Sporleder, "Acoustic side-channel attacks on printers.," in *USENIX Security symposium*, pp. 307–322, 2010.

[38] S. Mangard, E. Oswald, and T. Popp, *Power analysis attacks: Revealing the secrets of smart cards*, vol. 31. Springer Science & Business Media, 2008.

[39] F.-X. Standaert, "Introduction to side-channel attacks," in *Secure Integrated Circuits and Systems*, pp. 27–42, Springer, 2010.

[40] O. Kömmerling and M. G. Kuhn, "Design principles for tamper-resistant smartcard processors.," *Smartcard*, vol. 99, pp. 9–20, 1999.

[41] S. P. Skorobogatov and R. J. Anderson, "Optical fault induction attacks," in *International workshop on cryptographic hardware and embedded systems*, pp. 2–12, Springer, 2002.

[42] S. P. Skorobogatov, "Semi-invasive attacks: a new approach to hardware security analysis," 2005.

[43] H. Liao and C. Gebotys, "Methodology for em fault injection: Charge-based fault model," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 256–259, IEEE, 2019.

[44] B. W. Johnson, "An introduction to the design and analysis of fault-tolerant systems," *Fault-tolerant computer system design*, vol. 1, pp. 1–84, 1996.

[45] O. Benot, "Fault attack," *Encyclopedia of Cryptography and Security*, pp. 452–453, 2011.

[46] A. Barenghi, G. M. Bertoni, L. Breveglieri, M. Pellicioli, and G. Pelosi, "Fault attack on aes with single-bit induced faults," in *Information Assurance and Security (IAS), 2010 Sixth International Conference on*, pp. 167–172, IEEE, 2010.

[47] N. Bagheri, N. Ghaedi, and S. K. Sanadhya, "Differential fault analysis of sha-3," in *International Conference in Cryptology in India*, pp. 253–269, Springer, 2015.

[48] H. Choukri and M. Tunstall, "Round reduction using faults," *FDTC*, vol. 5, pp. 13–24, 2005.

[49] A. Dehbaoui, A.-P. Mirbaha, N. Moro, J.-M. Dutertre, and A. Tria, "Electromagnetic glitch on the aes round counter," in *International Workshop on Constructive Side-Channel Analysis and Secure Design*, pp. 17–31, Springer, 2013.

[50] Y. Yao, M. Yang, C. Patrick, B. Yuce, and P. Schaumont, "Fault-assisted side-channel analysis of masked implementations," in *2018 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pp. 57–64, IEEE, 2018.

[51] D. Boneh, R. A. DeMillo, and R. J. Lipton, "On the importance of checking cryptographic protocols for faults," in *International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 37–51, Springer, 1997.

[52] I. Biehl, B. Meyer, and V. Müller, "Differential fault attacks on elliptic curve cryptosystems," in *Annual International Cryptology Conference*, pp. 131–146, Springer, 2000.

[53] J. Blömer and J.-P. Seifert, "Fault based cryptanalysis of the advanced encryption standard (aes)," in *Computer Aided Verification*, pp. 162–181, Springer, 2003.

[54] S. Carlier, "Electro magnetic fault injection," *University of Amsterdam, Amsterdam, Tech. Rep*, pp. 2011–2012, 2012.

[55] R. Omarouayache, J. Raoult, S. Jarrix, L. Chusseau, and P. Maurine, "Magnetic microprobe design for em fault attack," in *Electromagnetic Compatibility (EMC EUROPE), 2013 International Symposium on*, pp. 949–954, IEEE, 2013.

[56] M. A. Salam, *Electromagnetic field theories for engineering.* Springer Science & Business Media, 2014.

[57] A. Boyer, B. Vrignon, and J. Shepherd, "Near-field injection at die level," in *2015 Asia-Pacific Symposium on Electromagnetic Compatibility (APEMC)*, pp. 478–481, IEEE, 2015.

[58] M. Ghodrati, "Thwarting electromagnetic fault injection attack utilizing timing attack countermeasure," Master's thesis, Virginia Tech, 2018.

[59] A. Dehbaoui, J.-M. Dutertre, B. Robisson, and A. Tria, "Electromagnetic transient faults injection on a hardware and a software implementations of aes," in *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2012 Workshop on*, pp. 7–15, IEEE, 2012.

[60] J.-M. Schmidt and M. Hutter, "Optical and em fault-attacks on crt-based rsa: Concrete results," in *Proceedings of the 15th Austrian Workshop on Microelectronics*, Graz University of Technology, 2007.

[61] N. Moro, K. Heydemann, A. Dehbaoui, B. Robisson, and E. Encrenaz, "Experimental evaluation of two software countermeasures against fault attacks," in *2014 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pp. 112–117, IEEE, 2014.

[62] A. Barenghi, C. Hocquet, D. Bol, F.-X. Standaert, F. Regazzoni, and I. Koren, "Exploring the feasibility of low cost fault injection attacks on sub-threshold devices through an example of a 65nm aes implementation," in *International Workshop on Radio Frequency Identification: Security and Privacy Issues*, pp. 48–60, Springer, 2011.

[63] T. Korak and M. Hoefler, "On the effects of clock and power supply tampering on two microcontroller platforms," in *2014 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pp. 8–17, IEEE, 2014.

[64] J. G. Van Woudenberg, M. F. Witteman, and F. Menarini, "Practical optical fault injection on secure microcontrollers," in *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pp. 91–99, IEEE, 2011.

[65] M. Agoyan, J.-M. Dutertre, A.-P. Mirbaha, D. Naccache, A.-L. Ribotta, and A. Tria, "How to flip a bit?," in *2010 IEEE 16th International On-Line Testing Symposium*, pp. 235–239, IEEE, 2010.

[66] S. Skorobogatov, "Optical fault masking attacks," in *2010 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pp. 23–29, IEEE, 2010.

[67] J.-M. Schmidt, M. Hutter, and T. Plos, "Optical fault attacks on aes: A threat in violet," in *2009 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pp. 13–22, IEEE, 2009.

[68] C. Roscian, J.-M. Dutertre, and A. Tria, "Frontside laser fault injection on cryptosystems-application to the aes'last round," in *2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pp. 119–124, IEEE, 2013.

[69] P. Maurine, K. Tobich, T. Ordas, and P. Y. Liardet, "Yet another fault injection technique: by forward body biasing injection," in *YACC'2012: Yet Another Conference on Cryptography*, 2012.

[70] N. Beringuier-Boher, M. Lacruche, D. El-Baze, J.-M. Dutertre, J.-B. Rigaud, and P. Maurine, "Body biasing injection attacks in practice," in *Proceedings of the Third Workshop on Cryptography and Security in Computing Systems*, pp. 49–54, ACM, 2016.

[71] K. Tobich, P. Maurine, P.-Y. Liardet, M. Lisart, and T. Ordas, "Voltage spikes on the substrate to obtain timing faults," in *2013 Euromicro Conference on Digital System Design*, pp. 483–486, IEEE, 2013.

[72] A. Cui and R. Housley, "{BADFET}: Defeating modern secure boot using second-order pulsed electromagnetic fault injection," in *11th {USENIX} Workshop on Offensive Technologies ({WOOT} 17)*, 2017.

[73] F. Poucheret, K. Tobich, M. Lisarty, L. Chusseaux, B. Robissonx, and P. Maurine, "Local and direct em injection of power into cmos integrated circuits," in *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pp. 100–104, IEEE, 2011.

[74] A. Dehbaoui, J.-M. Dutertre, B. Robisson, P. Orsatelli, P. Maurine, and A. Tria, "Injection of transient faults using electromagnetic pulses-practical results on a cryptographic system," *IACR Cryptology EPrint Archive*, vol. 2012, p. 123, 2012.

[75] A. Menu, S. Bhasin, J.-M. Dutertre, J.-B. Rigaud, and J.-L. Danger, "Precise spatio-temporal electromagnetic fault injections on data transfers," in *2019 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pp. 1–8, IEEE, 2019.

[76] S. K. Bukasa, R. Lashermes, J.-L. Lanet, and A. Leqay, "Let's shock our iot's heart: Armv7-m under (fault) attacks," in *Proceedings of the 13th International Conference on Availability, Reliability and Security*, p. 33, ACM, 2018.

[77] Riscure, "Em-fi data sheet." https://getquote.riscure.com/picdb/filedb/3792/EM-FI%20datasheet.pdf, 2013. original document from Riscure.

[78] S. Bhasin and D. Mukhopadhyay, "Fault injection attacks attack methodologies, injection techniques and protection mechanisms." http://www-users.math.umn.edu/~math-sa-sara0050/space16/slides/space2016121504-debdeep.pdf, 2016.

[79] F. Wang and V. D. Agrawal, "Single event upset: An embedded tutorial," in *21st International Conference on VLSI Design (VLSID 2008)*, pp. 429–434, IEEE, 2008.

[80] M. Abramovici, M. A. Breuer, and A. D. Friedman, *Digital systems testing and testable design*, vol. 2. Computer science press New York, 1990.

[81] W. Stallings, *Cryptography and network security: principles and practices.* Pearson Education India, 2006.

[82] J. Takahashi, T. Fukunaga, and K. Yamakoshi, "Dfa mechanism on the aes key schedule," in *Fault Diagnosis and Tolerance in Cryptography, 2007. FDTC 2007. Workshop on*, pp. 62–74, IEEE, 2007.

[83] C. H. Kim, "Improved differential fault analysis on aes key schedule," *IEEE Transactions on Information Forensics and Security*, vol. 7, no. 1, pp. 41–50, 2012.

[84] N. A. Anagnostopoulos, "Optical fault injection attacks in smart card chips and an evaluation of countermeasures against them," Master's thesis, University of Twente, 2014.

[85] "Pci-x cryptographic coprocessor (fc 4764; ccin 4764)." https://www.ibm.com/support/knowledgecenter/en/POWER7/p7hcd/fc4764.htm. Accessed: 2017-07-09.

[86] S. Guilley, L. Sauvage, J.-L. Danger, and N. Selmane, "Fault injection resilience," in *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2010 Workshop on*, pp. 51–65, IEEE, 2010.

[87] Juliato, Marcio, *Fault Tolerant Cryptographic Primitives for Space Applications.* PhD thesis, University of Waterloo, 2011.

[88] T. Roche, V. Lomné, and K. Khalfallah, "Combined fault and side-channel attack on protected implementations of aes," in *International Conference on Smart Card Research and Advanced Applications*, pp. 65–83, Springer, 2011.

[89] D. Peacham and B. Thomas, "A dfa attack against the aes key schedule," *SiVenture White Paper*, vol. 1, p. 26, 2006.

[90] T. Korak, M. Hutter, B. Ege, and L. Batina, "Clock glitch attacks in the presence of heating," in *2014 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pp. 104–114, IEEE, 2014.

[91] N. Moro, K. Heydemann, E. Encrenaz, and B. Robisson, "Formal verification of a software countermeasure against instruction skip attacks," *Journal of Cryptographic Engineering*, vol. 4, no. 3, pp. 145–156, 2014.

[92] "nomad883 pro data sheet." https://media.digikey.com/pdf/Data%20Sheets/Sparkfun%20PDFs/TOL-14772_Web.pdf, 2017.

[93] M. T. Inc, "Pickit™ 3 in-circuit debugger/programmer user's guide." http://ww1.microchip.com/downloads/en/devicedoc/52116a.pdf, 2013. original document from Microchip Technology Inc.

[94] anysilicon, "Flipchip package overview." https://anysilicon.com/flipchip-package-overview/, 2017.

[95] D. Oswald, I. C. Paar, and D.-I. T. Kasper, "Development of an integrated environment for side channel analysis and fault injection," Master's thesis, Diploma thesis, Ruhr-University Bochum, 2009.

[96] G. Piret and J.-J. Quisquater, "A differential fault attack technique against spn structures, with application to the aes and khazad," in *International workshop on cryptographic hardware and embedded systems*, pp. 77–88, Springer, 2003.

[97] N. Semiconductors, "Lpc1110/11/12/13/14/15 product data sheet." https://www.nxp.com/docs/en/data-sheet/LPC111X.pdf, 2014. original document from NXP.

[98] "Arm®v6-m architecture reference manual." https://silver.arm.com/download/ARM_and_AMBA_Architecture/AR585-DA-70000-r0p0-02rel0/DDI0419E_armv6m_arm.pdf, 2017. original document from ARM.

[99] J. Yiu, "Why does ldr takes two cycles to be executed?." https://community.arm.com/developer/ip-products/processors/f/cortex-m-forum/13367/why-does-ldr-takes-two-cycle-to-be-executed/36323#36323, 2019.

[100] "Cortex™-m0 devices generic user guide." http://infocenter.arm.com/help/topic/com.arm.doc.dui0497a/DUI0497A_cortex_m0_r0p0_generic_ug.pdf, 2009. original document from ARM.

[101] "Getting started with mdk create applications with uvision® for arm® cortex®-m microcontrollers." http://www2.keil.com/docs/default-source/default-document-library/mdk5-getting-started.pdf?sfvrsn=2[NC,L], 2017. original document from ARM.

[102] "Amba® 3 ahb-lite protocol." https://silver.arm.com/download/download.tm?pv=1085658, 2010. original document from ARM.

[103] "Cortex™-m0 technical reference manual." http://infocenter.arm.com/help/topic/com.arm.doc.ddi0432c/DDI0432C_cortex_m0_r0p0_trm.pdf, 2012. original document from ARM.

[104] J. Yiu, *The Definitive Guide to ARM® Cortex®-M0 and Cortex-M0+ Processors*. Academic Press, 2015.

[105] P. Dodd and F. Sexton, "Critical charge concepts for cmos srams," *IEEE Transactions on Nuclear Science*, vol. 42, no. 6, pp. 1764–1771, 1995.

[106] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 361–372, 2014.

[107] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *arXiv preprint arXiv:1801.01207*, 2018.

[108] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, *et al.*, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 1–19, IEEE, 2019.

[109] "Arm holdings q3 2017 roadshow." https://cdn.group.softbank/en/corp/set/data/irinfo/presentations/investors/pdf/2017/investor_20180402_01.pdf, 2016.

[110] "Arm security technology." http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf, 2019. original document from ARM.

[111] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovi, "The risc-v instruction set manual. volume 1: User-level isa, version 2.0," tech. rep., CALIFORNIA UNIV BERKELEY DEPT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCES, 2014.

[112] M. Werner, R. Schilling, T. Unterluggauer, and S. Mangard, "Protecting risc-v processors against physical attacks," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1136–1141, IEEE, 2019.

[113] J. Laurent, V. Beroulle, C. Deleuze, and F. Pebay-Peyroula, "Fault injection on hidden registers in a risc-v rocket processor and software countermeasures," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 252–255, IEEE, 2019.

[114] M. A. Elmohr, H. Liao, and C. H. Gebotys, "Em fault injection on arm and risc-v," To appear in *21st International Symposium on Quality Electronic Design (ISQED)*. IEEE, 2020.

[115] K. Nørvåg, "An introduction to fault-tolerant systems," *Department of Computer and Information Science, Norwegian University of Science and Technology, Trondheim*, 2000.

[116] J.-M. Schmidt and C. Herbst, "A practical fault attack on square and multiply," in *2008 5th Workshop on Fault Diagnosis and Tolerance in Cryptography*, pp. 53–58, IEEE, 2008.

[117] P.-A. Fouque, F. Muller, G. Poupard, and F. Valette, "Defeating countermeasures based on randomized bsd representations," in *International Workshop on Cryptographic Hardware and Embedded Systems*, pp. 312–327, Springer, 2004.

[118] E. Biham and A. Shamir, "Differential fault analysis of secret key cryptosystems," *Advances in Cryptology—CRYPTO'97*, pp. 513–525, 1997.

[119] A. Vasselle, H. Thiebeauld, Q. Maouhoub, A. Morisset, and S. Ermeneux, "Laser-induced fault injection on smartphone bypassing the secure boot," *IEEE Transactions on Computers*, 2018.

[120] N. Miura, Z. Najm, W. He, S. Bhasin, X. T. Ngo, M. Nagata, and J.-L. Danger, "Pll to the rescue: a novel em fault countermeasure," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2016.

Appendix A

Hardware Countermeasures for EM FIA

There were some countermeasures particularly designed for EM FIA from the previous research. Most of the countermeasures were designed based on associated fault models. They are summarized in Table A.1. The objective of these countermeasures is to raise an alarm when the EM pulse is detected. They were all verified on FPGAs which were executing cryptographic algorithms such as AES [13, 15, 16], Simon cipher [120] or PRESENT cipher [14]. In Table A.1, there are two different detection rates. The first one is the EM injection detection rate. This detection rate describes the percentage of the tests where the alarm is raised successfully with an EM pulse induced to the circuit. The fault detection rate is the percentage of the test where the alarm is raised when the induced EM pulse actually causes a faulty output.

Table A.1: Summary of countermeasures designed for EM FIA

Countermeasure	Detection rate		Fault model
	EM injection	Fault	
Voltage glitch detector [13]	32%	91%	delay fault
Phase Lock Loop (PLL)-based [120]	N/A	79.28%	phase change in the clock signal
Hogge-based phase detector [14]	N/A	93.15%	phase change in the clock signal
Dual-complementary flip flop [15]	7%	100%	N/A
D-Flip flop detector [16]	N/A	92%	Sampling fault

In 2014, Zussa et al. [13] proposed a glitch detector as a countermeasure to EM FIA. The glitch detector shown in Figure A.1 could detect the sudden delay increase which is

caused by fault injection in the critical path. The *delay* denoted in the figure is the guarding delay which is greater than the critical path delay but smaller than the clock period. As suggested by the delay fault model, faults are injected by the coupling between the EM pulse and the power ground network which reduces the supply voltage and increases the delay. The timing diagram in Figure A.2 shows how the glitch detector raises the alarm. When the supply voltage is reduced, the guarding delay is increased which causes the alarm to be set at 1 indicating a fault injection attempt. The experimental result showed that one glitch detector in the chip was sufficient for detecting the fault injection attempt by power glitch or clock glitch. However, for EM FIA, even five glitch detectors were not efficient possibly due to the local property of the EM pulse. Additionally, by making a smaller probe tip, the attacker can have an enhanced spatial resolution. The highest EM injection detection rate was 32%. However, the fault detection rate was 91%.

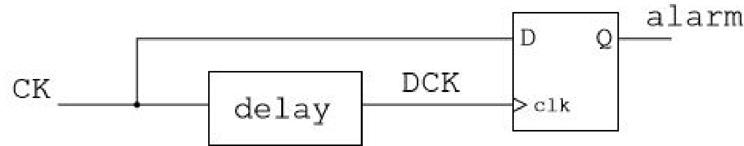
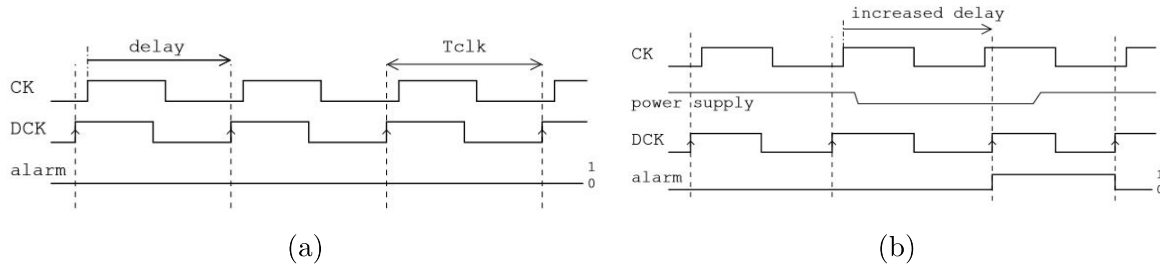


Figure A.1: Glitch detector [13]



(a) (b)

Figure A.2: (a): Normal operation when no extra delay is induced (b): Timing diagram when a glitch is detected and alarm is on [13]

Another countermeasure proposed for EM FIA utilized the PLL [120]. The author believed the EM pulse could cause phase shift in the clock network which could be further detected by the PLL. When an EM pulse is induced, it causes the PLL to enter an unlocked state. The EM injection detection rate was not revealed, however, the fault detection rate was 79.28% as presented in their second paper [14]. However, PLL is not always available in low-end chips, which limits the usage of this countermeasure. Breier et al. further improved

this countermeasure by replacing the PLL with a Hogge phase detector [14]. The structure of the Hogge phase detector is shown in Figure A.3. Since the $Data$ signal is synchronized to the CK signal, any phase change in the ring oscillator causes the same effect to the $Data$ signal. Therefore, a $DelayFactor$ is added to ensure the phase change in CK arrives earlier at the Hogge phase detector. The overall fault detection rate was 93.15%. However, this countermeasure only outputs an alarm when the EM pulse is induced before the delay factor shown in Figure A.3 [14]. If the EM pulse is injected directly to the Hogge phase detector, the circuit may fail to raise the alarm [14]. This behavior was also observed from their injection scan over the chip surface. There was a large area where no alarm was asserted. One solution might be put more on chip ring oscillators. However, this might increase the overall power consumption since the clock signal is always toggling in the ring oscillator.

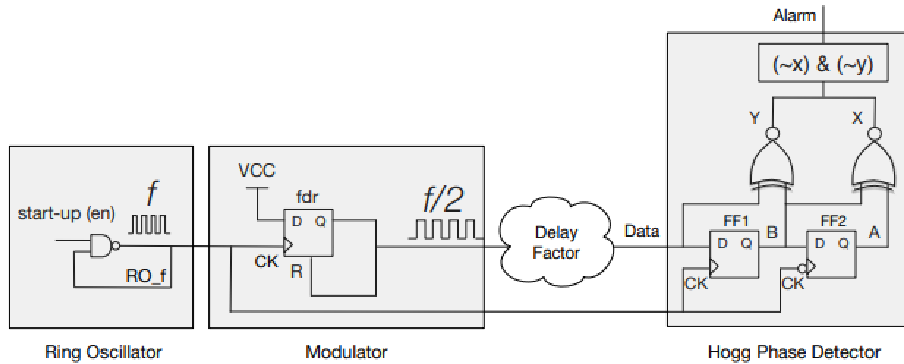


Figure A.3: Structure of the Hogge phase detector [14]

Deshpande et al. designed a dual complementary flip flop detector as a countermeasure to thwart EM FIA [15] as shown in Figure A.4. The alarm signal will arise if Qs equals Q. However, this countermeasure imposes a large overhead. In their final implementation on a Cyclone-IV FPGA, this countermeasure utilized 81.5% more dedicated registers. Compared with the other countermeasures [120, 14], it achieved a 100% percent fault detection rate even though only 7% of EM pulses created faults. However, there remains a possibility that the fault is injected to both flip flops, which will not be detected.

Another countermeasure proposed for EM FIA was based on the sampling fault model [16]. This countermeasure utilizes a detector composed of four D-type flip flops as shown in Figure A.5. These four flip flops are initialized with different values. Q1 and Q4 are initialized at 1 while Q2 and Q3 are initialized at '0'. Additionally, Q1 and Q3 switches at the rising edge of the clock while Q2 and Q4 switches at the falling edge of the clock.

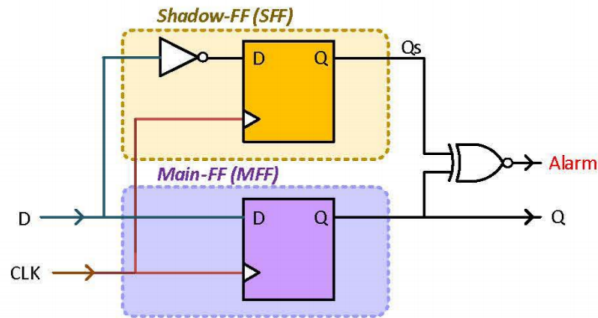


Figure A.4: Structure of the dual complementary flip flop [15]

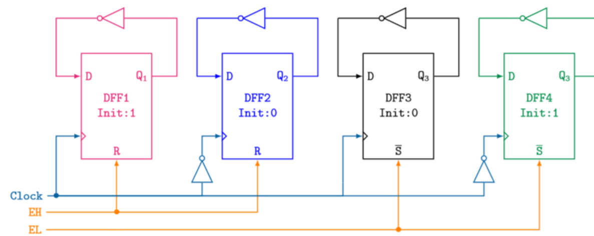


Figure A.5: The structure of the D-type flip flop countermeasure [16]

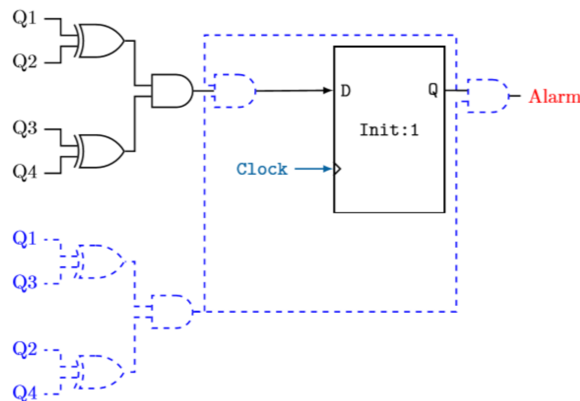


Figure A.6: The full schematic of the D-type flip flop countermeasure [16]

Based on the sampling fault model, the fault is induced at a triggering clock edge. Assume Q2 is faulted at a falling clock edge when it should be changed from '0' to '1'. This induced fault makes the $Q1 \oplus Q2$ stays at '0' when there is a rising edge of the clock. This '0' will further be sampled by the D flip flop and result in an alarm signal to detect the fault as shown in Figure A.6. The feedback from the output of the D flip flop back to the three-input AND gate will lock this '0'. The blue dashed circuit adds an extra protection for the combination where Q1, Q2, Q3, Q4 are '1', '0', '1', '0', respectively. This combination is also a faulty state. However, it does not lead to an alarm if this blue dashed circuit is not added. This detector circuit was regularly spread through an FPGA with an AES circuit. It detected 78% to 95% of the fault injected by the EM pulse, and 100% of the fault injected with the reverse body biasing injection.

In summary, countermeasures designed to particularly thwart EM FIA require a complete understanding of the injected fault. The countermeasures discussed in this appendix were all verified on FPGA but could not detect 100% of the faults injected by EM pulses. Further verification may be required for applying these countermeasures to other platforms such as ASIC, microcontroller, etc.

Appendix B

Additional Experiments on PIC16F687

B.1 Instruction/Macro description

This appendix lists the instructions and also the Macros which were not our target but used in test programs listed in Chapter 4.

Table B.1: Instruction description and the associated opcode

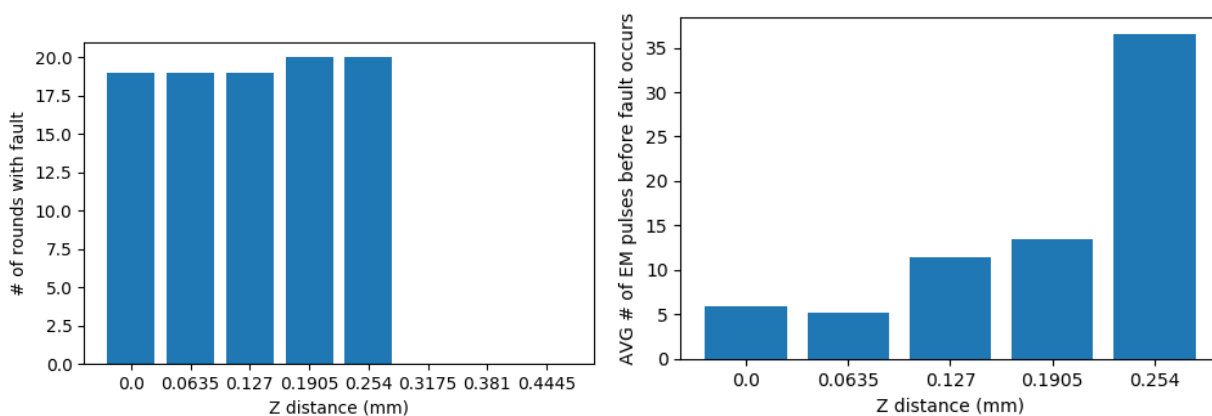
Instruction in PIC16F687	Description
BTFSS f, b	Bit test the b-th bit of register f, skip the next instruction if it is ‘1’
BTFSC f, b	Bit test the b-th bit of register f, skip the next instruction if it is ‘0’
MOVF f, d	Move the content in register f to a destination based on d
CALL k	Call a subroutine at address k
STATUS	The STATUS register in PIC16F687 which contains the arithmetic status, the reset status and the bank selection bits
RP0, RP1	The bank selection bit in the STATUS register
PORTC	PORTC register which contains eight readable and writable bits. The trigger signal is implemented on bit [5](RC5)

B.2 z-distance Effect at Best Coordinate

This appendix provides the experimental results for analyzing the z-distance effect. The probe was placed over the best coordinate as shown in Table 4.2. As mentioned in Section 4.3, the minimum z-distance was about 0.5 mm. We set this z-distance as the origin in z-axis. Then, we gradually increased the z-distance by 0.0635 mm per step. The V_p was set at 500 V to ensure the delivered EM pulse was as strong as possible. The fault injection experiment was conducted for 20 rounds for each z-distance as shown in Figure B.1. The number of pulses sent to the chip before the first fault was injected, was also recorded. Finally, we calculated the average number of pulses required to inject a fault at a specific location and the number of induced faults out of 20 rounds.

It requires more EM pulses to inject the fault when the z-distance is larger and finally when the z-distance is greater than 0.254 mm, no faults could be injected.

As mentioned in equation 2.4, the induced electromotive force drops as the cubed power of the z-distance. Therefore, it is expected that while the z-distance increases, it is more difficult to inject the fault.



(a) Number of faults injected over 20 round for each z-distance

(b) Average number of EM pulses before fault occurs

Figure B.1: Effect of the z-distance

Table B.2: Faulty data and the associated Flt_inst (The original value at 0x20 is 0x55 before executing the $XORWF$ instruction)

Tgt_inst	Faulty data at different addresses			Flt_inst
	0x20	0x21	0x30	
Correct data values	0x33	0x00	0x66	N/A
$XORWF\ 0x20,\ F$	0x55	0x77	0x66	$IORWF\ 0x20,\ W$
	0x55	0x66	0x66	NOP
	0x77	0x66	0x66	$IORWF\ 0x20,\ F$
	0x55	0x33	0x66	$XORWF\ 0x20,\ W$

B.3 Test Program for the $XORWF$ Instruction

Test program four was designed to analyze the faulty response of the $XORWF$ instruction. The correct data after executing the $XORWF\ 0x20,\ F$ instruction stored at address 0x20, 0x30, 0x21 are shown in second row of Table B.2.

```
1  MOVLW 0x20
2  MOVWF PORTC; Assert the trigger
3   3*NOP
4  MOVF 0x30, W
5  XORWF 0X20, F ; Target instruction
6  MOVWF 0x21 ; Move the content of W reg to register at 0x21;
7  MOVLW 0x00
8  MOVWF PORTC
```

Listing B.1: Test program four

The experiment was performed for three tests and each test consisted of 340 rounds with D_{t2p} set to 254 ns. Therefore, the D_{p22} of the P_{cyl} of the $XORWF$ instruction was 15.23 ns. There are four different combinations of faulty data found as shown in Table B.2. With the faulty data, the associated Flt_inst was found. For example, the Flt_inst was $IORWF\ 0x20,\ W$ when the faulty data was 0x55, 0x77, 0x66 at address 0x20, 0x21, 0x30 (row 4 of Table B.2), respectively. Hence, the data stored at the W register was changed to 0x77 after executing the faulty instruction. After executing the $MOVWF\ 0x21$ instruction at line 6 of listing B.1. This faulty data was stored at address 0x21 while the data stored at address 0x20 and 0x30 remained unchanged.

The largest hamming distance between the Tgt_inst and the Flt_inst is 3 where the Flt_inst is the NOP instruction as shown in Table B.3.

The occurrence of each Flt_inst over three tests is shown in Table B.4. FR denotes the total number of rounds with a fault injected and $FR_{N=1}$ denotes the number of rounds

Table B.3: Hamming distance between Tgt_inst and Flt_inst of test program four

Tgt_inst	opcode	Flt_inst	opcode	Hamming distance
$XORWF$ 0x20, F	0001101 0100000	$IORWF$ 0x20, W	0001000 0100000	2
		NOP	0000000 0100000	3
		$IORWF$ 0x20, F	0001001 0100000	1
		$XORWF$ 0x20, W	0001100 0100000	1

Table B.4: Statistical result of the fault injected on $XORWF$ 0x20, F in test program four

Test number	Flt_inst	Occurrence/FR	# of EM pulses before first fault				$FR_{N=1}$
			Max	Min	Avg	Std	
1	$IORWF$ 0x20, W	40.3%/137	23	1	4.64	3.41	23
	NOP	57.9%/197	11	1	4.70	2.70	31
	$IORWF$ 0x20, F	1.76%/6	9	1	4.33	3.27	2
2	$IORWF$ 0x20, W	43.8%/149	15	1	4.82	2.87	18
	NOP	53.2%/181	9	1	4.53	2.69	36
	$IORWF$ 0x20, F	2.94%/10	8	3	5.40	2.07	0
3	$IORWF$ 0x20, W	38.8%/132	18	1	4.61	3.03	25
	NOP	58.2%198	10	1	4.69	2.64	38
	$IORWF$ 0x20, F	2.65%/9	6	1	3.44	1.81	1
	$XORWF$ 0x20, W	0.29%/1	1	1	N/A	N/A	1

where a single EM pulse could successfully inject a fault. The Table also presents the number of EM pulses required to induce the fault. Faults could be injected with a small number of EM pulses. For all the four different Flt_insts, the average number of EM pulses required to inject the fault was around 5. Additionally, the total number of rounds where a fault could be induced with a single EM pulse was 175. Therefore, around 17% of the rounds, a fault could be induced with the first EM pulse. The induced fault was reproducible with a similar occurrence for each type of the fault and a small standard deviation. The average occurrence and the standard deviation for each occurrence are shown in Table B.5.

Table B.5: The average and standard deviation of the occurrence for each Flt_inst

Flt_inst	Avg	Std
$IORWF$ 0x20, W	41.0%	2.56%
NOP	56.4%	2.80%
$IORWF$ 0x20, F	2.45%	0.615%
$XORWF$ 0x20, W	0.097%	0.167%

B.4 Test Program for the *DECFSZ* Instruction

A test program was designed to find possible Flt_insts for the $DECFSZ$ instruction. Additionally, the probe tip was moved back to the location as indicated in Table 4.2. The test program is shown in listing B.2.

```
1  NORMAL:
2   MOVLW 0x20
3   MOVWF PORTC
4    4*NOPs
5   DECFSZ 0x20, F ; Target instruction
6   GOTO FAULT_LOOP
7    ...
8   GOTO NORMAL
9  FAULT_LOOP:
10  Write to EEPROM.
```

Listing B.2: Test program five

Initially, the data stored at address 0x20 is 0x01. Therefore, if the $DECFSZ$ $0x20,\ F$ is correctly executed, the program should skip the next $GOTO\ FAULT_LOOP$ instruction. The fault injection experiment was run for three tests with 340 rounds per test. The D_{t2p} was set to 254 ns. Hence, the D_{p22} of the P_{cyl} of the $DECFSZ$ was 15.23 ns. Three sets of faulty data were found at address 0x20 and the associated Flt_insts are shown in table B.6. The hamming distance between the Tgt_inst and the Flt_inst is shown in Table B.7. The largest hamming distance is 4 when the Flt_inst was the NOP instruction.

Table B.6: Faulty data and the associated Flt_inst for test program five

Tgt_inst	Faulty data at address 0x20	Flt_inst
$DECFSZ$ $0x20,\ F$	0x01	NOP
	0x20	$MOVWF$
	0x02	$INCF$ $0x20,\ F$

Table B.7: Hamming distance between Tgt_inst and Flt_inst of test program five

Tgt_inst	Opcode	Flt_inst	Opcode	Hamming distance
$DECFSZ$ $0x20,\ F$	0010111 0100000	NOP	0000000 0100000	4
		$MOVWF$ $0x20$	0000001 0100000	3
		$INCF$ $0x20,\ F$	0010101 0100000	1

Table B.8: Statistical result of the fault injected on $DECFSZ$ instruction in test program five

Test number	Flt_inst	Occurrence/FR	# of EM pulses before first fault				$FR_{N=1}$
			Max	Min	Avg	Std	
1	NOP	60.0%/204	35	1	4.41	3.42	29
	$MOVWF$ 0x20	25%/85	22	1	5.16	3.46	8
	$INCF$ 0x20, F	15%/51	41	1	8.55	9.16	6
2	NOP	53.5%/182	45	1	4.55	4.30	26
	$MOVWF$ 0x20	25.6%/87	18	1	5.09	3.09	10
	$INCF$ 0x20, F	20.9%/71	39	1	7.92	7.28	9
3	NOP	50.9%/173	27	1	4.38	3.65	26
	$MOVWF$ 0x20	29.7%/101	15	1	4.95	3.53	20
	$INCF$ 0x20, F	19.4%/66	40	1	7.95	8.69	6

The statistical result is presented in Table B.8. The percentage of the experiment where the first EM pulse injected the fault was around 13.7%. The average number of EM pulses before injecting the first fault with the Flt_inst as NOP was around 4.4. The average and the standard deviation of the occurrence of different Flt_insts during three tests are shown in Table B.9

Table B.9: The average and standard deviation of the occurrence for each Flt_inst of $DECFSZ$ in test program five

Flt_inst	Avg	Std
NOP	54.8%	4.69%
$MOVWF$ 0x20	26.8%	2.56%
$INCF$ 0x20, F	18.4%	3.07%

After the experiment on test program five, at this specific location, three possible Flt_insts were found and the NOP instruction was also one of the Flt_insts. Therefore, it is possible to perform a bypass security check attack (see Section 4.6.1) and a round addition attack on AES-128 (see Section 4.6.2).

B.5 Combined Countermeasure for both the $XORWF$ and $DECFSZ$

The code shown in listing B.3 explains the design of the countermeasure. The instructions added for this countermeasure are underlined. This countermeasure is a combination of the

countermeasure for the $XORWF$ and the $DECFSZ$ proposed in Section 4.8.3 and 4.8.4, respectively. Two trigger signals are generated so that EM pulses could be delivered over the $XORWF$ instruction and the $DECFSZ$ instruction during the last round of AES-128. A delay factor (approximate 100 us) is inserted in line 25 to provide enough delay so that both EM pulses could be generated consecutively. In this way, the verification of the countermeasure is more robust.

```
1  encrypt:
2    MOVLW 0x01
3    MOVWF rcon         ; RCON initialization
4    CALL    store_var ; Store the intermediate state to protect the XORWF
5    CALL  key_addition    ; initial key addition
6    MOVLW 0x0A    ; start the round counter for the loop
7    MOVWF round_counter   ;
8    MOVLW    0x0B
9    MOVWF    round_counter_sec    ; Store 0x0B to the second round counter
10 loop_encrypt:
11      DECF    round_counter_sec , F
12   BTFSC  STATUS, Z
13   GOTO    BACK_RET
14   CALL  substitution_S
15   CALL  enc_shift_row
16   CALL    store_var   ; Store the intermediate state to protect the XORWF
       instruction
17   DECF  round_counter ,W
18   BTFSC STATUS,Z
19   GOTO  last_round
20   CALL  mix_column
21   CALL    store_var
22 last_round:
23   CALL  enc_key_schedule
24   CALL  key_addition    ; key addition
25   CALL    Delay; A small delay added to make sure both the \gls{em} pulse
       could be delivered at both triggers
26   MOVLW    0x01
27   SUBWF    round_counter , w
28   BTFSS    STATUS, Z
29   GOTO     no_trigger
30   BCF      STATUS, RP0
31   BCF      STATUS, RP1
32   MOVLW    b'00100000'
33   MOVWF    PORTC
34 no_trigger:
35     4*NOPs
36   DECFSZ  round_counter ,F       ; Target instruction
```

```
37    GOTO    loop_encrypt
38 BACK_RET:
39    RETURN
```

Listing B.3: Countermeasure for protecting the AES from attacks proposed in Section 4.5.1 and 4.6.2

Appendix C

Unsuccessful Experiment Details Targeting the $LDR <Rt>, [<Rn>, \# <imme>]$ and $LDMIA <Rn>!, <registers>$ Instructions on Chip One of LPC1114

In this appendix, we will provide the experimental results which targeted the $LDR <Rt>, [<Rn>, \# <imme>]$ and $LDMIA <Rn>!, <registers>$ with chip #1 and probe tip one. The function fault was failed to be injected with chip one and probe tip one. These results could not further be verified on the second chip, nor analyzed in detail due to the damage of the probe tip one. The experiments in this appendix were also done with $F_{clk} = 104.4Mhz$ and $V_{DD} = 3.3V$.

To analyze if a fault could be injected with different kinds of LDR instructions, we designed several different test programs. The program shown in listing C.1 utilized the $LDMIA$ instruction as the target instead of the $LDR\ PC - relative$ instruction. We first stored the same value at address 0x10001000, 0x10001004 and 0x10001008 as shown in lines 2 to 6. The simulation of the result is shown in Table C.1. Instead of loading a single word from one single memory location, the $LDMIA$ instruction loads multiple words from consecutive memory locations. Thus, in our program, we checked register R1, R3 and R4 to see if a fault occurred. However, even though we scanned through all the delay settings from 274 ns to 404 ns with 10 ns per step, no faulty value was sent out by

the UART.

```
_main
  LDR R0, =(0x10001000);
  LDR R1, =(0xFEDCBA98);
  LDR R3, =(0xFEDCBA98);
  LDR R4, =(0xFEDCBA98);
  STMIA R0!, {R1,R3, R4}; Store 0xFEDCBA98 to address at 0x10001000 to 0
    x10001008
  LDR R0, =(0x10001000);
  Assert trigger;
    26*NOPs;

  LDR R2, =(0x00000000); To make the code similar to the LDR PC-relative
    program where fault could be injected successfully;
  LDMIA R0!, {R1,R3, R4};
  NOP;
  NOP;
  NOP;
  NOP;
  CMP R1, R3;
  BNE FAULT;
  CMP R3,R4;
  BNE FAULT;
  De-assert trigger;
  B _main;
```

Listing C.1: Test program two targeting the LDMIA instruction

Table C.1: Simulation result for test code where a $LDMIA$ instruction is the target

Address (in hexadecimal)	Encoding(in hexadecimal)	Instruction
0x00000A42	0x4C0C	$LDR\ R2, [PC,\ \#0x44]$
0x00000A44	0xC81A	$LDM\ R0!,\ [R1,\ R3-R4]$
0x00000A46	0xBF00	NOP
......	......	Other instructions

For test program 1-c to 1-f, the fault was induced since the offset was added to the PC or the set bits in the offset were reset. Another experiment was designed with test code three as shown in listing C.2. The $LDR\ R2,\ [R0,\ \#0x4]$ and the $LDR\ R3,\ [R1,\ \#0x8]$ instructions use the immediate offset addressing where the target address is calculated by using a base register plus the offset value. We want to check if the EM pulse could cause a fault where the offset is not added to the base register. To make the test program successfully detect the fault when the offset was not added, we wrote the same value to

```
0x10001000: 98 BA DC FE
0x10001004: 5A 5A 5A 5A
0x10001008: 00 00 00 00
0x1000100C: 00 00 00 00
0x10001010: 5A 5A 5A 5A
0x10001014: 00 00 00 00
```

Figure C.1: Data value stored in the memory when targeting *LDR* immediate offset instruction

the memory at address 0x10001004 and 0x10001010 (both words are in the SRAM region of LPC1114) and also initialized the memory at address 0x10001000 as 0xFEDCBA98.

```
_main
  Store values to associate memory address;
  LDR R0, =(0x10001000);
  LDR R1, =(0x10001008);
  Assert trigger;
    26*NOPs;

  LDR R2, [R0, #0x4];
  LDR R3, [R1, #0x8];
  NOP;
  NOP;
  NOP;
  NOP;
  CMP R1, R3;
  BNE FAULT;
  CMP R3,R4;
  BNE FAULT;
  De-assert trigger;
  B _main;
```

Listing C.2: Test program three targeting the LDR immediate offset instruction

The memory map is shown in Figure C.1 and the simulation result is shown in Table C.2.

We stored 0x10001000 at register R0 and 0x10001008 in register R1. After executing the two instructions ($LDR\ R2,\ [R0,\ \#0x4]$ and $LDR\ R3,\ [R1,\ \#0x8]$), R2 should have the same value as R3 if no fault is injected.

However, the experimental result indicated that no faulty data was output from the UART even though we scanned through all the possible D_{t2p} from 274 ns to 404 ns with 10 ns per step.

Table C.2: Simulation result for test code which uses a register + immediate offset as the destination address

Address (in hexadecimal)	Encoding(in hexadecimal)	Instruction
0x00000A16	0x6842	$LDR\ R2,\ [R0,\ \#0x4]$
0x00000A18	0x688B	$LDR\ R3,\ [R1,\ \#0x8]$
0x00000A1A	0xBF00	NOP
......	......	Other instructions