Side-Channel Security Analysis and Protection of SHA-3

A Dissertation Presented
by

Pei Luo

to

The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy

in

Computer Engineering

Northeastern University
Boston, Massachusetts

August 2017
To my family, and my friends. 献给我的家人和朋友，谢谢你们!
## Contents

**List of Figures**  
vi

**List of Tables**  
ix

**List of Acronyms**  
x

**Acknowledgments**  
xi

**Abstract of the Dissertation**  
xii

1 Introduction  
1.1 Motivation  
1.2 Research Agenda  

2 Preliminaries of SHA-3  

3 Side-Channel Power Analysis of SHA-3  
3.1 Introduction and Motivation  
3.1.1 Keccak Parameters Used in This Chapter  
3.2 Attack on the first round output  
3.2.1 Side-Channel Leakage of $R_1$  
3.2.2 Side-Channel Analysis on $R_1$  
3.2.3 Attacks on $R_1$ for Different Key Length  
3.3 Attacks Based on the Leakages of $\theta$ Operation in the First Round  
3.3.1 Side-Channel Leakages of $\theta$ Operations  
3.3.2 Side-Channel Analysis on $\theta_o$  
3.3.3 Side-Channel Analysis on $\theta_{plane}$  
3.3.4 Key Length  
3.3.4.1 How Key Length Affects the Attacks on $\theta_o$  
3.3.4.2 Key Length Affects the Attacks on $\theta_{plane}$  
3.4 Further discussion about the key-length and countermeasures  
3.4.1 Side-Channel Attacks on MAC-Keccak with State Register Initialized to $P$  
3.4.2 How to Detect the Key Length  
3.4.3 Simple Methods to Decrease Side-Channel Leakages  


3.4.4 Side-Channel Security of Keccak

3.5 Summary

4 Protect SHA-3 Against Side-Channel Power Analysis

4.1 Introduction and Motivation

4.2 Side-Channel Leakages Hidding Using Shuffling

4.2.1 Introduction and Motivation

4.2.2 Preliminaries

4.2.2.1 Countermeasure to Power Analysis Attacks - Shuffling

4.2.2.2 Data Dependence and Operation Consistency

4.2.3 Methodology and Algorithm

4.2.3.1 Statement Dependence Extraction

4.2.3.2 Automatic Shuffling

4.2.4 Experimental Results and Evaluation

4.2.4.1 The Tool for Source Code Transformation

4.2.4.2 Side-Channel Leakage Reduction

4.2.4.3 Overhead Evaluation

4.3 An Improvement of Both Security and Reliability of Keccak Software Implementations

4.3.1 Introduction and Motivation

4.3.2 Round Rotation Invariance Based Scheme

4.3.2.1 Invariance of Keccak Permutation Function

4.3.2.2 Invariance-Based Protection Scheme

4.3.2.3 Advantages of the Proposed Scheme

4.3.3 Side-Channel Power Analysis and Fault Injection Simulation of the Proposed Scheme

4.3.3.1 Side-Channel Analysis Results of Protected and Unprotected Implementations

4.3.3.2 Fault Injection Simulation Result

4.4 Summary

5 Compiler Assisted Threshold Implementation Design

5.1 Introduction and Motivation

5.2 Preliminaries of Threshold Implementation and LLVM

5.2.1 Notations

5.2.2 Threshold Implementation

5.2.3 LLVM

5.3 CATI - Compiler Assisted Threshold Implementation Pass

5.3.1 Automated Semi-TI Solutions

5.3.2 Achieving Uniformity with Limited Randomness

5.3.3 Minimum Modification of Given Implementation

5.3.3.1 Sensitive Variable-Free Branches

5.3.3.2 Function Interface

5.3.4 Modular Design

5.4 Result

5.4.1 Resource Overhead of the Generated TI Implementation
5.4.2 Side-Channel Attacks Results .................................................. 67
5.5 Discussion and Future Work ..................................................... 70
5.6 Summary .............................................................................. 70

6 Differential Fault Analysis of SHA-3 ........................................... 73
6.1 Introduction and Motivation ................................................... 73
6.2 Fault Model and Notations .................................................... 74
6.2.1 The Fault Model Used ....................................................... 75
6.3 Fault Propagation and Fault Signature .................................... 76
6.3.1 Observable Hash Digest ...................................................... 76
6.3.2 Fault Signature Generation ............................................... 77
6.3.2.1 Fault Propagation in $\chi_{22}$ - $FP_{\chi}$ .............................. 78
6.3.2.2 Fault Propagation in $\theta_{23}$ - $FP_{\theta}$ ............................. 80
6.4 Differential Fault Analysis of SHA3-384 and SHA3-512 .......... 82
6.4.1 Fault Position $P$ and Value $F$ Recovery .......................... 82
6.4.2 $\chi_{22}$ Bits Recovery ........................................................ 84
6.5 Differential Fault Analysis of SHA3-224 and SHA3-256 ........... 86
6.5.1 Basic Attacks of SHA3-224 and SHA3-256 ......................... 86
6.5.1.1 $\chi_{23}^i$ Bits Recovery from the Observable Digest .............. 87
6.5.1.2 Injected Fault Identification and $\chi_{22}^i$ Bits Recovery .......... 88
6.5.2 Improve the Attacks by Using both $FS_{\chi_{23}}$ and $FS_{\chi_{22}}$ ....... 89
6.5.2.1 Fault Signature $FS_{\chi_{23}}$ Construction ............................ 90
6.5.2.2 Simulation Results ....................................................... 90
6.5.3 Further Improve the Attacks by Injecting Faults at $\theta_{23}$ ...... 91
6.5.3.1 Recovering More $\chi_{23}^i$ by Injecting Faults at $\theta_{23}$ .......... 92
6.5.3.2 Simulation Results ....................................................... 93
6.6 Optimized Attacks with Chosen Faults .................................... 95
6.7 Discussions ...................................................................... 97
6.7.1 SHA-3 Systems with Long Input Message ......................... 97
6.7.2 Countermeasure against Differential Fault Analysis ............ 98
6.8 Summary ...................................................................... 98

7 Algebraic Fault Analysis of SHA-3 .............................................. 99
7.1 Introduction and Motivation ................................................ 99
7.2 Fault Models .................................................................. 100
7.3 Algebraic Fault Analysis of SHA-3 ....................................... 100
7.3.1 Overview of Algebraic Fault Analysis on SHA-3 ................. 100
7.3.2 SAT Problem Formulation ............................................... 101
7.3.2.1 Construction of the Equation Set for Keccak .................. 101
7.3.2.2 Constraints for the Injected Fault ................................. 103
7.3.3 Recovery of Internal State Bits ........................................ 103
7.4 Improvement - Utilize Multiple Faults Together for Tighter Constraints .................................................. 107
7.4.1 Tighter Constraints Construction ...................................... 107
7.4.2 Attack Results .............................................................. 107
7.5 Further Improvement - Multiple Fault Injection Points ......... 110
7.5.1 Recovery of $\chi^2_{23}$ .......................................................... 110
7.5.2 Attack Results ................................................................. 112
7.6 Discussion ................................................................. 113
   7.6.1 Result Discussion .................................................. 113
   7.6.2 Countermeasure and Future Work ......................... 115
7.7 Summary ................................................................. 115

8 Protect SHA-3 Against Fault Attacks 117
   8.1 Introduction and Motivation ........................................ 117
   8.2 Protect SHA-3 Using Error Detection Codes ................ 118
      8.2.1 Introduction and Motivation ................................ 118
      8.2.2 Preliminary of Error Detection ............................ 118
      8.2.3 Parity Checking of Keccak .................................. 119
         8.2.3.1 Analysis of the Protection of Keccak .............. 119
         8.2.3.2 Parity Checking of $\theta$ ......................... 121
         8.2.3.3 Parity Checking of $\rho$ and $\pi$ .......... 123
         8.2.3.4 Parity Checking of $\chi$ and $\iota$ ......... 123
      8.2.4 Implementation and Fault Injection Results ............ 125
         8.2.4.1 Implementation Results ............................ 125
         8.2.4.2 Fault Coverage Analysis ............................ 126
   8.3 Protect SHA-3 by Detecting System Disturbance ............ 127
      8.3.1 Introduction .................................................... 127
      8.3.2 Fault Injection Using Clock Glitch .................... 128
         8.3.2.1 Related Works ...................................... 128
      8.3.3 Clock Glitch and Power Disturbance Detection ....... 129
         8.3.3.1 System Clock Glitch Checking .................. 129
         8.3.3.2 Relaxation of the Design for Robustness .... 131
         8.3.3.3 Upper bound of the Proposed Scheme .......... 132
      8.3.4 Glitch Detection and Implementation Results .......... 132
         8.3.4.1 Internal Clock Signal Generation .............. 133
         8.3.4.2 Clock Glitch Detection Result .................. 133
         8.3.4.3 Overhead Result of the Proposed Scheme ...... 135
   8.4 Summary ................................................................. 136

9 Conclusion .......................................................................... 137

Bibliography .......................................................................... 139

Publication List ..................................................................... 152
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Terminology used in Keccak</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>The sponge construction</td>
<td>6</td>
</tr>
<tr>
<td>2.3</td>
<td>Keccak $\theta$ operation</td>
<td>7</td>
</tr>
<tr>
<td>2.4</td>
<td>Keccak $\rho$ operation</td>
<td>7</td>
</tr>
<tr>
<td>2.5</td>
<td>Keccak $\pi$ operation</td>
<td>8</td>
</tr>
<tr>
<td>2.6</td>
<td>Keccak $\chi$ operation</td>
<td>8</td>
</tr>
<tr>
<td>3.1</td>
<td>Power trace example and the leakage of $R_1$</td>
<td>14</td>
</tr>
<tr>
<td>3.2</td>
<td>Leakage of one row of $R_1$ vs. leakage of one bit of $R_1$</td>
<td>16</td>
</tr>
<tr>
<td>3.3</td>
<td>Key guess result of $R_1([0:4,4,0])$ using 40,000 traces</td>
<td>17</td>
</tr>
<tr>
<td>3.4</td>
<td>Correlation and success rate results of attacking $R_1([0:4,4,0])$ for key length 320 bits</td>
<td>18</td>
</tr>
<tr>
<td>3.5</td>
<td>Correlation and success rate results of attacking $R_1([0:4],4,0)$ for key length 640 bits</td>
<td>20</td>
</tr>
<tr>
<td>3.6</td>
<td>Correlation relationship of variables in Keccak.</td>
<td>21</td>
</tr>
<tr>
<td>3.7</td>
<td>Leaks of $\theta_o$ and $\theta_{plane}$</td>
<td>23</td>
</tr>
<tr>
<td>3.8</td>
<td>Correlation and success rate results of attacking $\theta_o([1:4],[0:7])$</td>
<td>25</td>
</tr>
<tr>
<td>3.9</td>
<td>Correlation and success rate results of attacking $\theta_{plane}(4,0,[0:7])$</td>
<td>26</td>
</tr>
<tr>
<td>3.10</td>
<td>The relationship among $n_b$, $l_o$ and $l_m$.</td>
<td>30</td>
</tr>
<tr>
<td>4.1</td>
<td>The dependence of the first 36 statements of Keccak</td>
<td>38</td>
</tr>
<tr>
<td>4.2</td>
<td>Automatic source code transformation with shuffling</td>
<td>40</td>
</tr>
<tr>
<td>4.3</td>
<td>Power leakage of the unprotected and shuffled Keccak on Microblaze</td>
<td>42</td>
</tr>
<tr>
<td>4.4</td>
<td>The proposed invariance-based protection scheme</td>
<td>46</td>
</tr>
<tr>
<td>4.5</td>
<td>Side-channel power analysis results: the left side are the results of attacking the original implementation, the right side are the results of attacking the implementation protected with the proposed scheme.</td>
<td>49</td>
</tr>
<tr>
<td>5.1</td>
<td>The structure of candidate tree</td>
<td>58</td>
</tr>
<tr>
<td>5.2</td>
<td>Example TI solution for the AND operation</td>
<td>60</td>
</tr>
<tr>
<td>5.3</td>
<td>Divide-and-conquer strategy</td>
<td>63</td>
</tr>
<tr>
<td>5.4</td>
<td>IR representation of the CATI process</td>
<td>65</td>
</tr>
<tr>
<td>5.5</td>
<td>Correlation Power Analysis of the original AES implementation</td>
<td>67</td>
</tr>
<tr>
<td>5.6</td>
<td>Correlation Power Analysis of the original AES implementation</td>
<td>68</td>
</tr>
<tr>
<td>5.7</td>
<td>T-test results for the original and generated TI implementation of AES S-box</td>
<td>71</td>
</tr>
<tr>
<td>5.8</td>
<td>T-test results for the original and generated TI implementation of Keccak</td>
<td>72</td>
</tr>
</tbody>
</table>
6.1 Notations for operations and intermediate states 75
6.2 Fault signature at $\chi_{22}^i$ for the example single-bit fault injected at $\theta_{22}^i$ 78
6.3 Fault signature at the output of $\chi_{22}^i$ 81
6.4 Fault signature at $\chi_{23}^i$ (Bottom plane) 82
6.5 Distribution of the number of recovered $\chi_{22}^i$ bits for single-byte faults 85
6.6 Recover process 86
6.7 Number of recovered $\chi_{22}^i$ bits for different number of effective faults 89
6.8 Number of recovered $\chi_{22}^i$ bits for different number of effective faults 91
6.9 Recovery of $\chi_{22}^i$ bits in the bottom plane by injecting faults at $\theta_{23}^i$ 92
6.10 Effective fault ratio with different number of $\chi_{23}^i$ rows recovered 93
6.11 Number of recovered $\chi_{22}^i$ bits for different number of effective faults with a number of $\chi_{23}^i$ rows recovered, SHA3-224 94
6.12 Optimized fault injection attacks at byte level 96

7.1 Two numerical solutions 106
7.2 Distribution of the number of recovered $\chi_{22}^i$ bits for SHA-3 functions using Method II 109
7.3 Attack results under Method II 109
7.4 Recovery of $\chi_{23}^i$ 112
7.5 The distribution of number of $\chi_{22}^i$ bits recovered with part of $\chi_{23}^i$ bits recovered 113
7.6 The attack results of the proposed attack Method III 114

8.1 Conception of error detection 119
8.2 Error detection structure of Keccak 120
8.3 Fault injection using clock glitch 128
8.4 Clock width count results 130
8.5 Clock width count results when glitch injected 130
8.6 Clock width monitoring system 131
8.7 Upper bound of the detection 132
8.8 Clock glitch detection rate at different glitch frequency ($\lambda = 2$) 134
8.9 Clock glitch detection rate with two ROs ($\lambda = 2$) 135
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Comparison of resources and execution cycles among three schemes</td>
<td>43</td>
</tr>
<tr>
<td>4.2</td>
<td>Comparison of countermeasures</td>
<td>52</td>
</tr>
<tr>
<td>5.1</td>
<td>LLVM IR variables overhead of the proposed scheme with RNG off</td>
<td>66</td>
</tr>
<tr>
<td>6.1</td>
<td>Fault propagation of operation $\chi^{22}$</td>
<td>80</td>
</tr>
<tr>
<td>6.2</td>
<td>Simulation results for SHA3-224 and SHA3-256 with fault injected at $\theta^{22}$</td>
<td>89</td>
</tr>
<tr>
<td>7.1</td>
<td>Variables for SAT equations of SHA3-512</td>
<td>103</td>
</tr>
<tr>
<td>7.2</td>
<td>Compare Attach Method I with DFA</td>
<td>105</td>
</tr>
<tr>
<td>7.3</td>
<td>Number of recovered $\chi^{23}$ bits</td>
<td>111</td>
</tr>
<tr>
<td>7.4</td>
<td>Attack results of proposed attack methods</td>
<td>115</td>
</tr>
<tr>
<td>8.1</td>
<td>XOR gates overhead of $\theta$ protection</td>
<td>122</td>
</tr>
<tr>
<td>8.2</td>
<td>Resource overhead and error coverage results</td>
<td>126</td>
</tr>
<tr>
<td>8.3</td>
<td>FPGA resource consumption comparison</td>
<td>135</td>
</tr>
</tbody>
</table>
List of Acronyms

**SHA-3** Secure Hash Algorithm 3. A subset of the cryptographic primitive family Keccak, it is a cryptographic hash function designed by Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche, building upon RadioGatún.

**MAC** Message Authentication Code. A message authentication code (MAC) is a short piece of information used to authenticate a message - in other words, to confirm that the message came from the stated sender (its authenticity) and has not been changed in transit (its integrity).

**SCA** Side-Channel Attack. Side-channel attack is any attack based on information gained from the physical implementation of a cryptosystem, rather than brute force or theoretical weaknesses in the algorithms (compare cryptanalysis).

**DFA** Differential Fault Analysis. Differential fault analysis is a type of side channel attack in the field of cryptography, specifically cryptanalysis. The principle is to induce faults—unexpected environmental conditions—into cryptographic implementations, to reveal their internal states.

**AFA** Algebraic Fault Analysis. It is a type of side channel attack which combines algebraic cryptanalysis with fault attacks.

**SAT** Boolean Satisfiability Problem. Boolean Satisfiability Problem is the problem of determining if there exists an interpretation that satisfies a given Boolean formula.

**SMT** Satisfiability Modulo Theories. In computer science and mathematical logic, SMT problem is a decision problem for logical formulas with respect to combinations of background theories expressed in classical first-order logic with equality.

**TI** Threshold Implementation. TI is special kind of masking countermeasure, it splits the original variables into multiple shares and thus the attacker cannot conquer the target system unless he has control of all the shares. TI is provable secure if all the requirements are followed.

**FPGA** Field-Programmable Gate Array. An integrated circuit designed to be configured by a customer or a designer after manufacturing - hence “field-programmable”.
Acknowledgments

Thanks to my family and my friends.
感谢每一个陪伴过，以及陪伴着我的人！
Abstract of the Dissertation

Side-Channel Security Analysis and Protection of SHA-3

by

Pei Luo

Doctor of Philosophy in Computer Engineering
Northeastern University, August 2017
Dr. Yunsi Fei, Advisor

Keccak is the hash function selected by NIST as the SHA-3 (Secure Hash Algorithm) standard. Keccak is built on a new structure - sponge construction. The new algorithm and construction have raised questions regarding the side-channel security of SHA-3 and its MAC mode, MAC-Keccak.

Over the past decade, side-channel attack has shown to be a real and effective threat to many crypto systems, where inadvertent side-channel leakages can be exploited to retrieve secret information of the target system. Side-channel power (EM) analysis and fault injection attacks have been widely used to break various crypto algorithms, while the side-channel vulnerability and resilience of SHA-3 systems have not been addressed much. This dissertation investigates both side-channel security analysis and protection of SHA-3 systems. I propose novel power analysis and fault analysis methods, and also effective countermeasures against these attacks.

For side-channel power analysis, I examine hardware implementations and propose effective attack methods using power leakages from the first round output and the first round \( \theta \) operation. To protect SHA-3 systems against side-channel power analysis, I adopt the operation shuffling method as the countermeasure. I propose algorithms to identify and explore the shuffling space automatically and then add shuffling into SHA-3 implementations. Recently Threshold Implementation (TI) has been a prevalent countermeasure against power analysis attacks with provable security. I adopt the method and implement it within the compilation process to automatically generate secure SHA-3 code.

For fault injection attacks on SHA-3, we implement differential fault analysis attacks targeting all four SHA-3 modes under much more relaxed fault models. I then introduce algebraic techniques into the attacks, to significantly improve the power of fault analysis attacks on SHA-3 systems. Finally, I make use of the clear algebraic properties of SHA-3 to protect it using error detection codes efficiently with minimal area and time overhead.
Chapter 1

Introduction

1.1 Motivation

Keccak, a family of sponge functions, can be used to build various security modules widely used in crypto systems, including hash function, symmetric cryptographic function, pseudo-random number generator and authenticated encryption [1]. The new SHA-3 standard [2] is based on Keccak. Two candidates for CAESAR (Competition for Authenticated Encryption: Security, Applicability, and Robustness), Ketje and Keyak, are also built upon Keccak [3]. Therefore, the security of Keccak against different kinds of attacks is critical to security system design.

Security analysis of Keccak (and SHA-3) started with cryptoanalysis of the algorithms [4, 5, 6, 7, 8, 9], including collision attacks [6] and cube attacks [7], etc. There are some prior works focusing on security analysis of the implementations, including fault injections attacks [10] and side-channel power analysis of the software implementations [11, 12]. However, different side-channel vulnerabilities of Keccak have not been evaluated thoroughly, not mentioning the follow-on protections against these attacks.

Side-channel attacks is a kind of attack method using side-channel leakages of cryptosystem implementations such as power consumption, electromagnetic emanation (EMs) and timing to extract the secret information [13, 14], and it has been widely used to break various cryptographic systems such as DES, AES, RSA, etc [13, 15]. Another kind of implementation attacks is fault analysis, which utilizes the dependency of the output faults on the internal intermediate variables to recover the secret. Two fault analysis methods, namely differential fault analysis (DFA) and algebraic fault analysis (AFA), have been devised to break various crypto implementations [16, 17, 18, 19, 20, 21, 22, 23, 24, 25].

Responding to the side-channel analysis and fault analysis threats, many works also focus on
the protection of crypto systems against such attacks [26, 27, 28, 29, 30, 31, 32]. For side-channel power/EM attacks, there are two kinds of widely used countermeasures, namely power balance circuits and masking. Power balance circuits make the power consumption independent of the input data [26, 27]. Masking introduces random numbers to mask sensitive information [29, 30]. To resist against fault attacks, error detection codes (EDCs) and different kinds of sensors are designed to detect injected faults [33, 34, 35, 36, 37, 38, 39, 40, 41, 42].

For side-channel power analysis of SHA-3, previous papers all focus on software implementations [11, 12] in which intermediate results are stored in registers, while no side-channel methods specifically for hardware implementations of Keccak have been proposed. Compared to Keccak software implementations, hardware implementations of Keccak have coarser granularity of operations and storage, and therefore some previous models for software systems are not suitable for hardware implementations. Since hardware implementations of Keccak will certainly be used extensively in performance-sensitive crypto systems, their side-channel leakages must be thoroughly assessed with more general side-channel attacking methods for hardware implementations. Furthermore, a good understanding of the side-channel power leakage will also facilitate secure hardware design and implementation.

Only one prior work on fault analysis of SHA-3 has been published [10]. The work is preliminary in terms of both the fault model (a simple and unrealistic single-bit model), and the limited modes of SHA-3 (only the two modes with longer digests, SHA3-512 and SHA3-384). This single-bit fault model requires sophisticated, high-precision, and invasive fault injection methods, such as laser emission and ion beaming. While general fault injection methods, such as clock glitches and supply voltage disturbance, would affect a group of bits in intermediate states all together. The prior work also leaves out other two modes with shorter digest, SHA3-224 and SHA3-256. In view of these shortcomings of the state-of-the-art work, we significantly advance fault analysis attacks on SHA-3 by developing effective attack methods under much more relaxed realistic fault models, e.g., single-byte or single-word, and extending the attacks to all four modes of SHA-3, with the aid of algebraic techniques.

Various countermeasures have been proposed to protect different ciphers and their implementations against side-channel power analysis and fault analysis [33, 35, 38, 43, 44, 45, 46, 47]. Such countermeasure are not designed for SHA-3 implementations specifically. I make use of the properties of Keccak function to design effective and efficient countermeasures against side-channel power attacks and fault injection attacks. I also adopt the prevalent TI method against power analysis attacks, and implement TI in an open-source compiler to automatically generate side-channel attacks resilient code.
CHAPTER 1. INTRODUCTION

1.2 Research Agenda

This dissertation investigates both side-channel analysis and protection of SHA-3 systems. We consider both power analysis and fault injection attacks.

I will first give preliminaries of SHA-3 in Chapter 2 and then address the following topics in the remaining chapters.

1. **Side-channel power analysis of SHA-3 hardware implementation.** In Chapter 3 I will develop side-channel power leakage models and attack method specifically for hardware implementations of SHA-3. All attacking data will be sampled from the implementation on SASEBO-GII board [48]. The target is to extract the secret key used in MAC-Keccak hardware implementations.

2. **Protection of SHA-3 against side-channel power analysis.** I will present protection methods against side-channel power analysis attacks:

   (a) In Chapter 4 I will protect SHA-3 against side-channel attacks by automatically shuffling the operations thus to decrease the side-channel leakages.

   (b) In Chapter 5 I will implement the TI method in a compiler so as to automatically generate protected SHA-3 implementations.

3. **Fault injection attacks on SHA-3.** I will analyze the fault propagation in SHA-3 by using DFA techniques. I will also improve the attacks with algebraic techniques. The common goal of attacks is to extract the secret key when SHA-3 is used in the MAC mode, and recover the input messages when SHA-3 is used in the general hashing mode.

   (a) **Differential fault analysis of SHA-3.** I will extend DFA to all four SHA-3 functions, under much more relaxed fault models. All attacks will be simulated in C++. Details will be given in Chapter 6.

   (b) **Algebraic fault analysis of SHA-3.** In Chapter 7 I will employ algebraic techniques into the fault analysis of SHA-3 such that the attacks will be much more effective and efficient comparing with DFA.

4. **Protection of SHA-3 against fault injection attacks.** In Chapter 8 I will present our efficient error detection codes utilizing special properties of SHA-3 operations, to detect injected faults early so as to prevent the attackers from getting faulty digests for analysis. I will also propose a method to detect system clock glitches and power disturbance commonly used for fault injection.
CHAPTER 1. INTRODUCTION

Finally, I will conclude my works in Chapter 9.
Chapter 2

Preliminaries of SHA-3

Standardized by NIST, SHA-3 functions operate in modes of Keccak-\(f[1600, d]\), where each internal state is 1600-bit organized in a 3-D array (\(5 \times 5 \times 64\), i.e., 5 bits in each row, 5 bits in each column and 64 bits in each lane), and \(d\) is the capacity and also the output length at choice. All of the 1600-bit states are organized in a 3-D array, as shown in Figure 2.1. Each bit is addressed with three coordinates, written as \(S(x, y, z)\), \(x, y \in \{0, 1, ..., 4\}, \ z \in \{0, 1, ..., 63\}\). 2-D entities, plane, sheet and slice, and 1-D entities, lane, column and row, are also defined in Keccak and shown in Figure 2.1.

![Figure 2.1: Terminology used in Keccak](image)

The SHA-3 family includes four output lengths, called SHA3-224, SHA3-256, SHA3-384, and SHA3-512 [2]. Keccak relies on a Sponge architecture to iteratively absorb message inputs and squeeze out digests by a \(f\) permutation function. Each \(f\) function works on a state at a fixed length \(b = r + c\).

For attacks, we simplify the setting of SHA-3 by assuming that only one \(f\) function is involved...
for absorbing and squeezing. Then we extend the proposed attacks to systems with longer input messages which involves multiple $f$ functions. The attack goal is to recover the authentication key when SHA-3 is used in MAC mode, or to recover the input message when SHA-3 is used in hash mode. The $f$ function consists of 24 rounds for 1600-bit operations, and each round has five sequential steps:

$$S_{i+1} = \iota \circ \chi \circ \pi \circ \rho \circ \theta(S_i), \ i \in \{0, 1, \cdots, 23\} \quad (2.1)$$

in which $S_0$ is the initial input. Details of each step are described below:

$\theta$ is a linear operation which involves 11 input bits and outputs a single bit. Each output state bit is the XOR between the input state bit and two intermediate bits produced by its two neighbor columns. The operation is given as follows:

$$\theta_0(x, y, z) = \theta_1(x, y, z) \oplus (\oplus_{y=0}^{4} \theta_i(x-1, y, z)) \oplus (\oplus_{y=0}^{4} \theta_i(x+1, y, z-1)). \quad (2.2)$$

As shown in Figure 2.3, the two intermediate bits are the parity of the two columns, $\oplus_{y=0}^{4} \theta_i(x-1, y, z)$ and $\oplus_{y=0}^{4} \theta_i(x+1, y, z-1)$, respectively.

As described in [11], $\theta$ operation is computed over two successive steps. In the first step, $\theta_1$ calculates the parity of each column and compresses a 1600-bit state into a 320-bit plane called the $\theta_{plane}$:

$$\theta_{plane}(x, z) = \oplus_{y=0}^{4} \theta_i(x, y, z)$$

$$x \in \{0, 1, \ldots, 4\}, \ z \in \{0, 1, \ldots, 63\}. \quad (2.3)$$

$\theta_1$ can be viewed as 320 independent operations, each of which XORs the five bits in a column to generate one bit of parity.
In the second step, $\theta_2$ computes the XOR between every bit of the state and two neighboring parity bits of the $\theta_{plane}$, and outputs $\theta_o$. 

$$\theta_o(x, y, z) = \theta_l(x, y, z) \oplus \theta_{plane}(x - 1, z) \oplus \theta_{plane}(x + 1, z - 1)$$

$$x, y \in \{0, 1, \ldots, 4\}, \quad z \in \{0, 1, \ldots, 63\}. \quad (2.4)$$

- $\rho$ is a rotation over the state bits along z-axis (in lanes), and it is shown in Figure 2.4.

- $\pi$ is a permutation over the bits of the state within slices and $\pi$ operation is shown in Figure 2.5. Only the center bit ($x = 0, y = 0$) of the slice does not move. All other bits are permuted to other positions depending on their original coordinate. We also notice that five bits of each row will be moved to different rows but the same column. For each output row of $\pi$, one and only one bit is from the bottom row ($y = 0$) of the input.
− $\chi$ is a non-linear step that contains mixed binary operations. Every bit of the output state is the result of an XOR between the corresponding input state bit and its two neighboring bits along the x-axis (in a row) and its operation is shown in Figure 2.6.

$$\chi_o(x, y, z) = \chi_i(x, y, z) \oplus (\chi_i(x + 1, y, z) \cdot \chi_i(x + 2, y, z)).$$

The $\chi$ operation is reversible, each $\chi_i$ bit can be expressed in below formula which involves all
five bits of $\chi_o$ in a row \[49\]:

$$\chi_i(x, y, z) = \chi_o(x, y, z) \oplus \chi_o(x + 1, y, z) \cdot (\chi_o(x - 1, y, z) \oplus \chi_o(x + 2, y, z)$$

$$\oplus \chi_o(x - 1, y, z) \cdot \chi_o(x + 3, y, z)). \quad (2.6)$$

$\iota$ is a binary XOR with a round constant.

For MAC function, MAC-Keccak is recommended by the Keccak designers \[50\] \[1\]:

$$\text{MAC}(M, K) = H(K||M). \quad (2.7)$$

Further details of Keccak and Sponge construction can be found in \[1\].
Chapter 3

Side-Channel Power Analysis of SHA-3

3.1 Introduction and Motivation

Keccak has been implemented in software or a library for various general-purpose processors, embedded processors, and DSPs. In addition, there have also been hardware implementations of Keccak, which can accelerate hash computations in security-sensitive applications. As an example, a virtual network server typically uses a hash function both to verify the identity of its clients and the integrity of the messages sent by the clients. Hardware implementations of hash functions are preferred for their high throughput. High-speed integrated circuits (ICs) and FPGAs-based hash modules are widely used in cryptographic systems, Trusted Platform Modules (TPM), and crypto processors. However, the resiliency of hardware implementations of MAC-Keccak against side-channel power analysis has not been evaluated quantitatively.

In [51], six SHA-3 candidates (including Keccak) were analyzed for side-channel leakage for the first time. In the paper, the authors analyzed the probability of attacking MAC-Keccak at the $\theta$ step in the first round and proposed the basic steps for the attack. They presented a general side-channel attack feasibility analysis for all six candidates, though no detailed leakage models or attack methods are given for Keccak. In [11] and [12], the side-channel vulnerability of software implementation of MAC-Keccak was analyzed. The work demonstrates a practical side-channel analysis attack on MAC-Keccak implemented on a 32-bit Microblaze processor. They focused on a software implementation, but did not describe how to carry out attacks on hardware implementations.

The previous works focus on software implementations of MAC-Keccak while these methods cannot be applied to MAC-Keccak hardware implementations directly. Intermediate variables used for software implementation attacks are hidden to attackers in hardware implementations. Thus the leakage models and attack methods cannot be directly used for attacking hardware implementations.
CHAPTER 3. SIDE-CHANNEL POWER ANALYSIS OF SHA-3

Compared to the software implementations of MAC-Keccak [11][12], we anticipate much lower side-channel leakage of hardware implementations of MAC-Keccak with their highly parallel implementations (versus the temporally spread-out software implementations). Thus, side-channel analysis on a hardware implementation of MAC-Keccak would be much more challenging to implement.

Without a good understanding of the sources and amount of side-channel leakage of MAC-Keccak, any protection against side-channel attacks [52][53][54] cannot be evaluated accurately either. To address these issues and facilitate secure MAC-Keccak design and implementation, in this work we propose several side-channel leakage models of hardware MAC-Keccak and launch practical power analysis attacks on the implementations. In light of the popular usage of Keccak as the new hash standard, our work should make a significant contribution for side-channel security analysis of SHA-3. We further discuss the factors that affect side-channel leakage of MAC-Keccak and suggest countermeasures to improve its system security.

The contributions of this work are as following:

• We design attack methods specifically for hardware implementations of Keccak using the leakages of the first round output.

• We find the mathematical properties which cause leakages of intermediate variables that can be used for attacks as in software implementations.

• We discuss how different key length affects the side-channel security and propose a simple method to detect the key length.

3.1.1 Keccak Parameters Used in This Chapter

Keccak allows the use of a variable-length key, which is concatenated with the message and then broken down into message blocks (each at the bitrate, and the last one is padded according to a certain rule). Similar as previous papers [11][12], we start with a key length of 320 bits (i.e., the key fills the bottom plane in the input state, and the message length is 1024 – 320 = 704 bits). The sponge function (shown in Figure 2.2) contains only one $f$ function. Then we extend the attacks to implementation with 640 key bits and discuss how key length affect the side-channel security of MAC-Keccak implementations.

For the first setting, the key bits fill the first plane (320 bits, denoted as $K$ plane) of the input state and the other four planes (1, 280 bits, denoted as $M$ planes) only contain message bits and padding bits which are known to the attackers. Thus for $\theta$ operation, each bit of $\theta_{plane}$ involves four bits of $M$ and one bit of $K$, and each bit in $M$ planes of $\theta_o$ involves eight bits of $M$ and two bits of $K$. For the 640 key bits setting, the key bits fill the first two planes, and each bit of $\theta_{plane}$ involves
three bits of $M$ and two bits of $K$, and each bit in $M$ planes of $\theta_o$ involves six bits of $M$ and four bits of $K$.

We note here that after padding operation, $M$ includes both the message bits and the appended 576 bits. For the simplified setting in this work, there is only one message block and all the appended bits are 0, and all bits of $M$ including message bits and padding bits are known to users and attackers. We still use $P$ to denote the input of the first round for convenience.

In software implementations, all the intermediate variables are stored into processor registers and thus the caused leakages can be used for attacks. For example, in [11] and [12], the authors make use of the intermediate variables of $\theta_1$ operation to recover the key bits for software implementations. For hardware systems, all operations of one round are implemented using combinational logic and thus these intermediate variables are hidden from the attackers. In this work, we will show how to extract leakages of these intermediate variables based on the mathematical properties of Keccak and the hardware implementation details.

We referred to [55] and implement the most widely used implementation [56] and [57] on a SASEBO-GII board [48] which contains a Xilinx Virtex-5 FPGA as crypto engine. For the official VHDL implementation Version 3.1 [56] and [57], each round takes one clock cycle and the five steps of one round ($\theta$, $\rho$, $\pi$, $\chi$ and $\iota$) are all implemented in combinational circuits. We use Xilinx ISE 14.6 for implementation and we use all the default settings for all operations including synthesis, mapping and routing. The MAC-Keccak system runs at 12 MHz and no other operations running on the crypto FPGA. We collect all the traces using an Agilent MSOX4104A oscilloscope.

We note here that we implement and attack both [56] and [57], their leakage results are very similar and attack methods are the same. Thus in this work, we use all results from [56] for analysis. Meanwhile, we use the most widely used side-channel analysis methods, both CPA and Mutual Information Analysis (MIA) [58] for analysis, and get similar results for these two methods. Thus we use CPA to discuss the side-channel leakages in this work.

3.2 Attack on the first round output

For previous side-channel attacks on software MAC-Keccak implementations, the leakages of first round output $R_1$ have not been investigated. This is because previous papers [11, 12] focus on the leakages of intermediate variables in $\theta$ operation which involve the key bits directly, and attacks based on these intermediate variables are efficient for software implementations. In this section, we investigate the leakage of the first round output $R_1$, and show the key bits extraction method based on the $R_1$ leakages.
CHAPTER 3. SIDE-CHANNEL POWER ANALYSIS OF SHA-3

3.2.1 Side-Channel Leakage of \( R_1 \)

For most hardware implementation of MAC-Keccak, all five steps are implemented using combinational circuits and they are executed in one clock cycle. Thus only the round output are stored in registers while the intermediate results are not. If the initial data of the state register is \( P \) before the first round operation, the Hamming distance will be \( HD(P, R_1) \) when the first round result \( R_1 \) is stored into the state register. Because the first round input \( P \) is partially known to the attackers (the message bits and appended 0 bits), this Hamming distance model can be used for side-channel analysis.

For implementations like \([56]\), the state register \( R \) changes from 0 to \( R_1 \) at the end of the first round, and the Hamming distance model can be further simplified as \( HW(R_1) \). Based on the activity of the state register after the first round, we then have the following leakage model:

- **Model I** The state register changes from 0 to \( R_1 \) at the end of the first round. There is a strong correlation between \( HW(R_1) \) and power: \( \text{correlation}(HW(R_1), power) \).

We note here that for implementations with state register initialized as \( P \), the side-channel analysis method is similar and further discussions will be given in Section 3.4. Using the settings described in Section 3.1.1, we first implement the design with 320 key bits and sample 500,000 power traces for analysis. We show one power trace and the Pearson correlation result between \( HW(R_1) \) and power in Figure 3.1.

Figure 3.1(a) shows an example power trace of MAC-Keccak, 24 rounds can be clearly observed on the trace. In Figure 3.1(a), the first trough corresponds to the loading of message \( M \) into the padding module and then \( P \) to the \( \theta \) module. The second trough corresponds to the state register changing to the first round result \( R_1 \). Meanwhile, Figure 3.1(b) shows the correlation between \( HW(R_1) \) and the power. This correlation is strong and reaching about 0.5 (negative). Thus it’s absolutely possible to conquer the MAC-Keccak hardware system by attacking \( R_1 \).

3.2.2 Side-Channel Analysis on \( R_1 \)

In each round of Keccak, \( \iota \) only adds a constant number to the output of \( \chi_o \), and this constant number is publicly known. Thus attacking \( R_1 \) is attacking \( \chi \) operations for Keccak. Each \( \chi_o \) bit involves three bits of \( \chi_i \), which can be mapped back to three bits of \( \theta_o \) according to the operations of Keccak. Thus each bit of \( R_1 \) involves at least six key bits and thus attacking single bit of \( R_1 \) is with high complexity.

For side-channel attacks, multiple bits are usually attacked together to improve the signal-to-noise-ratio. We will show how different bits in \( R_1 \) can be attacked together in this section. Specif-
Figure 3.1: Power trace example and the leakage of $R_1$

...we show a practical attack method on $R_1$ which attacks five bits in one row together, i.e., $R_1(X, y, z), X = [0 : 4], y \in \{0, 1 \cdots 4\}, z \in \{0, 1 \cdots 63\}$. We notice that the five bits in each row of input state of $\chi$ operate on each other:

$$\chi_o(X, y, z) = \pi_o(X, y, z) \oplus (\pi_o(X + 1, y, z) \cdot \pi_o(X + 2, y, z)),$$

$$X = [0 : 4], y \in \{0, 1, \cdots, 4\}, z \in \{0, 1, \cdots, 63\}. \quad (3.1)$$

Each bit of $\pi_o(x, y, z)$ can be mapped to a bit of $\theta_o(x', y', z')$ according to the permutations rules of $\pi$ and $\rho$. Then we can denote the mapping as:

$$(x', y', z') = map^{-1}_\pi(map^{-1}_\rho(x, y, z)). \quad (3.2)$$

Meanwhile, $\theta_o(x', y', z')$ can be denoted as:

$$\theta_o(x', y', z') = P(x', y', z') \oplus (\oplus_{i=0}^4 P(x' - 1, y', z')) \oplus (\oplus_{i=0}^4 P(x' + 1, y', z' - 1)),$$
in which \( P(\theta_i) \) is the first round input of Keccak. Thus all the input bits of Equation (3.1) can be denoted using the first round input (including the known plaintext and the key bits). We note here that according to the property of \( \pi \), each row of \( \pi_o \) involves one and only one bit from the bottom plane of \( \theta_o \), and four bits from the \( M \) planes of \( \theta_o \). Thus in total, each output row of \( \chi \) actually involves 11 bits of \( K \).

Comparing with attacking each bit of \( R_1 \) separately, attacking one row of \( R_1 \) together will help to increase signal-noise-ratio while the key-guess complexity only increases slightly. Attackers can make assumptions of the key bits and combine them with the known plaintext bits to calculate the first round output \( R_1 \). The first round output \( R_1 \) based on the correct key assumption should have higher correlation with the power consumption than incorrect assumptions. We use real attack result as an example to demonstrate the attacks on \( R_1 \).

**Example 3.2.1** For the row \( R_1([0:4], 4, 0) \), the mapping relations are as following:

\[
\begin{align*}
\pi_o(0, 4, 0) & \leftrightarrow \theta_o(2, 0, 2) \\
\pi_o(1, 4, 0) & \leftrightarrow \theta_o(3, 1, 9) \\
\pi_o(2, 4, 0) & \leftrightarrow \theta_o(4, 2, 25) \\
\pi_o(3, 4, 0) & \leftrightarrow \theta_o(0, 3, 23) \\
\pi_o(4, 4, 0) & \leftrightarrow \theta_o(1, 4, 62)
\end{align*}
\]

As \( \theta_o(2, 0, 2) \) is in the bottom plane, it involves \( P(2, 0, 2), P(1, 0, 2) \) and \( P(3, 0, 1) \), three key bits. The other four \( \theta_o \) bits each involves two key bits. Therefore, this row involves 11 key bits in total. Using this attack model, for each row of \( R_1 \) we can recover one XOR of three key bits and four XORs of two key bits. In this example, we can recover:

\[
\begin{align*}
kg_1^I &= P(2, 0, 2) \oplus P(1, 0, 2) \oplus P(3, 0, 1) \\
kg_2^I &= P(2, 0, 9) \oplus P(4, 0, 8) \\
k_g^I &= P(3, 0, 25) \oplus P(0, 0, 24) \\
kg_4^I &= P(4, 0, 23) \oplus P(1, 0, 22) \\
kg_5^I &= P(1, 0, 62) \oplus P(2, 0, 61)
\end{align*}
\]

Using the Hamming distance power model \( HW(R_1([0:4], 4, 0)) \) to calculate correlations, the correlation result of 5 state bits (\( R_1([0:4], 4, 0) \)) based on 500,000 traces is shown in Figure 3.2(a), which leaks five XOR operations of key bits, while Figure 3.2(b) gives the correlation result of attacking a single bit \( R_1(0, 4, 0) \), which leaks three key bits operations, \( (kg_3^I, kg_4^I \text{ and } kg_5^I) \). For one row, the maximum correlation is about 0.042; while for one single bit it is about 0.022. Compared to
CHAPTER 3. SIDE-CHANNEL POWER ANALYSIS OF SHA-3

Figure 3.2: Leakage of one row of $R_1$ vs. leakage of one bit of $R_1$

The correlation result (up to 0.5) for 1600 state bits shown in Figure 3.1 these two correlations are much smaller.

Figure 3.2 indicates that aggregating five bits in a row for attack would give higher leakage, i.e., correlation of 0.042 to recover five XORs results between 11 key bits. Using the Hamming distance power model $HW(R_1([0:4], 4, 0))$ to run CPA, the correlation result of 5 state bits for different key guesses is shown in Figure 3.3 where the Y-axis is the correlation value and X-axis is the enumeration (key guess) of $[kg_5^I kg_4^I kg_3^I kg_2^I kg_1^I]$, with $kg_5^I$ as the most significant bit.

Figure 3.3 shows that the highest correlation value (negative) is $-0.042$, which is indeed at the correct guess. We can see that our attack model can effectively distinguish a group of five correct results of key bits’ operations from each row. The correlation result shown in Figure 3.3 is quite different from the previous block ciphers (e.g., AES, DES) where the correct key would give a much higher correlation than other incorrect keys. This is because Keccak does not have modules similar
CHAPTER 3. SIDE-CHANNEL POWER ANALYSIS OF SHA-3

Figure 3.3: Key guess result of $R_1([0:4], 4, 0)$ using 40,000 traces

to $S$-box which has very high non-linearity, therefore the difference between key guesses is not as large as in block ciphers.

To evaluate the efficiency of the proposed attack method, we run the attacks using different number of power traces and show the correlation and success rate results in Figure 3.4

Figure 3.4(a) shows the correlation results for different key guess ($[k_{g_5} k_{g_4} k_{g_3} k_{g_2} k_{g_1}]$) using different number of traces, the x-axis is the number of traces, while the y-axis is the corresponding correlation. In Figure 3.4(a), the correlation for the correct key guess is plotted in black color, while the correlation results for the wrong key guess are in grey color. From Figure 3.4(a), we can see that the correct key guess stands out quickly and our attack model can effectively find the correct key. Figure 3.4(b) shows the success rate of attacking the row $R_1([0:4], 4, 0)$. From Figure 3.4(b), we can see that the attacker needs about 35,000 traces to recover the secret key with a success rate of about 100%.

According to the above analysis and Example 3.2.1, where we use one row at a time to attack $R_1$, the attacker can recover one XOR of three key bits and four XORs of two key bits. Attackers can recover the XORs of key bits by attacking each row separately, and the XORs should be combined to recover the secret key bits.
Example 3.2.2 If we attack on row $R_1([0:4], 2, 8)$, we can recover:

$$
\begin{align*}
kg_1' &= P(1, 0, 7) \oplus P(0, 0, 7) \oplus P(2, 0, 6) \\
kg_2' &= P(1, 0, 2) \oplus P(3, 0, 1) \\
kg_3' &= P(2, 0, 41) \oplus P(4, 0, 40) \\
kg_4' &= P(3, 0, 0) \oplus P(0, 0, 63) \\
kg_5' &= P(4, 0, 54) \oplus P(1, 0, 53)
\end{align*}
$$

With the result of attacking $R_1([0 : 4], 4, 0)$ in Example 3.2.1 $P(2, 0, 2)$ can be recovered by combining the results of $kg_1' (P(2, 0, 2) \oplus P(1, 0, 2) \oplus P(3, 0, 1))$ and $kg_2' (P(1, 0, 2) \oplus P(3, 0, 1))$.

The output state $R_1$ consists of 320 rows, and these 320 rows can be attacked in parallel and their results can be utilized together to recover the key bits. For 320 rows, we can recover 320 3-bit...
XORs and 1280 2-bit XORs. For these 1280 2-bit XORs, the redundancy is very high and there are only 320 distinct 2-bit XORs with each appearing four times. We can denote all these XORs as following:

\[
\begin{align*}
KG_1 &= \{ P(x-1,0,z) \oplus P(x,0,z) \oplus P(x+1,0,z-1) \} \\
KG_2 &= \{ P(x-1,0,z) \oplus P(x+1,0,z-1) \}
\end{align*}
\]

for \( x \in \{0, 1 \cdots 4\}, z \in \{0, 1 \cdots 63\}. \)

Because each of the 320 3-bit XORs from group \( KG_1 \) contains one more extra key bit than one 2-bit XORs from group \( KG_2 \), we can combine them to recover all of the 320 key bits immediately.

For the case where the key length is greater than 320 bits, we will discuss the details of attacks in Section 3.2.3.

From Examples 3.2.1 and 3.2.2, we can see that attackers can recover all the key bits by attacking \( R_1 \), and it shows that an unprotected MAC-Keccak hardware implementation is vulnerable to side-channel attacks. This attack method is very different from the previous attacks for software implementations. For software implementations, five bits for one row of \( R_1 \) are operated at different time, thus this attacking method is not suitable for software implementations.

### 3.2.3 Attacks on \( R_1 \) for Different Key Length

For key-length \( \leq \) plane-size, the key bits will occupy no more than one plane and there is no operation between the key bits in the same column. We can attack on \( R_1 \) to recover all the key bits, as discussed in Section 3.2.2.

When plane-size < key-length \( \leq 2 \times \) plane-size, the key bits occupy more than one plane. There will be operations between key bits from different planes. We can recover XORs of different numbers of key bits. For example, assume the key is 640 bits, thus the key bits take two planes \( P_0 (P(X,0,Z)) \) and \( P_1 (P(X,1,Z)) \). Then attackers can attack all the rows of \( R_1 \) to recover higher-dimension key bit XORs:

\[
\begin{align*}
KG_1^{I*} &= \{ P(x-1,0,z) \oplus P(x-1,1,z) \\
&\quad \oplus P(x+1,0,z-1) \oplus P(x+1,1,z-1) \} \\
KG_2^{I*} &= \{ P(x-1,0,z) \oplus P(x-1,1,z) \oplus P(x,0,z) \\
&\quad \oplus P(x+1,0,z-1) \oplus P(x+1,1,z-1) \} \\
KG_3^{I*} &= \{ P(x-1,0,z) \oplus P(x-1,1,z) \oplus P(x,1,z) \\
&\quad \oplus P(x+1,0,z-1) \oplus P(x+1,1,z-1) \}
\end{align*}
\]

in which \( x \in \{0, 1 \cdots 4\}, z \in \{0, 1 \cdots 63\}. \)
CHAPTER 3. SIDE-CHANNEL POWER ANALYSIS OF SHA-3

In this case, we can recover 320 5-bit XORs $K G_1^1$, 320 5-bit XORs $K G_2^2$ and 320 distinct 4-bit XORs $K G_3^3$ in total. Similar as the method when the key size is no greater than 320 key bits, we can combine XORs results to recover the entire $K$.

When $key-length > 2 \times plane-size$, the key bits will occupy more than two planes and there will be no plane composed only by message bits. It may not be possible to attack and recover all the key bits.

To verify the above conclusion, we implement the 640 key bits MAC-Keccak based on the discussion in Section 3.1.1 and we also sample 500,000 traces for this implementation to run side-channel analysis. We attack the same row as Example 3.2.1 and this will help us to recover four 4-bit XORs and one 5-bit XOR, the corresponding key enumeration and success rate results are shown in Figure 3.5.

![Figure 3.5](image-url)

Figure 3.5: Correlation and success rate results of attacking $R_1([0:4],4,0)$ for key length 640 bits

Figure 3.5 shows that the success rate for 640 key bit implementation is almost the same as
CHAPTER 3. SIDE-CHANNEL POWER ANALYSIS OF SHA-3

the success rate for the 320 key bit implementation. This is because both implementations use
the Hamming weight of five \( R_1 \) bits (one row) and thus their signal-noise-ratios are equal while
implemented on the same platform. Thus, the success rates are not changing when the key length
changes, and the attackers can still plug the 4-bit XORs into the 5-bit XORs to extract all the key bits.

3.3 Attacks Based on the Leakages of \( \theta \) Operation in the First Round

\[ \text{correlation}(\text{HW}(R_1), \text{HW}(\theta o)) \]

\[ \text{correlation}(\text{HD}(P, R_1), \text{HD}(P, \theta o)) \]

Figure 3.6: Correlation relationship of variables in Keccak

In previous section, we show a method to recover all the key bits in MAC-Keccak by attacking
the first round output \( R_1 \). Our method is different from previous papers which focus on attacking \( \theta \)
operations in software implementations. In this section, we show that although intermediate results
of \( \theta \) operations are not stored in registers, the combinational circuits activities and the mathematical
properties of Keccak will still cause leakages of \( \theta \), and we will further show how these leakages can
be used for side-channel analysis in hardware implementations.

3.3.1 Side-Channel Leakages of \( \theta \) Operations

For hardware implementations, leakages not only come from the register activities, but also
the combinational circuits activities. Thus we look into the properties of the circuit for leakage of \( \theta \)
operations. Meanwhile, in this section, we also show that the mathematical properties of Keccak
make the leakage of \( \theta \) very strong and such leakages can be used to recover the key bits.
CHAPTER 3. SIDE-CHANNEL POWER ANALYSIS OF SHA-3

As defined in previous sections, we denote the input of the first round as $P$, the result of the first round as $R_1$, the result of the $\theta$ operation of the first round as $\theta_o$. We find that there is a strong correlation between $HW(R_1)$ and $HW(\theta_o)$. This is because $R_1 = \iota \circ \chi \circ \pi \circ \rho(\theta_o)$, in which $\pi$ and $\rho$ are linear operations and they only change the positions of the bits and will not affect the Hamming weight of the temporary value:

$$HW(\theta_o) = HW(\pi \circ \rho(\theta_o)). \quad (3.3)$$

According to the details of $\chi$ operation, $\chi_o(x, y, z)$ keeps the same value as $\chi_i(x, y, z)$ with a probability 0.75 if $\chi_o(x + 1, y, z)$ and $\chi_o(x + 2, y, z)$ are random. At the same time, $\iota$ only changes one single bit of $\chi_o$ for the first round. So for randomly generated input, $R_1$ and $\pi_o$ have most of the bits with the same value, thus there should be a strong correlation between $HW(\theta_o)$ and $HW(R_1)$.

To verify the above assumption, we randomly generate 200,000 messages (all 1,600 bits are random) and run Keccak on them. Then we run Pearson’s correlation on the results and get the correlation results shown in Figure 3.6.

Figure 3.6 shows that the correlation between $HW(R_1)$ and $HW(\theta_o)$ is very strong, which is about 0.538. Meanwhile, we find that there is also a strong correlation between $HW(R_1)$ and $HD(P, \theta_o)$, between $HW(R_1)$ and $HD(P, R_1)$, and between $HD(P, R_1)$ and $HD(P, \theta_o)$:

$$\begin{align*}
\text{correlation}(HW(R_1), HW(\theta_o)) &= 0.538 \\
\text{correlation}(HW(R_1), HD(P, R_1)) &= 0.340 \\
\text{correlation}(HW(R_1), HD(P, \theta_o)) &= 0.344 \\
\text{correlation}(HD(P, R_1), HD(P, \theta_o)) &= 0.176
\end{align*}$$

Meanwhile, the combinational circuits activities can also result in leakages of $\theta$ operations. We run correlation on the collected 500,000 traces to verify the proposed leakage models and show the result in Figure 3.7.

Figure 3.7 shows that although the results of $\theta$ operations are not stored in registers, there are still strong leakages caused by combinational leakages and mathematical properties. Thus, the leakages of $R_1$ and combinational circuits activities will cause leakages of $\theta$ operations. These leakages can be utilized by attackers to attack on $\theta$ operations to recover the key bits like in software implementations [11, 12]. Thus we have the following side-channel leakage models:

- **Model II** There is a strong correlation between $HW(\theta_o)$ and the power consumption ($correlation(HW(\theta_o), power)$), and this leakage can be used for side-channel analysis.

- **Model III** There is a strong correlation between $HW(\theta_{plane})$ and the power consumption ($correlation(HW(\theta_{plane}), power)$), and this leakage can be used for side-channel analysis.
CHAPTER 3. SIDE-CHANNEL POWER ANALYSIS OF SHA-3

![Figure 3.7: Leakages of $\theta_o$ and $\theta_{plane}$](image)

Attack results on $\theta$ operations can be used as an assistance for attacking $R_1$ output, and their results can be cross-checked for verification.

### 3.3.2 Side-Channel Analysis on $\theta_o$

For MAC-Keccak input, $P = K || M$ where the unknown part is the key $K$ (320 bits), and the known part is the message $M$ (704 bits) and the padding bits (576 bits). According to (2.2), the Hamming weight for a single bit of $\theta_o$ is:

\[
HW(\theta_o(x, y, z)) = HW(P(x, y, z) \oplus \theta_{plane}(x - 1, z)) \\
\oplus \theta_{plane}(x + 1, z - 1) = HW(P(x, y, z) \oplus (\oplus_{y=0}^4 P(x - 1, y, z)) \oplus (\oplus_{y=0}^4 P(x + 1, y, z - 1))).
\]  

(3.4)
Equation (3.4) shows that the each bit of $\theta_o$ involves the input bit and its two neighboring columns. If $y = 0$, the target bit $P(x, y, z)$ is also a key bit and thus three key bits, $P(x-1, 0, z)$, $P(x, 0, z)$ and $P(x+1, 0, z-1)$ are involved in (3.4); otherwise, two unknown key bits, $P(x-1, 0, z)$ and $P(x+1, 0, z-1)$ are involved.

In this work, we use the second situation which involves two key bits for attacks. To increase the leakage signal strength of CPA, instead of using 1 bit in a column for the select function, we use 4 bits ($y = 1, 2, 3, 4$) instead. To make a balance between the leakage signal strength and the attack complexity, each time we can attack multiple bits ($l, 0 < l \leq 64$) in each key lane. The larger the $l$, the stronger the leakage signal, but also the higher complexity due to enumeration of the key bits. The Hamming weight model becomes:

$$
\sum_{x=1}^{i+l-1} \sum_{y=1}^{4} \text{HW}(\theta_o(x, y, z)),
$$

$$
i \in \{0, 1, ..., 63\}, l \in \{1, 2, ..., 64\}, (3.5)
$$

and we can recover the corresponding $P(x-1, 0, Z) \oplus P(x+1, 0, Z-1)$ bits, $x \in \{0, 1, \cdots, 4\}$, $Z = [i : i + l - 1]$.

**Example 3.3.1** We choose $l = 8$ in each lane for attacks, and try to recover the relationship between $K(0, 0 : 7) \oplus K(2, 63 : 6)$, or denoted as $P(0, 0, 0 : 7) \oplus P(2, 0, 63 : 6)$ as an example here. We use the Hamming weight of $\theta_o(1, [1 : 4], [0 : 7])$ for power analysis. Thus the corresponding Hamming weight is $\text{HW}(\theta_o(1, [1 : 4], [0 : 7]))$. We run this attack for different key guess, and the corresponding result is shown in Figure 3.8(a).

Figure 3.8(a) shows the correlation analysis result for key guesses based on Model II. It shows that the correct key candidate has the highest (negative) correlation and stands out of different key guesses. For MAC-Keccak, $\theta$ is a linear operation and one bit key change has a limited affect on the Hamming weight of the output value. Thus we can see that the correlation result in Figure 3.8(a) is very different from attack results on previous block ciphers.

Then, we run CPA using the above Hamming distance model to get the corresponding empirical success rate, which is shown in Figure 3.8(b). The success rate based on Model II in Figure 3.8(b) shows that the attacker needs about 30,000 traces to recover the secret information with 100% success rate.

From the above analysis and Example 3.3.1 we can see that the attackers can efficiently recover the XOR relationship between key bits. Using this method, attackers can recover

$$
KG^{II} = \{P(x - 1, 0, z) \oplus P(x + 1, 0, z - 1)\},
$$

(3.6)
in which $x \in \{0, 1, \ldots, 4\}, z \in \{0, 1, \ldots, 63\}$.

Thus, by attacking $M$ lanes of $\theta_o$, attackers can recover the XORs between key bits. The recovered results can be used together with $R_1$ attacking results for attacks and cross-checking.

### 3.3.3 Side-Channel Analysis on $\theta_{\text{plane}}$

For the leakages of $\theta_{\text{plane}}$ shown in Figure 3.7(b), the correlation is between power and $HW(\theta_{\text{plane}})$, and the first step of theta, $\theta_1$, does not involve operations between two key bits. According to (2.3), each bit of the $\theta_{\text{plane}}$ is decided by 4 bits of $M$ and 1 bit of $K$ in one column. Thus, we can use this model to recover the key bits in one key lane directly.

For a single key bit, the SNR may be very low for $\theta_{\text{plane}}$, and more than one key bit can be attacked concurrently to increase the SNR. To strike a balance between SNR and complexity, we
CHAPTER 3. SIDE-CHANNEL POWER ANALYSIS OF SHA-3

attack 8 bits for each key lane at once. For the first 8 bits in lane \( x \), the Hamming weight is:

\[
HW(\theta_{\text{plane}}(x,[0:7])) = \sum_{z=0}^{7}(\oplus_{y=0}^{4}P(x,y,z)).
\] (3.7)

The CPA model is:

\[
correlation(\sum_{z=0}^{7}(\oplus_{y=0}^{4}P(x,y,z)), \text{power}).
\] (3.8)

The key byte guess with the highest correlation is the right key byte.

Example 3.3.2 We take \( \theta_{\text{plane}}(4,[0:7]) \) as an example, and launch CPA on this \( \theta_{\text{plane}} \) byte to recover the corresponding key byte. The corresponding Hamming weight is \( HW(\theta_{\text{plane}}(4,[0:7])) \). The key guess correlation result is shown in Figure 3.9(a) while the corresponding success rates of CPA are shown in Figure 3.9(b).

![Figure 3.9: Correlation and success rate results of attacking \( \theta_{\text{plane}}(4,0,[0:7]) \)]](image-url)
From Figure 3.9(b) we can see that when using about 500,000 traces, we can recover one byte of one key lane with SR around 90% while there are 8 bytes in each lane. The success rates are much lower for Model III than for Model II due to the much weaker correlation.

From the above analysis and Example 3.3.2 we can see that using the intermediate result of the \( \theta \) compression operation, we can recover all 64 bits in one key lane:

\[
KG^{III} = \{K(x, z)\} = \{P(x, 0, z)\}
\]  
(3.9)

for \( x \in \{0, 1 \cdots 4\}, z \in \{0, 1 \cdots 63\} \).

We note here that the leakage of \( \theta_{\text{plane}} \) is caused by combinational circuits activities and this leakage is much weaker than the leakages of \( R_1 \) and \( \theta_o \). Thus for attackers with limited number of traces, the attacks based on the leakages of \( R_1 \) and \( \theta_o \) are preferred to conquer MAC-Keccak.

### 3.3.4 Key Length

#### 3.3.4.1 How Key Length Affects the Attacks on \( \theta_o \)

For the attacks on \( \theta \) operations, when \( \text{plane-size} < \text{key-length} \leq 2 \times \text{plane-size} \), the key guess result will involve more than one key bits in the same column. For \( \text{key-length} = 2 \times \text{plane-size} \), use the method shown in Section 3.3.2 the attackers can recover

\[
KG^{II'} = P(x - 1, 0, z) \oplus P(x - 1, 1, z) \oplus P(x + 1, 0, z - 1) \oplus P(x + 1, 1, z - 1)
\]

for \( x \in \{0, 1 \cdots 4\}, z \in \{0, 1 \cdots 63\} \).

When the key length is greater than 704 bits, some columns will have no message bits any more and for such columns, our method will be no longer applicable.

#### 3.3.4.2 Key Length Affects the Attacks on \( \theta_{\text{plane}} \)

If the key size is greater than plane size, \( \text{plane-size} < \text{key-length} \leq 2 \times \text{plane-size} \), the key guess result will involve multiple bits in the same column. For \( \text{key-length} = 2 \times \text{plane-size} \), use the method shown in Section 3.3.3 the attackers can extract the following relationship:

\[
KG^{III'} = \{P(x, 0, z) \oplus P(x, 1, z)\}
\]  
(3.10)

for \( x \in \{0, 1 \cdots 4\}, z \in \{0, 1 \cdots 63\} \).

The same as the attacks on \( \theta_o \), when the key length is greater than 704 bits, some columns will have no message bits any more and for such columns, our method will be no longer applicable. The attackers can still attack on other columns which still have message bits and recover the relationship of the corresponding key bits.
CHAPTER 3. SIDE-CHANNEL POWER ANALYSIS OF SHA-3

3.4 Further discussion about the key-length and countermeasures

3.4.1 Side-Channel Attacks on MAC-Keccak with State Register Initialized to $P$

In this work, we focus on the official implementation which initializes the state register to 0 before the first round operations, and this makes the Hamming distance after the first round independent on $R_0$. Another kind of common implementation may initialize the state register to the first round input $R_0$ (which is also denoted as $P$ in this work). In this section, we will discuss side-channel attacks on this kind of implementation and show that our attack methods can be applicable to other unprotected MAC-Keccak hardware implementations.

For Model I, if the state register $reg$ is initialized to $R_0$, and the Hamming distance of the state register at the end of the first round will be:

$$HD(P(x_1, y_1, z_1), R_1(x_1, y_1, z_1)).$$

(3.11)

In this case, based on Model I, attacking the 256 rows in the top four planes gives us similar relationship we have found in previous section - 1024 2-bit XORs with 320 distinct ones and 256 3-bit XORs:

$$KG_{I_1}' = \{P(x - 1, 0, z) \oplus P(x, 0, z) \oplus P(x + 1, 0, z - 1)\},$$

$$x \in \{1, 2, 3, 4\}, z \in \{0, 1 \cdots 63\};$$

$$KG_{I_2}' = \{P(x - 1, 0, z) \oplus P(x + 1, 0, z - 1)\},$$

$$x \in \{0, 1 \cdots 4\}, z \in \{0, 1 \cdots 63\}.$$

For this situation, we can plug the 2-bit XORs $KG_{I_2}'$ into the 3-bit XORs $KG_{I_1}'$ and recover 256 key bits first. These 256 bits will be all the key bits except for the 64 bits in the lane $R_0(0, 0, Z)$. We then use the recovered 256 key bits and plug them back into the 2-bit XORs $KG_{I_2}'$ to recover the remaining 64 key bits. Thus, we can see that the attack method based on Model I we propose in this work is also applicable to implementations with $reg$ initialized to other known values instead of 0, e.g., $P$.

If the state register is initialized to $P$, for Model II, we have the following relationship:

$$\text{correlation}(HD(P, R_1), HW(\theta_o)) = 0.197$$

$$\text{correlation}(HD(P, R_1), HD(P, \theta_o)) = 0.176.$$  

Thus for this kind of operations, there is still a very strong leakage of $\theta_o$. Which means that we can use the same leakage model and attack method described in Section 3.3 to extract the relationship between key bits.
CHAPTER 3. SIDE-CHANNEL POWER ANALYSIS OF SHA-3

From the above discussion, we can see that the leakage models and attack methods described in this work can be widely used in other kinds of unprotected MAC-Keccak hardware implementations. The leakage models and attack methods change little for other kinds of implementations.

3.4.2 How to Detect the Key Length

In previous sections of this chapter, we just assume that the attacker has full knowledge of the key length and makes use of this knowledge for side-channel analysis. But in fact, this is not the case in real attacks and attackers need to find out the key length for MAC-Keccak himself. In this part, we present the method for attackers to find out the length of the key for real MAC-Keccak systems.

In [51], the authors propose to get the key length by performing differential power analysis (DPA). They start with the first message bit and repeatingly attacking the next message bit until there is no correlation in a directly subsequent point in time. This method is effective for software implementations which has all the operations expanded and the intermediate variables can be used for attacks. For hardware systems, their method is no longer applicable and we will show an applicable method in the following part.

Assume the key length is \( l_k \) and the message length is \( l_m \), and denote the appended length (for bitrate part) as \( l_a \) \((0 \leq l_a < r)\). Thus we have the following equation:

\[
l_k + l_m + l_a = n_b * r \tag{3.12}
\]

in which \( n_b \) is the number of blocks shown in Figure 3.10, and \( r \) is the bitrate which public to attackers. For Equation 3.12, \( l_a \) and \( n_b \) will change with \( l_m + l_k \) while \( l_k \) is a constant number and \( l_m \) is controllable by attackers. Attackers can decide the number \( n_b \) by counting how many runs of Keccak will be executed for the input message. Then he can figure out the length of the key. Before we demonstrate how attackers find the right key length, we give the an example of the relation between \( n_b \) and \( l_m \), the relation between \( l_a \) and \( l_m \) in Figure 3.10 \((l_m < r \text{ in this example})\).

Figure 3.10 contains the following information:

1. \( l_m + l_k + l_a \) should be a multiply of \( r \), and \( l_a \) will change to satisfy this relation;

2. When \( l_m + l_k \) is a multiply of \( r \), \( l_a \) will be 0, and increase of \( l_m \) will cause an abrupt change of \( n_b \) and \( l_a \).

Thus the attackers can gradually increase the length of message bits from 0 to find the length for this abrupt change. For example, if the number of block is \( n_b \) when the message length is \( l_m \), and \( n_b + 1 \) for \( l_m + 1 \), then length of key should be \( n_b * r - l_m \).
3.4.3 Simple Methods to Decrease Side-Channel Leakages

To protect MAC-Keccak from side-channel attacks, different methods have been proposed \cite{52, 53, 54}. These methods are based on secret sharing and multiple shares are needed to hide the secret key information from the attackers. This makes the resource overhead very high. To protect MAC-Keccak from side-channel attacks based on the leakage models in this work, there are some simple and effective methods.

In previous sections, we show that if the state register is initialized to either 0 or $P$, there will be strong leakages of $R_1$. At the same time, we show that if the state register is initialized to 0, there will be even strong leakages of $\theta_{plane}$. Then a direct sense is that we can initialize the state register to a random number to avoid such leakages.

To verify this assumption, we implement a 1600-bit random number generator based on a linear-feedback shift register (LFSR) and use the random numbers to initialize the state register $\text{reg}$ before the MAC-Keccak operations. We collect 400,000 traces and run Pearson’s correlation to look for correlation between $HW(R_1)$ and $\text{power}$. Results show that for such a simple protection
method, \textit{correlation}(HW(R_1), power) is only about 0.06, much smaller than the unprotected implementation (which reaches 0.5). The correlation is about 10 times less than the unprotected implementation, and therefore attackers will need about 100 times the number of traces to achieve the same success rate \cite{59} \cite{60} as on unprotected implementation. This method is very simple and has much lower resource overhead comparing with the secret sharing method.

The leakages of $\theta_{\text{plane}}$ exist because of the compression step of $\theta$, which is implemented using combinational circuits. Although all five steps are executed in one clock cycle and there is no register activity directly related the result of $\theta$ compression operations, the combinational circuits activities still leak information of $\theta_{\text{plane}}$. We modified the synthesis settings to see how they affect the side-channel leakages. We find that for different synthesis settings, the amount of side-channel leakages is very different. For example, for the synthesis with optimization mode ‘\textit{timing}’, the correlation results are as following:

\[
\begin{align*}
\text{correlation}(HW(R_1), \text{power}) &= 0.5 \\
\text{correlation}(HW(\theta_{\text{plane}}), \text{power}) &= 0.09
\end{align*}
\]

For the synthesis with optimization mode ‘\textit{area}’, the correlation results are as following:

\[
\begin{align*}
\text{correlation}(HW(R_1), \text{power}) &= 0.15 \\
\text{correlation}(HW(\theta_{\text{plane}}), \text{power}) &= 0.17
\end{align*}
\]

Thus, designers should try different settings to make the best choice in design phase to minimize the side-channel leakages.

### 3.4.4 Side-Channel Security of Keccak

Our previous results show that unprotected hardware implementations of MAC-Keccak are vulnerable, and attackers can use the leakage of $R_1$ to recover all the key bits. However, compared to other cryptographic algorithms, MAC-Keccak is actually much more resilient to side-channel attacks. For example, previous papers showed that for unprotected AES, the attackers need only several thousands of traces to recover the key bytes \cite{61}. For previous hash functions, like HMAC-based SHA-2 function, only 4000 traces are needed for successful side-channel attacks \cite{62}. While for unprotected MAC-Keccak, we need 35,000 traces to get the five key bits operation results when attacking one row. When the key length is 320 bits, the attacker has to attack 320 rows to get the entire key, which is much more complex than previous cryptographic algorithms like AES and DES, where the number of attack iterations would be just 16 or 6.

In addition, previous block ciphers and hash functions based on them only have limited flexibility of key length, like AES-128 and AES-256. The attack methods for different key lengths are similar,
CHAPTER 3. SIDE-CHANNEL POWER ANALYSIS OF SHA-3

using the divide-and-conquer approach. However, for MAC-Keccak, the key length is variable and the security level can be improved simply by increasing the key length, i.e., the difficulty for side-channel analysis will increase drastically as well.

3.5 Summary

In this work, we explore multiple side-channel leakage models of MAC-Keccak and implement CPA based on these models. Results show that MAC-Keccak has strong side-channel leakages and these leakages can be used by attackers to extract the secret key.
Chapter 4

Protect SHA-3 Against Side-Channel Power Analysis

4.1 Introduction and Motivation

Side-channel analysis (SCA) has been an effective and practical attack on many critical embedded systems that employ cryptographic algorithms for security [13]. SCA exploits the correlation between physical leakages of a crypto system and its secret key-dependent intermediate variables to retrieve the key. Various countermeasures have been presented to protect embedded systems from side-channel analysis. Secret sharing (random masking) [30] introduces random numbers into the system to mask the secret key. It requires algorithm-specific modifications together with expertise knowledge in side-channel attacks, and normally incurs large implementation overhead as each share needs one copy of the original algorithm.

Other countermeasures such as random delay [43] and shuffling [44] have also been studied. These methods spread the side-channel leakage (correlated to an intermediate variable) from a single time point onto multiple points so as to decrease the leakage. In [44] [45], random permutations are applied on AES operations to resist first-order differential power analysis (DPA), based on the fact that AES is a block cipher where the 16 operations on different key bytes and state bytes in each round are independent. In [63], a simplified version of shuffling, Random Start Index (RSI), is presented. RSI is easier to implement but can be significantly weaker than the random permutation method.

Previous works show that the structures of crypto algorithms can also be used by designers to improve the security and reliability of crypto systems [64] [53]. This kind of method makes use of the properties of ciphers to fight against various attacks.
CHAPTER 4. PROTECT SHA-3 AGAINST SIDE-CHANNEL POWER ANALYSIS

To protect Keccak against side-channel analysis, the designers proposed to hide the leakages using secret sharing in [52]. This method introduces random numbers to mask the secret key bits and can effectively protect the systems against side-channel power analysis. The deficiency of this method is that it has very high resource consumption overhead. For software implementations, two-share masking will cause two times resource consumption, while three times resource consumption will be introduced for hardware implementations with three-share masking.

In this chapter, we present two methods to protect SHA-3 against side-channel attacks:

- In Section 4.2, we propose a methodology to analyze the source code of crypto algorithms to automatically detect the dependence of statements. We then devise an algorithm to implement shuffling automatically at the source code level.

- In Section 4.3, we present a protection scheme based on rotation invariance property for SHA-3 smart card implementations [65]. We will show that besides fault injection attacks, this property can also be used to protect SHA-3 against power analysis.

4.2 Side-Channel Leakages Hidding Using Shuffling

4.2.1 Introduction and Motivation

Compared to masking, shuffling does not require modifications of the algorithm. It is an algorithm-agnostic implementation and can possibly be automated for any cryptographic algorithms. What’s more, it can be easily implemented after other countermeasures as an add-on protection for cryptographic systems. However, manual implementation of shuffling still requires knowledge of the specific algorithm and may not fully exploit the independence between operations in complex algorithms. Recent works [28, 66, 67, 68, 69] indicate a nascent trend towards automating the application of countermeasures against SCA to increase the security of systems. They have focused on masking AES, including automatic instruction sensitivity quantification and local random precharging [28], a general code morphing engine design with alternative code segments that mitigate power leakage [66], compiler assisted masking implementation [67], and automatic security evaluation and verification [68, 69]. However, to the best of our knowledge, there is no automation work for operation shuffling/permutation yet.

In this work, we propose a methodology to analyze the source code of crypto algorithms to automatically detect the dependence of statements. We then devise an algorithm to implement shuffling automatically at the source code level. We start from the high level, source code, as shuffling statements at this level are both algorithm and platform independent. We test our algorithm on SHA3
standard. The experimental results show that our algorithm automatically identifies independence among operations and implements shuffling, which significantly improves the resilience of crypto software against power analysis attacks. The main contribution of the work lies in a framework of source code transformation to randomize operations without any knowledge of the underlying systems, which is generally applicable to other cryptographic algorithms as well.

The rest of this section is organized as follows. Section 4.2.2 introduces some preliminaries on shuffling. In Section 4.2.3, we present our algorithm to extract the dependence among software statements and propose a methodology for automatic shuffling implementation. In Section 4.2.4, we demonstrate correlation power analysis results on Keccak protected with shuffling based on our algorithm.

4.2.2 Preliminaries

4.2.2.1 Countermeasure to Power Analysis Attacks - Shuffling

Shuffling is an effective countermeasure to mitigate the vulnerability of cryptographic systems against power analysis. As the leakage comes from intermediate variables produced by operations, if there are $n_p$ possible time locations of a leaky operation, i.e., the shuffling space is $n_p$, the leakage of such intermediate variable is randomly spread onto $n_p$ time points. The corresponding correlation and signal-noise-ratio (SNR) for power analysis attacks will be effectively decreased to $1/n_p$ [44, 45].

When one piece of code $C$ is composed of $N$ independent segments with some of them producing key-dependent intermediate variables, $C = \{C_0, C_1, \cdots, C_{N-1}\}$, they can be executed in any order without changing the algorithm functionality. Their execution order is determined by an array, $\text{Order}$, which is a random permutation of $\{0, 1, \cdots N-1\}$.

Shuffling implementation includes two critical steps, extraction of statement dependence and generation of the random execution order array $\text{Order}$. Fisher-Yates algorithm [70] is an efficient algorithm for generating such an unbiased array with linear computational complexity $O(n)$. It is an unbiased algorithm, which means the resulting $\text{Order}$ array will be fully random [71]. Thus we adopt this algorithm for random order array generation and expect it to impose small execution overhead in large crypto systems.

4.2.2.2 Data Dependence and Operation Consistency

For shuffling, the most important step is to extract the dependence of statements to identify the space for shuffling. We first make some definitions for software source code:
We denote a piece of code $C$ as a set of sequential statements $A = \{A[0], A[1], \cdots, A[N-1]\}$. Here $N$ is the number of statements, called code size. We assume the code is branch-free and loop-free, i.e., consisting of only sequential static single assignments (SSA).

For each statement $A[i]$, it includes the variable set $V[i] = \{v_i[0], v_i[1], \cdots\}$, with $v_i[0]$ being its output and other variables being its inputs. Statements may have different numbers of input variables.

For statements $A[i]$ and $A[j]$, with $i > j$, if there is dependence between the two statements, we say that $A[i]$ depends on $A[j]$, denoted as $A[i] \Rightarrow A[j]$.

There are three types of dependence between two sequential statements $A[i]$ and $A[j]$, $i > j$, defined in [72]:

- Read after write (RAW): if $A[i]$ uses a variable that was defined by $A[j]$, their execution order has to be preserved;
- Write after read (WAR): if $A[i]$ redefines a variable that was used by $A[j]$, their execution order has to be preserved;
- Write after write (WAW): if $A[i]$ redefines a variable that was defined by $A[j]$, their execution order has to be preserved. Some compilers may optimize this WAW dependence if no usage of the variable is found between the two statements. At the source code level, we keep such dependence.

Shuffling, although aims at changing the execution order of statements, should preserve their inherent dependence. We define this as the rule of operation consistency.

**Example 4.2.1** Here we give a piece of code as an example, and denote it as a sequence of statements as following:

\[
\begin{align*}
\begin{cases}
a = b + c + d \\
e = a + f \\
a = g + h
\end{cases}
\end{align*}
\]
\[
\begin{align*}
A[0] : & \quad V[0] = \{a, b, c, d\} \\
A[1] : & \quad V[1] = \{e, a, f\} \\
\end{align*}
\]

CHAPTER 4. PROTECT SHA-3 AGAINST SIDE-CHANNEL POWER ANALYSIS

4.2.3 Methodology and Algorithm

4.2.3.1 Statement Dependence Extraction

The simple Example 4.2.1 shows that the dependence determines the constraints for operation shuffling and should be extracted first. For a piece of code with \( N \) statements, we use an \( N \times N \) matrix \( M \) to denote the dependence of these \( N \) statements, and it should only be lower-triangular according to the definition of data dependencies. With the row index \( i \) and the column index \( j \), \( M[i][j] = 1 \) means that \( A[i] \Rightarrow A[j] \), for \( 0 \leq i, j \leq N - 1, j < i \). The algorithm for dependency matrix generation is shown in Algorithm 1.

**Algorithm 1** Dependence extraction

**Input:** The source code of the crypto algorithm  
**Output:** Lower triangular dependency matrix \( M \) for the algorithm

1: \( N \leftarrow \) number of statements \( |A| \)
2: Generate an \( N \times N \) matrix \( M \), initialized at zero
3: for \( i = 1 \rightarrow N - 1 \) do
4:   for \( j = 0 \rightarrow i - 1 \) do
5:     if \( V[j][0] \in V[i] \) then \( \triangleright A[i] \Rightarrow A[j] \), WAW and RAW
6:       \( M[i][j] \leftarrow 1 \)
7:     end if
8:   if \( V[i][0] \in V[j] \) then \( \triangleright A[i] \Rightarrow A[j] \), WAW and WAR
9:     \( M[i][j] \leftarrow 1 \)
10: end if
11: end for
12: end for

Using Algorithm 1 we can extract the dependence among all the statements. This dependency matrix helps designers automate shuffling. Finding a group of statements for shuffling is to find a sub-matrix in \( M \) along the main diagonal with all its elements zero, i.e., the corresponding statements are mutually independent.

We use the Keccak source code (Keccak-simple32BI [73]) as an example to apply Algorithm 1. There are 344 statements (SSAs) for the first two rounds (in one loop), and thus the dependency matrix size is \( 344 \times 344 \). Figure 4.1 shows a \( 36 \times 36 \) dependency sub-matrix for the first 36 statements.

For the \( 36 \times 36 \) matrix, the square \((i, j)\) in black color means \( M[i][j] = 1 \), i.e., \( A[i] \Rightarrow A[j] \). For these 36 statements, it is clear to see that \( A[0] \) to \( A[9] \) are mutually independent, and thus these 10 statements can be randomly permuted. This is also true for \( A[10] \) to \( A[19] \). Note that
the sub-matrix does not need to be composed by consecutive rows and columns. For example, \{A [21], A [23], A [25], A [27], A [29]\} and \{A [20], A [22], A [24], A [26], A [28]\} can also be permuted as their corresponding sub-matrices are both zero. We also analyze the AES code from \[74\] using our algorithm and similar dependence result is obtained. For larger cryptographic software, large \(N\) and complex dependence require automatic methods for dependence extraction and operation shuffling to be designed.

So far we have shown that shuffling is done on operations within basic blocks, i.e., relying on the data flow of basic blocks and exploiting the dependence. Shuffling can also be applied at the higher control flow level between basic blocks if the basic blocks are independent from each other. For example, the MixColumns step of an AES round consists of four basic blocks with each basic block working on one column of the AES state. These four blocks are mutually independent and can be permuted. However, the space for shuffling at the basic block level may be limited. To fully explore the shuffling space, we can apply multi-level permutations, where independent statements inside each basic block are permuted and meanwhile the serialization of basic block execution is also randomized (permuted).
CHAPTER 4. PROTECT SHA-3 AGAINST SIDE-CHANNEL POWER ANALYSIS

4.2.3.2 Automatic Shuffling

With the dependence of statements extracted, we separate all the statements into different levels, such that the statements at the same level are mutually independent and can be permuted randomly without violating the rule of operation consistency. Statements at different levels have data dependence and should preserve a certain execution order: statements at the lower level should be executed before the higher level ones.

Algorithm 2 shows the procedure to separate all statements into different levels. It is actually running a reverse depth-first search on the data flow graphs of the program, and assigns the statements (nodes in the DFGs) to different depths.

\begin{algorithm}
\caption{Execution level extraction}
\begin{algorithmic}[1]
\Input{The $N \times N$ dependency matrix $M$}
\Output{Execution level array $L[N]$}
1: Initialize an zero array $L[N]$
2: \For{$i = 1 \rightarrow N - 1$}
3: \hspace{1em} $level = 0$, $newlevel = 0$
4: \For{$j = 0 \rightarrow i - 1$}
5: \hspace{2em} \If{$M[i][j] = 1$}
6: \hspace{3em} $newlevel = L[j] + 1$
7: \hspace{3em} \If{$newlevel > level$}
8: \hspace{4em} $level = newlevel$
9: \hspace{3em} \EndIf
10: \hspace{2em} \EndIf
11: \EndFor
12: $L[i] = level$
13: \EndFor
\end{algorithmic}
\end{algorithm}

We apply our algorithms on AES and Keccak, and the results show that our algorithms can effectively extract the dependence of statements and explore the shuffling space. For example, for the loop-free AES implementation, 55 statements are separated into 8 levels and the maximum number of statements at a level is 16; for Keccak (Keccak-simple32BI), the 344 statements can be separated into 26 levels, with the maximum number of statements at a level as 50. When the number of statements in one level is too small, some dummy statements can be added into this level to increase the shuffling space but with a moderate performance penalty. Note here the independent statements may not all be leaky operations, i.e., producing key-dependent intermediate variables. They just represent data flow dependence. Only permutation of the leaky operations would decrease the side-channel leakage.
4.2.4 Experimental Results and Evaluation

4.2.4.1 The Tool for Source Code Transformation

Our algorithms above operate on statements at the source code level without the need to understand the target cryptographic algorithm. Therefore, tools based on the proposed algorithms only need some lexical analysis, but do not need the complex semantic analysis that compilers have. In this work, we develop a source code transformation tool based on our algorithms using Python. The structure of our tool is shown in Figure 4.2.

Figure 4.2: Automatic source code transformation with shuffling

Our automatic source code transformation tool consists of the following steps for shuffling:

1. **Lexical analysis**: Read and parse the loop-free source code, extract all the variables in each statement. Lexical analysis is the first step for compilation process and there exist mature tools that can be applied, for example, we use FLEX [75] for test here.

2. **Dependence analysis**: Use Algorithm 1 shown in Section 4.2.3 to analyze the dependence among the statements in source code and generate the dependency matrix.

3. **Space exploration**: Use Algorithms 2 to explore the space for statement shuffling.

4. **Code generation**: Based on the shuffling space exploration results, embed a run-time permutation engine that contains code for generating the random execution order array, and generate a report about dependence and shuffling space at the same time.

As described before, only part of the independent statements leak secret information and permutation on them would reduce the leakage. If some knowledge about the secret information can be used in the “Random Shuffling” step, the implementation will be more efficient. For example, as most of the previous attacks on Keccak software implementation focus on \( \theta \) step, we can apply
the code transformation to this part of source code only and the implementation can be effective in reducing the leakage and still be efficient.

The designers only need to run this tool one time at the design stage. The generated code will contain a module that generates the random order array at run-time. The computational complexity of Algorithm 1 and 2 are both $O(n^2)$. For one round of Keccak which contains 344 statements, our Python implementation needs less than one second to finish all the work including dependence extraction and shuffling generation. Note that the transformations done by our tool will be orthogonal to normal compilation processes and will not be changed by later compilation. At runtime, a random array Order will be generated each time before the cryptographic execution, and the computational complexity of random array generation algorithm is $O(n)$. Details about the overhead on a target platform will be discussed in Section 4.2.4.3.

4.2.4.2 Side-Channel Leakage Reduction

To evaluate the effect of our automatic shuffling application, we run our algorithm on the source code of Keccak, Keccak-simple32Bl, and generate an automatically shuffled version. We implement both the original and shuffled implementations on a 32-bit Microblaze processor running on an SASEBO-GII board with a Virtex-5 FPGA [48].

We use similar setting for Keccak as previous papers [11], [76], [77]. Without loss of generality, we assume the key size is 320 bits. For software implementations, there are many intermediate variables that can be used for side-channel analysis. We also attack the first step of $\theta$ which compresses every five bits (including one key bit) into one bit. With our dependence detection algorithm, statements of $\theta_1$ are all separated into the first level which contains 10 statements. Meanwhile, the results of $\theta_1$ are used in step $\theta_2$ and they are separated to second level which also contains 10 statements. We only permute the two groups with each group containing 10 leaky statements. Thus each of the original two leakage points is spread onto 10 points and we expect to see a 10 times decrease of the leakage. For power analysis, we collect power traces for the two implementations using a LeCroy WaveRunner 640Zi oscilloscope. We first run CPA in time domain on the two MAC-Keccak implementations. The leakage model is Hamming weight of eight bits of $\theta_1$, and the correlation results are shown in Figure 4.3.

Figure 4.3(a) shows the correlation between the Hamming weight of eight bits of $\theta_1$ output and the power consumption for the unprotected Keccak implementation based on 1,000 power traces. With just 1,000 traces for the unprotected MAC-Keccak, the peak correlations at the two leakage points (0.55 and 0.27) stand out from the noisy correlations at other non-leaky time points (varying between 0.1 and -0.1). Figure 4.3(b) shows the correlation result of the shuffled implementation with
15,000 power traces. The peak correlation is much smaller, which implies that more power traces are required to reduce the variance of other noisy correlations. For the shuffled implementation, the two leakage points are spread onto twenty time points clearly. The average correlation of these points is close to 0.055, which is about \( \frac{1}{10} \) of 0.55, the original correlation. The largest correlation is only about 0.09. With the decrease of correlation (SNR), by the formula in \([78]\), the number of traces needed for protected implementation, to achieve the same success rate, will be 36 times of the number for unprotected implementation.

Note that Figure 4.3 only shows the leakage reduction of one key byte. When groups of statements are permuted, since different statements involve different key bytes, the leakages on other key bytes are also reduced significantly. This will make the full key recovery much harder.

Previous works show that while some countermeasures like random delay can reduce the leakages in time domain, they are ineffective in frequency domain \([79][80]\). Thus we also check if
our operation shuffling algorithm can improve the side-channel security of embedded crypto systems in frequency domain. We find that in frequency domain, the SNR is very small and thus we use Hamming weight of 32 bits instead of 8 bits together for correlation analysis. For the unprotected MAC-Keccak implementation, the highest correlation at certain frequency is very clear and reaches 0.2. While for shuffled implementation, the largest correlation is very unclear and only about 0.05 instead. This result shows that shuffling can also effectively improve the resilience of the crypto system to side-channel attacks in frequency domain.

4.2.4.3 Overhead Evaluation

Evaluating the execution time and resource overhead is important for countermeasure design because of the limited resources in embedded systems. The comparison of binary file size and clock cycles for three different MAC-Keccak implementations on Microblaze is shown in Table 4.1. Note here the secret sharing scheme is a manual two-share masking implementation and one random number is involved for masking [52]. The shuffling implementation is the one that our automation tool generates.

Table 4.1: Comparison of resources and execution cycles among three schemes

<table>
<thead>
<tr>
<th>Implementations</th>
<th>File size (Byte)</th>
<th>Clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original [73]</td>
<td>31040</td>
<td>1670</td>
</tr>
<tr>
<td>Shuffling</td>
<td>40128</td>
<td>2580</td>
</tr>
<tr>
<td>Secret Sharing  [52]</td>
<td>69272</td>
<td>6780</td>
</tr>
</tbody>
</table>

All three implementations are implemented in plain C language on the same platform with the same compilation and measurement settings. The second column of Table 4.1 gives the binary file size of each implementation and the third column shows the execution clock cycles. We can see that shuffling incurs 29.3% memory overhead and the execution time is 1.54X compared to the unprotected version. For secret sharing, the memory overhead is 123% and the execution time is 4.06X. This is because secret sharing involves share generation and de-masking and this needs more resources than other schemes.

From the overhead results and the analysis above, we can see that our algorithm is effective to automate shuffling implementations to decrease the side-channel leakage. While some other countermeasures such as masking involve modification of the source code which is labor-intensive and requires a thorough understanding of the crypto algorithm, our scheme is easy to design and
requires no knowledge of the target algorithm. Our algorithms and tool can be applied to different cryptographic algorithms to help improve the system security.

4.3 An Improvement of Both Security and Reliability of Keccak Software Implementations

4.3.1 Introduction and Motivation

To protect Keccak against fault injection attacks, the authors in \[65\] proposed to protect Keccak on FPGAs using the round rotation invariance property. The problem of these countermeasures is that they still have strong side-channel leakages and thus vulnerable to side-channel analysis attacks. In this work, we propose to use round rotation invariance property of Keccak sponge function to protect Keccak implementations on smart card platform against both side-channel attacks and fault injection attacks. The advantage of this scheme is that it can protect Keccak against both fault attacks and side-channel attacks, thus it will have much lower resource overhead than combining two protection methods together directly. Meanwhile, this scheme requires only minor modification of Keccak sponge function, thus it can be used together with with other protection schemes for even higher security and reliability level.

In this work, we focus on the implementations for resource restricted platform, smart card, and test our scheme based on the implementation of AVR-Crypto-Lib \[74\]. Meanwhile, we run fault injection simulation at algorithm level for accurate error coverage results, which show that the proposed scheme can significantly reduce the side-channel leakages of Keccak implementations and improve its reliability against random errors and injected faults.

The rest of this section is organized as follows. In Section 4.3.2, the proposed scheme will be introduced, then attacks and simulation results will be given in Section 4.3.3.

4.3.2 Round Rotation Invariance Based Scheme

Previous papers proposed to used the structures of crypto algorithm for security and reliability enhancement. For example, in \[64, 63\], the authors propose to use the invariance of AES for error detection. Such schemes make use of the structures of crypto algorithms for protections, thus they are usually efficient and effective. In this section, we introduce the round rotation invariance property of Keccak sponge function, and how to use it for both side-channel leakage reduction and fault detection on smart card platform.
4.3.2.1 Invariance of Keccak Permutation Function

Keccak reference manual explained that the mapping $\iota$ is added to disrupt the symmetry of Keccak operations (translation-invariant in the $z$ direction), thus to avoid slide attacks $[1]$. Operations other than $\iota$ are translation-invariant in the $z$ direction, which means the input can be rotated in $z$ direction and then rotated back with the result unchanged:

$$g(in) = ROT^{-1}(g(ROT(in, \alpha), \alpha)). \quad (4.1)$$

In (4.1), $ROT$ stands for the round rotation at $z$ direction and $\alpha$ is a random number ($0 \leq \alpha \leq 63$), $g$ stands for Keccak operations $\theta$, $\rho$, $\pi$, and $\chi$. While (4.1) holds for other four operations, it it not true for $\iota$ operation, because $\iota$ involves constant numbers $\iota_c = \{\iota_c[0], \iota_c[1], \cdots, \iota_c[23]\}$ besides the input $in$ for each round. Thus we can rotate $\iota_c$ also $\alpha$ bits for operation invariance:

$$\iota(in, \iota_c) = ROT^{-1}(\iota(ROT(in, \alpha), ROT(\iota_c, \alpha)), \alpha). \quad (4.2)$$

Which means that we can denote Keccak sponge function over the first round input $S_0$ and $\iota_c$ as following:

$$f(in, \iota_c) = ROT^{-1}(f(ROT(in, \alpha), ROT(\iota_c, \alpha)), \alpha). \quad (4.3)$$

In this work, we can also denote the right side of (4.3) as $f'(in, \alpha)$. It means the modified Keccak function has $in$ and $\alpha$ as input, and round rotates each lane of $in$ and $\iota_c \alpha$ bits for Keccak operations, and rotate the final results back with the result unchanged. We give the details as follows.

For the first round input $S_0 = \{L_{i,j}\}, i, j \in \{0, 1, 2, 3, 4\}$, rotate each lane $L_{i,j}$ $\alpha$ bits at $z$ direction:

$$L'_{i,j} = ROT(L_{i,j}, \alpha), i, j \in \{0, 1, 2, 3, 4\}, 0 \leq \alpha \leq 63. \quad (4.4)$$

Denote the new input composed of these rotated lanes as $S'_0$:

$$S'_0 = \{L'_{i,j}\}, i, j \in \{0, 1, 2, 3, 4\}. \quad (4.5)$$

We use $\iota'_c$ to stand for the rotated $\iota_c$ (the set of 24 constant numbers for $\iota$ operations in 24 rounds):

$$\iota'_c[i] = ROT(\iota_c[i], \alpha), i \in \{0, 1, \cdots, 23\}. \quad (4.6)$$

The Keccak operation results ($S_{24}$ and $S'_{24}$) based on these two input ($S_0, \iota_c$ and $S'_0, \iota'_c$) are as following:

$$\begin{align*}
S_{24} &= f(S_0, \iota_c) = f'(S_0, 0) \\
S'_{24} &= f(S'_0, \iota'_c) = f'(S_0, \alpha).
\end{align*} \quad (4.7)$$
Then the following equation holds:

\[ S_{24} = ROT^{-1}(S'_{24}, \alpha), \]  

(4.8)

which means that Keccak result over the rotated input lanes and \( \iota_c \) constant numbers can be rotated back to get the original result \(^65\). In the following section, we will demonstrate how to use this round rotation invariance property to reduce the side-channel leakages while increase reliability of Keccak implementations on smart card platform.

### 4.3.2.2 Invariance-Based Protection Scheme

The invariance property of Keccak permutation function can be used for side-channel leakage reduction because it can distribute leakages from one point to multiple points. Meanwhile, this property can also be used for fault detection by comparing the results of two rotated Keccak implementations with different \( \alpha \). We devise the structure in Figure 4.4 for Keccak implementation on smart card for both side-channel leakages reduction and reliability enhancement.

![Figure 4.4: The proposed invariance-based protection scheme](image-url)
CHAPTER 4. PROTECT SHA-3 AGAINST SIDE-CHANNEL POWER ANALYSIS

It works like this, when the Keccak implementation receives the input message in, it starts working by generating one random number $\alpha_1$, and use this random number for the rotated Keccak computation shown in Section 4.3.2.1. Then another random number $\alpha_2$ will be generated and used for rotated Keccak to generate $O_2$. The output $O_1$ and $O_2$ are compared for fault detection.

How it reduces side-channel leakages

For Keccak implementations on smart card, if we randomly rotate each input lanes before Keccak operations, the leakage of one key bit will be distributed from one time point to multiple time points. We assume 64 bits in one lane ($L[0 : 63]$) are rotated by $\alpha$ bits ($\alpha \geq 1$) at $z$ direction, and the rotated lane bits are $\{L[64 - \alpha : 63] L[0 : 63 - \alpha]\}$. For 8-bit architectures (either microprocessor or FPGA), only 8 bits in each lane will be processed in one clock cycle. Take eight key bits at position $L[0 : 7]$ as example, after round rotation, these bits will be shifted $\alpha$ bits along $z$ axis to be $L[\alpha : \alpha + 7]$ ($\alpha + 7$ is modular 64). Then the leakages of $L[0 : 7]$ will be distributed from one clock cycle to multiple clock cycles. Thus the leakages of these bits will be significantly reduced.

For differential power analysis (DPA), one bit may be moved into eight different bytes. For correlation power analysis (CPA), previous papers always attack eight bits in one lane together for higher signal-to-noise ratio (SNR). Thus:

- If $\alpha$ is a time of eight, these eight bits are still in one byte, the leakage of these eight key bits are distributed from one time point to another time point.

- If $\alpha$ is not a time of eight, these eight bits are distributed into two adjacent bytes, and leakages of other key bits in the same bytes will be working like noise and help to further reduce their leakages.

To evaluate the resilience of proposed scheme against side-channel analysis, we implement original Keccak and Keccak with the proposed scheme on a SASEBO-W board, which is designed specifically for side-channel evaluations [81]. Keccak has been implemented in 64-bit, 32-bit, 16-bit and 8-bit structures, examples are source code provided Keccak official site [82]. For compact platforms such as 8-bit smart cards, example implementations are like [83] (compact IC design) and AVR-Crypto-Lib [74] (for smart card software design). In this work, we refer to AVR-Crypto-Lib [74] for implementation, and run side-channel analysis and fault injection on it.

How it improves the reliability

For error detection methods like duplication or redundancy, they are incapable of detecting those faults that cause the same errors in both copies [64]. For such redundancy based schemes, attackers who have the ability to inject the same faults at the same positions of both copies will be able to bypass the error detection modules. But for the proposed scheme in Figure 4.4 the positions
CHAPTER 4. PROTECT SHA-3 AGAINST SIDE-CHANNEL POWER ANALYSIS

of message bytes will be different for different random number $\alpha$. Thus it will be impossible for attacker to inject faults at the same positions if he has no knowledge of two random numbers $\alpha_0$ and $\alpha_1$. To evaluate the error detection coverage of the proposed scheme, we run fault injection simulation at algorithm level, and details and results will be given in Section 4.3.3.

4.3.2.3 Advantages of the Proposed Scheme

As discussed in previous sections, the proposed scheme can significantly reduce side-channel leakages of Keccak, and also improve reliability of the Keccak implementations on smart card platform. Comparing with combining two different countermeasures together directly, the proposed scheme is resource friendly. This scheme has the following advantages:

- Comparing with countermeasures like secret sharing, the proposed scheme is easy to implement. The proposed scheme mainly operates on the input message lanes, and only needs to change the $r_e$ table for different random rotation number $\alpha$, the sponge function needs no extra modifications.

- It can be easily combined with other countermeasures. The proposed scheme does not change the operations of sponge function, it can be easily combined with other countermeasures such as secret sharing, random permutation and random delay to improve the side-channel security. Error detection methods such as parity checking can be directly added on the proposed scheme to improve the error coverage for higher reliability.

- It is resource friendly and can be efficiently implemented on resource restricted applications. The proposed scheme is resistant to both fault attacks and side-channel attacks at the same time. Comparing with combining two different schemes together, the proposed scheme is resource efficient.

4.3.3 Side-Channel Power Analysis and Fault Injection Simulation of the Proposed Scheme

In this section, we attack the original Keccak implementation and the implementation protected with the proposed scheme. For power analysis, we use a LeCroy WaveRunner 640Zi oscilloscope to sample all power traces of the implementations on SASEBO-W board, and we use CPA attack as an example to demonstrate the results. Meanwhile, we simulate the error detection by injecting fault at algorithm level.
CHAPTER 4. PROTECT SHA-3 AGAINST SIDE-CHANNEL POWER ANALYSIS

Figure 4.5: Side-channel power analysis results: the left side are the results of attacking the original implementation, the right side are the results of attacking the implementation protected with the proposed scheme.
4.3.3.1 Side-Channel Analysis Results of Protected and Unprotected Implementations

For the original implementation, we sample 500 traces and use CPA to attack the first step of \( \theta \) operation, the same as \([12,11,76]\). We attack eight bits each time for higher SNR, and use the result for the first eight bits of the first \( \theta_1 \) lane (\( \theta_1(0, [0 : 7]) \)) in this work. The correlation between the power consumption and \( HW(\theta_1(0, [0 : 7])) \) is shown in Figure 4.5(a). Figure 4.5(a) shows that the correlation is reaching 0.7 (or negative -0.7), which is very strong for cryptographic systems. This strong leakage can be easily used by attacker to retrieve key bits information.

Figure 4.5(b) shows the correlation between different key assumptions and power consumption. The blackened trace is for the correct key and the gray traces are for false keys. It shows that the correlation between the correct key guess will stand out of the key guesses very quickly. Figure 4.5(c) shows the success rate of CPA attacks. The attackers only need about 200 traces to recover the key bits with success rate 100%. Thus, without protection, the Keccak implementations on smart card are vulnerable to side-channel attacks, and a very small number of power traces are enough to successfully extract all the key bits.

For the proposed scheme, the leakages at one point are distributed to multiple points, and we anticipate that the proposed scheme can improve the side-channel security of Keccak implementations. For the proposed implementation, we sample 5,000 traces and run CPA on them, and show the correlation result in Figure 4.5(d). It shows that for the implementation protected with the proposed countermeasure, the leakage decreases significantly. Comparing with the leakage of the original implementation, the correlation of the proposed scheme is much smaller, almost covered by noise in the system. Thus it’s anticipated that the protected implementation will be much more difficult to conquer than the original implementation.

Figure 4.5(e) shows the correlation between different key assumptions and power consumption. It shows that the right key’s correlation is significantly reduced and it stands out of the key guesses very slowly. For the original implementation shown in Figure 4.5(c), it only needs more than 100 traces for the correct key guess to stand out of the wrong key guesses. For the implementation protected with the proposed scheme, the leakages are reduced significantly and it needs many more traces for the right key to stand out of the wrong key guesses. Success rate result in Figure 4.5(f) shows that attackers need about 4,000 traces to recover the key bits for this implementation, 20 times more than the original implementations.

Above results show that the proposed scheme can effectively protect Keccak implementation against side-channel analysis. Keccak implementation with the proposed countermeasure is much more difficult to conquer than the original implementation. Besides the reduction of side-channel leakages, the round rotation invariance of Keccak can also be used to protect Keccak against random
CHAPTER 4. PROTECT SHA-3 AGAINST SIDE-CHANNEL POWER ANALYSIS

errors and injected faults, and simulation results will be given in Section 4.3.3.2.

4.3.3.2 Fault Injection Simulation Result

In this work, we do not differentiate faults and errors, which means that we only care about faults which generate errors at the output. We simulate fault injection and get the error coverage of the proposed scheme at algorithm level. In this work, we assume two attacker models, the weak attacker model and strong attacker model:

- **Weak** attacker model: the attackers can precisely inject faults at a given clock cycle, but have no control of the injected faults.
- **Strong** attacker model: the attackers can precisely inject faults at a given clock cycle, and have fully control of the injected faults.

For fault injection simulation, we target at four implementations, (a) the implementation protected with our proposed scheme, (c) the implementation protected with secret sharing, (c) the implementation protected with double copy redundancy and (d) implementation with parity checking error detection.

For both attacker models, we assume the attacker can distort one byte at a specific clock cycle, and he can inject either single or multiple byte faults into the system. For single byte fault model simulation, we randomly inject random faults (0-255) into one random byte of the input (200 bytes) of both copies. If the results of these two copies are equal while different from the original result, we think the injected errors are not detected. For multiple bytes fault injection, we inject random faults into from one to five bytes of both copies. Besides the fault positions and faults injected, the number of faults are also randomly generated.

For each implementation, we run the fault injection simulation for \( \sim 2 \times 10^8 \) times for both single fault and multiple faults models. Simulation results show that for single fault model, the proposed scheme can detect 99.998% of the injected faults. For multiple faults model, the proposed scheme can detect 99.999975% of the injected faults. Thus the proposed scheme has a very high fault coverage for Keccak implementations on smart card platform. We summarize the single byte fault injection results of different protection schemes in Table 4.2.

From the fault injection simulation result, we can see that the proposed scheme can efficiently protect different applications of Keccak function, such as integrity checking, hashing, stream encryption, etc. While advanced attacker can bypass some simple error detection methods, it will be almost impossible for him to bypass the error detection module in the proposed implementation without know ledges of the random numbers \( \alpha_1 \) and \( \alpha_2 \).
CHAPTER 4. PROTECT SHA-3 AGAINST SIDE-CHANNEL POWER ANALYSIS

Table 4.2: Comparison of countermeasures

<table>
<thead>
<tr>
<th></th>
<th>Leakage Reduction</th>
<th>Error coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Weak II</td>
</tr>
<tr>
<td>Proposed scheme</td>
<td>85%</td>
<td>99.99%</td>
</tr>
<tr>
<td>Secret sharing</td>
<td>100%</td>
<td>0%</td>
</tr>
<tr>
<td>Double redundancy</td>
<td>0%</td>
<td>99.602%</td>
</tr>
<tr>
<td>Parity checking</td>
<td>0%</td>
<td>99.606%</td>
</tr>
</tbody>
</table>

I First order side-channel leakage reduction, the ratio of correlation;  
II Weak fault injection model, attackers can inject faults into the same byte, but cannot control the injected fault values;  
III Strong fault injection model, attackers have fully control of the fault injection positions and values.

For double copy redundancy and parity checking, although they can detect most of the injected faults under weak attacker model, they will be easily bypassed by attackers under strong attacker model. What’s more, such simple error detection schemes have no resilience to side-channel power analysis at all. Theoretically, secret sharing scheme can delete all the side-channel power leakages, but it has no effect against fault injection attacks. Thus, comparing with secret sharing, the advantage of the proposed scheme is that it can also improve the reliability of Keccak implementations.

4.4 Summary

In this chapter, we present the methods to protect SHA-3 implementations against side-channel power/EM attacks. Both methods make use of the structure properties of SHA-3 to mitigate side-channel leakages. These two methods are easy to implement and can be used together with other protection methods. Impelmentation results show that the proposed methods can mitigate side-channel leakages effectively with limited resource overhead.
5.1 Introduction and Motivation

As shown in previous chapter, side-channel attack is a realistic threat to various cryptographic systems, including DES, AES, RSA, SHA-3, etc. [15] [13]. Although various countermeasures have been proposed to protect cryptographic systems against power or EM analysis attacks [26][27][29][30], such countermeasures implementation requires expertise knowledge in side-channel attacks and the target implementations.

Existing manual implementations of countermeasures require deep understanding of the cipher and the target implementation, and expertise knowledge of side-channel attacks. What’s more, the security evaluation is also ad hoc for such manual implementation, as they are error-prone and there lacks proof or guarantee of security [28]. To address these issues, some efforts have been made towards automated protection design against side-channel attacks [28][31][32]. In [28], the authors analyze the leakages of AVR platform automatically and transform the vulnerable instructions into leakage-free code. In [32], SMT solvers are employed to find a perfectly masked implementation for the original program at the source code level. They both belong to masking techniques. However, one is still platform-specific as it works on machine code, and the other works at the fairly high-level source code but still requires to re-design the code.

To automatically implement countermeasures for ciphers in different programming languages and target various platforms, we argue that compiler is the suitable stage and platform. We propose to choose the open LLVM (Low Level Virtual Machine) framework as the platform, and implement the construction of protected implementations as an automated pass at the Intermediate Representation
CHAPTER 5. COMPILER ASSISTED THRESHOLD IMPLEMENTATION DESIGN

(IR) level [84]. Such an automated pass can be used to make changes on the original design at the IR level, thus it is language-agnostic and the generated protected design can be ported to different platforms directly using mature back-ends.

We choose Threshold Implementation (TI), a provable secure countermeasure against power and EM analysis attacks, as the protection scheme in this work. TI is a strict masking scheme, which splits the original sensitive variables into multiple shares such that the attacker would not be able to break the target system unless he has control of all the shares [85, 86]. The side-channel resilience of a TI implementation is guaranteed if a number of design rules are followed. TI has been applied to protect ciphers like DES, AES, SHA-3, etc [87, 88, 89, 53, 90]. In this work, we implement an LLVM middle-end pass to generate TI design for a given cipher, and use mature available back-end passes to generate binary code for the TI implementation for different hardware platforms. This Compiler Assisted Threshold Implementation (CATI) pass is language-agnostic, and does not require any knowledge of the cipher or the target platform.

In this work, we take AES and SHA-3 as example ciphers, and automatically generate TI designs for their widely used implementations directly [91, 92, 82]. We implement the automatically generated TI implementations on a commercial ARM Cortex-M3 processor to evaluate the side-channel resilience. Results show that our scheme can generate TI implementation for given design efficiently, and the generated implementations are secure against power analysis attacks.

The rest of this chapter is as follows. In Section 5.2 we introduce preliminaries of TI and the LLVM framework. In Section 5.3 details of the proposed CATI pass are presented. In Section 5.4 we evaluate the generated TI implementations, in terms of both resource overhead and the side-channel resilience improvement. In Section 5.5 we discuss the results and the future works.

5.2 Preliminaries of Threshold Implementation and LLVM

5.2.1 Notations

Assume a function $F$ has three inputs, $x$, $y$ and $z$. We use $(x, y, z)$ to denote the vector of input. If the input $x$ is split into $s$ additive shares, $x_s$, with $x = \oplus_{i=1}^{s} x_i$, the vector of $s$ shares of $x$ is denoted as: $x = (x_1, x_2, \cdots, x_s)$. The vector of the $s$ shares of $(x, y, z)$ is denoted as $(x, y, z)$. The shares of $F$ function are denoted as $F'$ in this work.

5.2.2 Threshold Implementation

TI is a kind of side-channel attack countermeasure based on secret sharing and multi-party computation. In TI, a variable $x \in F_m^m$ is split into $s$ additive shares $x$. Thus, the knowledge of up to
CHAPTER 5. COMPILER ASSISTED THRESHOLD IMPLEMENTATION DESIGN

$s − 1$ shares is incomplete and does not reveal information of $x$.

In order to implement a TI for a function, $a = F(x, y, z, \cdots)$, in which $a \in F^n_2$, a set of shared functions $F_i$ which together compute the output of $F$ are required. There are three properties each shared function should satisfy:

1. Correctness: $F(x, y, z) = \oplus_i F_i(x, y, z)$ for all $(x, y, z)$ and $(x, y, z)$ if $\oplus_{i=1}^{s} x_i = x, \oplus_{i=1}^{s} y_i = y, \oplus_{i=1}^{s} z_i = z$.

2. Non-completeness: Every function $F_i$ is independent of at least one share of each variable $(x, y, z)$, such that even the attacker has fully control of this $F_i$, he cannot get any information of the sensitive variables.

3. Uniformity: For all $a = (a_1, a_2, \cdots, a_s)$ satisfying $\oplus_{i=1}^{s} a_i = a$, the number of tuples $(x, y, z, \cdots) \in F^{ms}$ for which $F_j(x, y, z, \cdots) = a_j$, $1 \leq j \leq s$, is equal to $2^{(s−1)(m−n)}$ times the number of $(x, y, z) \in F^m$ for which $a = F(x, y, z, \cdots)$. Hence, if $F$ is a permutation on $F^m$, the function $F_i$ together define a permutation on $F^{ms}$. In other words, the sharing preserves the output distribution.

It has been formally proved that if the above properties are preserved for each shared function, the generated implementation will be provable secure. As the number of shares represents randomness and incurs implementation cost, in real implementations there may be limitations on the number of shares. It has also been shown that some operations have no uniform TI solutions for a given constraint on the number of shares. To achieve uniformity, either the number of shares for the input and/or output has to be relaxed (increased), or random numbers can be introduced on intermediate results for re-masking to generate TI implementations [54, 85].

5.2.3 LLVM

The LLVM framework is a collection of modular and reusable compiler and toolchain technologies. LLVM is composed of three parts, the front-end translate software implementations in different programming languages into LLVM IRs, the mid-level passes optimize and improve the IRs, and the back-end code generators transfer IRs into binary files for different architectures [84].

Take the AES S-box multiplication function in finite field of $GF(2^2)$ as example, the original C code of this function is shown in Listing 5.1 [91, 92]. The corresponding LLVM IR is shown in Listing 5.2.

```c
void G4_mul( unsigned char x, unsigned char y, unsigned char *result ) {
    unsigned char a, b, c, d, e, p, q;
```
CHAPTER 5. COMPILER ASSISTED THRESHOLD IMPLEMENTATION DESIGN

```c
a = (x & 0x2) >> 1;
b = (x & 0x1);
c = (y & 0x2) >> 1;
d = (y & 0x1);
e = (a ^ b) & (c ^ d);
p = (a & c) ^ e;
q = (b & d) ^ e;
*result = ( (p<<1) | q );
```

Listing 5.1: G4_mul function in C

```llvm
define void @G4_mul(i8 zeroext %x, i8 zeroext %y, i8* nocapture %result) #0 {
    %1 = lshr i8 %x, 1
    %2 = lshr i8 %y, 1
    %3 = and i8 %2, 1
    %4 = and i8 %y, 1
    %5 = xor i8 %1, %x
    %6 = xor i8 %3, %4
    %7 = and i8 %6, %5
    %8 = and i8 %3, %1
    %9 = xor i8 %7, %8
    %10 = and i8 %4, %x
    %11 = xor i8 %7, %10
    %12 = shl nuw nsw i8 %9, 1
    %13 = or i8 %12, %11
    store i8 %13, i8* %result, align 1, !tbaa !0
    ret void
}
```

Listing 5.2: G4_mul function in LLVM IR

LLVM IR is similar to assembly language while it is target-independent. The IR is also the point where the majority of LLVM target-independent optimization takes place. Various middle-end optimization passes can be designed to modify/optimize the given IR, including our TI scheme. In this work, we design a middle-end CATI pass in LLVM to transfer the original program IR into its TI implementation. We are using LLVM version 4.0.0 running on Ubuntu version 12.04.5 in this work.
CHAPTER 5. COMPILER ASSISTED THRESHOLD IMPLEMENTATION DESIGN

5.3 CATI - Compiler Assisted Threshold Implementation Pass

We propose to get threshold implementation design for a given software cipher at the LLVM IR level. First, we present the method to find the semi-TI solutions (with no uniformity guarantee) for a given small operation automatically in Sec. 5.3.1. In Sec. 5.3.2 we give details of achieving uniformity for non-uniform solutions. In Sec. 5.3.3 we show the modifications of a given design to make it suitable for automatic TI construction. Then we show how to use the divide and conquer method to get TI solution for a given cipher efficiently in Sec. 5.3.4.

5.3.1 Automated Semi-TI Solutions

Currently the side-channel attack community still focuses on finding TI solutions manually. For example, previous works have found TI implementations for the $\chi$ operation in SHA-3, a non-linear operation involving AND and NOT $[54, 53]$. For large modules like AES S-box, the divide-and-conquer method can be used to decompose the original design into smaller operations and get the TI implementation for each operation separately $[90]$.

To get the TI solution for a given program automatically, we transfer the problem into a Boolean Satisfiability Problem (SAT) in this work. The complexity of SAT problem increases dramatically with the size of the given problem. In this section, we first show how to get the TI implementation for each small part automatically using SAT/SMT solvers.

Boolean operations can be separated into two categories, linear and nonlinear operations. For each linear operation like $\text{Shift}$ and $\text{XOR}$, the TI implementation is straightforward, with the randomness, non-completeness and uniformity of the inputs propagating to its output directly. Take operation $c = (a \ll 1) \oplus b$ as an example, its 3-share TI can be denoted as following:

$$
\begin{align*}
    c_1 &= (a_1 \ll 1) \oplus b_1 \\
    c_2 &= (a_2 \ll 1) \oplus b_2 \\
    c_3 &= (a_3 \ll 1) \oplus b_3
\end{align*}
$$

and the three output shares $c$ will preserve the randomness, non-completeness and uniformity of the input $(a, b)$ directly.

For nonlinear operations, automatic TI generation is not so straightforward. In this work, we formulate the automatic TI design problem for nonlinear operations into a satisfiability problem, and use SAT or SMT (Satisfiability Modulo Theories) solvers to solve the problem.

We denote the original program as $P$ and its TI version as $newP$. The first step in computing $newP$ from $P$ is to create a parameterized abstract syntax tree (AST) that captures all possible...
Boolean programs satisfying TI rules up to a bounded size. We call this parameterized AST as **skeleton**, following the notations in [32]. We use AND operation $c = a \& b$ with three shares $(a = a_1 \oplus a_2 \oplus a_3, b = b_1 \oplus b_2 \oplus b_3, c = c_1 \oplus c_2 \oplus c_3)$ as example here, and a candidate skeleton is shown in Fig. 5.1.

![Figure 5.1: The structure of candidate tree](image)

Each share of $c$ can be represented as an AST in Fig. 5.1, and the constraints for the ASTs are as following:

1. Each **root** node is one share of $c$. For example, three ASTs are used to represent the three shares of $c$, and the three root nodes are $c_1$, $c_2$ and $c_3$ respectively.

2. The **internal** nodes are instantiated to any bit-level logic operation such as OR, XOR, AND and NOT.

3. The **leaf** nodes are instantiated to shares of the input variables in the original program. For example, each **leaf** node can be a variable in $(a_1, a_2, a_3, b_1, b_2, b_3, c_1, c_2, c_3)$.

There are several constraints to represent the TI rules of correctness and non-completeness (without enforcing uniformity here):

1. The XOR of three root nodes should be equal to $c$ ($c = c_1 \oplus c_2 \oplus c_3$) for any $(a, b)$ and $(a, b)$. This will ensure correctness.

2. Each share skeleton cannot include all the shares for one variable in the leaf nodes. For example, the leaf nodes of $c_1$ skeleton should exclude $a_1$ and $b_1$, while $c_2$ skeleton should exclude $a_2$ and $b_2$, and $c_3$ skeleton should exclude $a_3$ and $b_3$. This will ensure non-completeness.

We transfer the above rules into constraints in SMT solvers, and then use SMT solvers to find the TI solutions that satisfy correctness and non-completeness. Initially, there may be no TI solution for an operation with the given skeleton, for example, when the size of skeleton (the depth of the binary tree) is small. We iteratively increase the skeleton size until we find a solution that satisfies...
CHAPTER 5. COMPILER ASSISTED THRESHOLD IMPLEMENTATION DESIGN

The above rules. However, the TI solution given by a SMT solver under the above constraints may not satisfy the uniformity requirement. Non-uniformity in TI will incur side-channel leakages, and such leakages can still be used to conquer the target system [85, 86]. In Section 5.4 we will show that the generated non-uniform TI designs still have some weak leakages.

To avoid such leakages and get uniform TI designs, we can either increase the skeleton size or use the strategies in Section 5.3.2 re-masking. Re-masking will keep the share numbers for each operation the same, and introduce randomness to add onto the non-uniform outputs. We present the method to compute the TI solution newP for a given program P in Algorithm 3.

Algorithm 3 Inductive synthesis of a TI version of the input program P

1: ComputeTI(P, input, output)
2: blocked ← {}
3: size ← 3
4: Candidate ← 0, CandidateP ← {}
5: while size < MAX_CODE_SIZE do
6:   newP ← ComputeCandidate(P, inputs, outputs, size, blocked)
7:   if newP does not exist then
8:     size ← size + 1
9:   else
10:      if CheckUniformity(newP) then
11:        return newP
12:      else
13:         blocked ← blocked ∪ {newP}
14:         if Candidate==0 then
15:            Candidate=1, CandidateP=newP
16:         end if
17:      end if
18:   end if
19: end while
20: return Remask(CandidateP)

In Algorithm 3 CheckUniformity is to enumerate all the possible input for the given program and count the number of each different output. If all the different output have the same number of possibility, the generated solution is uniform. We keep looking for uniform TI solution with skeletons smaller than MAX_CODE_SIZE. If there is uniform TI solution under size MAX_CODE_SIZE, Algorithm 3 returns with the uniform TI solution. Otherwise, Algorithm 3 will re-mask the first non-uniform TI solution which has the smallest skeleton size, details of re-masking will be provided in Section 5.3.2.
In this work, we use Z3 version 4.5.0 as the SMT solver [93]. We use its C++ interface for programming, and an example solution for $c = a \& b$ is given in Fig. 5.2. In this work, Z3 needs less than one second to find the first non-uniform solution for the given operation. Note that the skeletons of three shares of $c$ have the same structure here, and this makes the solving process much more efficient because of much smaller searching space. Skeletons for different shares usually have different structures and this will give more possibilities for the TI solutions.

5.3.2 Achieving Uniformity with Limited Randomness

For the algorithm in Section 5.3.1 there may be no uniform solution for the given operation with limited skeleton size and number of shares. To achieve uniformity for non-uniform solutions, we can either increase the number of shares or introduce random numbers to re-mask the generated solutions. For example, there is no uniform 3-share TI solution for AND operation, and we can either increase the number of shares to four or introduce two random numbers to re-mask the three shares to achieve uniformity [85].

If we increase the number of shares to achieve uniformity, the resource overhead in terms of Boolean operations will increase significantly because of the larger size of the skeletons. What’s more, there may be discrepancies between the number of shares required for different operations, and it will incur extra overhead and randomness for shares splitting and recombination. Thus, we keep the number of shares the same for all operations, and use re-masking to achieve uniformity for certain operations.

It has been shown that outputs of previous cryptographic runs can be used as random numbers for re-masking, and this will save random number generators (RNGs) in embedded systems [54]. In this work, we use this strategy to save randomness and use results of previous cryptographic runs as random numbers. We check the uniformity of the TI solution given by Algorithm 3 and call corresponding functions to return random numbers stored in memory to re-mask the generated semi-TI if it is not uniform.
CHAPTER 5. COMPILED ASSISTED THRESHOLD IMPLEMENTATION DESIGN

5.3.3 Minimum Modification of Given Implementation

As stated before, one advantage of the proposed scheme is it does not require knowledge of the given crypto algorithm and implementation details. However, there are two requirements that the program has to conform to so as to ensure the TI properties:

- The given program should not have branches depending on the secret information.
- The function which will be implemented in TI should not return parameters with sensitive information, and it should be in `void` type instead.

Note here that actually no branch conditions depending on the sensitive information is a basic requirement for embedded crypto implementation, because such branches will introduce timing difference which can be used for timing attacks. Thus the first requirement should be fulfilled even without the consideration of TI design. We will explain the above two requirement in details and give example solutions as follows.

5.3.3.1 Sensitive Variable-Free Branches

In TI, each sensitive variable will be split into multiple shares and they should not be recombined in the middle, as this is against the rules of non-completeness and will incur leakage. However, some cryptographic implementations may contain secret-dependent control flows, and some modifications are necessary to resolve such branches. An example is the finite field mapping operation in Canright’s S-box. The variable $x$ contains sensitive information in the original code in Listing 5.3.

```c
int G256_newbasis( int x, int b[] ) {
  int i, y = 0;
  for ( i=7; i >= 0; i-- ) {
    if ( x & 1 ) y ^= b[i];
    x >>= 1;
  }
  return ( y );
}
```

Listing 5.3: Original finite field mapping function in C

This function iteratively checks each bit of $x$ from the least significant bit (LSB). For TI construction of this code, whether to take the branch is unknown unless re-combining the multiple shares of $x\&1$, while this combination will leak information of $x$ directly. Thus the original design should avoid such branches that depend on sensitive information, and other similar operations that
cannot be represented using Boolean operations in IR. One example modification of the above branch is as following:

```c
int lsb_1 = x&1;
int lsb_2 = (lsb_1<<1)|lsb_1;
int lsb_4 = (lsb_2<<2)|lsb_2;
int Mask = (lsb_4<<4)|lsb_4;
y ^= (Mask) & b[i];
```

Listing 5.4: Modified finite filed mapping operation

The above algorithm in Listing 5.4 can be represented in Boolean operations, and then transferred to TI without combining shares of sensitive variables.

5.3.3.2 Function Interface

For functions in TI, the returned sensitive parameter will also be split into multiple shares. Therefore, the original function should be in `void` type and the sensitive parameter should be passed in and out of the function in a pointer type. To achieve this, designers should modify the target implementation before TI generation. For example, for the multiplication function in finite field \( GF(2^2) \), the original function interface is like:

```c
unsigned char G4_mul(unsigned char x, unsigned char y )
```

This function should be re-designed into a `void` type with the returning parameter as a pointer as follows:

```c
void G4_mul(unsigned char x, unsigned char y, unsigned char *result )
```

The LLVM IR representation for the above function interface is thus as following:

```llvm
define void @G4_mul(i8 zeroext %x, i8 zeroext %y, i8* nocapture %result)
```

The corresponding TI implementation of this function would be \( G4\_mul\_TI \) function in LLVM IR:

```llvm
define void @G4_mul\_TI(i8 zeroext %x.0, i8 zeroext %x.1, i8 zeroext %x.2, i8 zeroext %y.0, i8 zeroext %y.1, i8 zeroext %y.2, i8* nocapture %result.0, i8* nocapture %result.1, i8* nocapture %result.2)
```

It can be seen that the two input variables \( x \) and \( y \) each is split into three shares, and the output parameter point (\( i8\star %result \)) is also split into three pointers: (\( i8\star %result.0, i8\star %result.1, i8\star %result.2 \)).
Correspondingly, an example of calling this function in TI design is as following:

```plaintext
call void @G4_mul_TI(i8 %81, i8 %82, i8 %83, i8 %27, i8 %34, i8 %41, i8* %12,
                     i8* %13, i8* %14)
```

Note that due to the first requirement, our proposed methods are not applicable to some type of cipher implementations, like RSA. We will discuss more in Section 5.5. There is no other special requirement other than the above two, and the proposed method can be directly applied to most ciphers like AES, DES, SHA-3, etc.

### 5.3.4 Modular Design

As discussed before, it is beyond the current computation power to find TI solutions for an entire cipher, and we have to rely on the divide-and-conquer method for practical TI construction. We will take modular design, as shown in Fig. 5.3, and the interfaces between different modules will be important.

![Figure 5.3: Divide-and-conquer strategy](#)

We denote the TIs of $G$ and $F$ as $G'$ and $F'$ respectively, and the input of $G'$ as $(x, y, z)$, the output of $F'$ as $(d, e, f)$. The output of $G'$, which is also the input of $F'$, can be denoted as $(a, b, c)$. Then the combination of $F'$ and $G'$ is the TI of the whole program, and this TI design fulfills all the requirements if $G'$ and $F'$ fulfill the requirements, respectively.

- **Correctness**: For all possible $(x, y, z)$ and $(a, b, c)$, $G'(x, y, z) = G(x, y, z)$; for all possible $(a, b, c)$ and $(a, b, c)$, $F'(a, b, c) = F(a, b, c)$. Thus we have $F'(G'(x, y, z)) = F(G(x, y, z))$ for all $(x, y, z)$ and $(x, y, z)$.
• **Uniformity**: In this work, we check the uniformity of \( G' \) and \( F' \) respectively and introduce randomness for re-masking if necessary. Thus if \( G' \) and \( F' \) are uniform respectively, the whole TI module should be uniform.

• **Non-completeness**: Non-completeness holds for each module respectively. In software implementations, there is a synchronization layer (registers) between \( G' \) and \( F' \) to store the intermediate variables \((a, b, c)\). For hardware implementations, like in [90], a synchronization layer of registers should be added to avoid glitches in the output of \( G' \). Thus non-completeness is ensured.

Thus we can separate the original problem into smaller sub-problems and get their TIs instead. In LLVM, we can iteratively scan the IR and combine \( \alpha \) LLVM IR instructions together and try to find its TI solution using Algorithm 3. The size of \( \alpha \) will affect the efficiency and resource overhead directly. For larger \( \alpha \), the size of the SAT problem input to SMT solver will increase and thus the solving time for each group of instructions will increase dramatically. For smaller \( \alpha \), a larger number of sub-problems will be generated and this will incur larger resource overhead in the end. Thus there is a balance between the solution speed and area overhead for the proposed method in this work. We believe that with the development of SAT/SMT solver and increase of computation power, we can increase \( \alpha \) in the future to achieve higher efficiency.

We use some IR statements in Listing 5.2 as an example to demonstrate the modular TI construction proposed in this work. First we modify the function interface according to the description in Sec. 5.3.3.2, and the shares of \( \%x \) are \((\%x.0, \%x.1, \%x.2)\). We set \( \alpha = 1 \) to process each IR instruction individually, and the TI construction process is shown in Fig. 5.4.

As operations lshr (linear shift right) and XOR are linear, the TI construction of \( L1 \) and \( L2 \) are straightforward, and the variable of \( \%1 \) in the original code becomes three shares \((\%0, \%1, \%2)\) in the generated TI design, while the variable of \( \%5 \) becomes \((\%60, \%61, \%62)\). For AND operation, the input variables \( \%5 \) and \( \%6 \) become multi-share \((\%60, \%61, \%62)\) and \((\%63, \%64, \%65)\), respectively. The computation to generate one share of the output \( \%7 \) is shown in green color, with computations for generating other two shares of the output omitted considering the space. As AND operation in \( L3 \) is non-linear, we use (pseudo) random numbers to re-mask the generated shares, and the re-masking operations are shown in red color. For 3-share AND operation, we use two random numbers \( \%66 \) and \( \%67 \) to re-mask the three shares. For example, the first share of \( \%7 \) is \( \%75 \) and it is masked with \( \%66 \) here.

As shown in Fig. 5.4, the generated TI of AND operation has some redundancies, and this is caused by the structure of skeleton. We can use the optimization passes of LLVM to improve the efficiency, which will be shown in Sec. 5.4.1.
5.4 Result

We implement the proposed scheme in LLVM V4.0.0 together with Z3 V4.5.0, and target a commercial ARM Cortex-M3 development board. We generate the 1st-order TI designs for SHA-3 and AES using the proposed scheme automatically. We will present the resource overhead of the generated TI implementations, and perform real side-channel attacks on the TI implementations in this section.

5.4.1 Resource Overhead of the Generated TI Implementation

In this work, we use the standard 32-bit implementation of SHA-3 [73] and 8-bit AES implementation based on Canright’s S-box [91, 92] as the benchmarks. They both are commonly used public implementations. We make few modifications to the code according to the description in Section 5.3.3.

We generate the 1st-order TI implementations of both AES and SHA-3 using our algorithm. Here we also use ‘-O3’ optimization option in LLVM to improve the generated IR code. As the back-end for different architectures will generate different assembly code, we use the number of LLVM IR variables to evaluate the overhead instead in this work. We present the numbers of IR variables for the original design, the generated TI design, and the TI design after ‘-O3’ optimization.

Figure 5.4: IR representation of the CATI process
In Table 5.1, we show the results for KeccakF function in SHA-3 and four functions in AES S-box. The increase in the number of variables for the TI KeccakF function is about eight times the original implementation, and the inherent ‘-O3’ optimization can reduce 40%. The original SHA-3 KeccakF function requires 996 IR variables in LLVM, while its generated TI version requires 7000 IR variables. We use ‘-O3’ optimization pass to improve the generated TI implementations and results show that after optimization, the generated TI needs about 4179 LLVM IR variables instead.

Similarly, for the four S-box functions, the increase for the TI implementation is about one order of magnitude, and the ‘-O3’ optimization reduces about 30%. we present the resource consumption for some functions in Canright S-box implementation. For multiplication in finite field of $GF(2^2)$, the original implementation of G4_mul requires 13 LLVM IR variables, while the optimized TI implementation needs 100 variables. Similarly, the resource overhead for square function in $GF(2^2)$ (G4_sq), the multiplication function in $GF((2^2)^2)$ (G16_sq) and the inversion function in $GF(((2^2)^2)^2)$ (G256_inv) are shown in Table 5.1. Results show that the resource overhead is around $4-8$ times, depending on the target implementations. The backends have different optimization based on target architectures’ instructions and features, and we will not report them here.

Provable security is achieved with cost. For example, LUT-Masked Dual-rail with Precharge Logic (LMDPL) incurs about 10 times resource overhead \cite{29}. While Garbled Circuits significantly improves the security of the target platform, the overhead will be over 1000 times \cite{94}. The overhead for our TI implementations of around $4-8$ times is at a reasonable level. With the improvement of computation power and SMT solvers, we can increase $\alpha$ and combine multiple instruction every time for TI design, and this will decrease the resource overhead significantly in the future.

<table>
<thead>
<tr>
<th>S-box</th>
<th>Original</th>
<th>TI</th>
<th>opt-O3</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHA-3</td>
<td>KeccakF</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>996</td>
<td>7000</td>
<td>4179</td>
</tr>
<tr>
<td>S-box</td>
<td>G4_mul</td>
<td>13</td>
<td>146</td>
</tr>
<tr>
<td></td>
<td>G4_sq</td>
<td>5</td>
<td>68</td>
</tr>
<tr>
<td></td>
<td>G16_mul</td>
<td>18</td>
<td>152</td>
</tr>
<tr>
<td></td>
<td>G256_inv</td>
<td>12</td>
<td>86</td>
</tr>
</tbody>
</table>
CHAPTER 5. COMPILER ASSISTED THRESHOLD IMPLEMENTATION DESIGN

5.4.2 Side-Channel Attacks Results

To evaluate the side-channel resilience, we implement the original and the generated 1st-order TI designs of AES and SHA-3 on a commercial ARM Cortex-M3 (STM32F103C8T6) development board. We sample the EM traces using a Teledyne LeCroy WaveRunner 640zi oscilloscope with the sampling rate of 5G samples/second.

First, we implement the unprotected AES based on Canright’s S-box and run Correlation Power Analysis (CPA) for leakage detection. The CPA result using $3 \times 10^3$ traces for the correct key is shown in Fig. 5.5a. It shows that for the unprotected AES implementation on Cortex-M3 processor, the leakage is very strong, reaching about negative 0.25 with $3 \times 10^3$ traces. Attacker can use this leakage to extract the encryption/decryption key.

![Leakage detection of original AES](image1)

(a) Leakage detection of original AES implementation using $3 \times 10^3$ traces

![Key guess results using different number of traces](image2)

(b) Key guess results using different number of traces

Figure 5.5: Correlation Power Analysis of the original AES implementation
As the point of leakage is obvious in Fig. 5.5a, we use the EM signal at this point for further evaluation and show the key guess results in Fig. 5.5b. In Fig. 5.5b, the curve in black is the correlation result for the correct key, while curves in gray are for other key guesses. It shows that the correct key stands out quickly among all key guesses with only several hundred of traces, and the attacker needs a very small number of traces to recover the secret key byte successfully.

For the generated TI of AES, we use the same settings for data acquisition and attacks. The leakage detection results with $8 \times 10^5$ traces are shown in Fig. 5.6a, which shows that the protected AES implementation is very secure under first-order attacks. The key guess results are shown in Fig. 5.6b in which we check both the maximum and minimum correlation results. It shows that the correct key guess cannot stand out of the key guesses, thus the attacker cannot get useful information through the side-channel EM/power leakages of this generated TI design.

Figure 5.6: Correlation Power Analysis of the original AES implementation
CHAPTER 5. COMPILER ASSISTED THRESHOLD IMPLEMENTATION DESIGN

We also evaluate the side-channel leakages of the generated TI S-box using Information Leakage Amount (ILA) tool [95], and the results show that the ILA value of the generated TI is 0 when the random number re-masking is on, and it is 0.0016 when the randomness is off. It means that there is still weak leakages when the re-masking operation is off, and these leakages are from the non-uniformity of the generated TI design. The introduced random numbers will mask the non-uniformity and thus eliminate the leakages.

We also use T-test [96] to evaluate the side-channel leakages of both implementations, and present the results using 100,000 traces for the original AES S-box implementation in Figure 5.7a. Figure 5.7a shows that the original AES S-box is very leaky according to the T-test results, with T-value reaching about 115. Usually a threshold of 4.5 is used for security evaluation, which means that the original AES S-box is very leaky for such a threshold. We also evaluate the side-channel leakages of the generated TI of AES S-box, both the nonuniform and uniform (with re-masking) ones, and show the results in Figure 5.7.

Figure 5.7b shows that the generated TI implementation without re-masking has a T-value reaching 4.9988, while the T-value of the uniform TI implementation with re-masking in Figure 5.7c has a maximum number smaller than 4.5 [96]. Thus the non-uniform TI implementation still small some side-channel leakages, though very weak, while the re-masked implementation is first-order side-channel attacks secure.

We also evaluate the side-channel leakages through T-test for the original and generated TI implementations of SHA-3. The T-value results of the original Keccak implementation is shown in Figure 5.8a. Figure 5.8a shows that the side-channel leakages of the original Keccak implementation is very strong, and the T-value is reaching almost 90. For the generated nonuniform and uniform TI solutions, the corresponding T-test results are shown in Figure 5.8.

The same as AES, Figure 5.8b shows that the nonuniform Keccak implementation has some leaky points, and the maximum T-value is reaching about 4.9153. Thus the generated nonuniform TI Keccak implementation is much securer than the original Keccak implementation. We also notice that the maximum T-value of the re-masked TI Keccak implementation is lower than the threshold, which means that the re-masked TI SHA-3 is secure against side-channel attacks.

It has been mentioned in previous works [97, 98] that the threshold for T-test should be higher when the number of points for each power/EM trace is large. While we have more than $5 \times 10^5$ points for each trace, this threshold should be higher and thus the generated nonuniform/uniform TI implementations are both very secure, while the nonuniform implementations still have weak leakages.
CHAPTER 5. COMPILER ASSISTED THRESHOLD IMPLEMENTATION DESIGN

5.5 Discussion and Future Work

In this work, we present an automatic TI scheme based on LLVM platform, and implement the generated TI design on a commercial ARM Cortex-M3 processor for side-channel leakage evaluation. The results show that the compiler-assisted TI automation is effective. The generated designs have $4-8$ times resource overhead in terms of the number of LLVM IR variables. The real attacks on commercial Cortex-M3 board show that the generated TI designs are secure against side-channel attacks.

In this work, we generate TI designs for both 8-bit AES implementation and 32-bit SHA-3 implementation using our algorithm in LLVM. With the development of LLVM, our scheme will be applicable to more architectures and platforms \cite{84,99,100,101}.

It has been shown that TI is an excellent countermeasure against side-channel attacks for both software and hardware implementations \cite{86,85}. Threshold implementation can fight against glitches in hardware implementations, thus it is a good side-channel countermeasure for hardware systems like FPGA and ASIC. The scheme in this work can be extended to hardware implementations in Verilog and VHDL with mature front-end and back-end support. Meanwhile, some works on High-Level Synthesis (HLS) have been published \cite{100,101}. These works compile C/C++ implementations to hardware implementations using LLVM framework. To implement automatic TI generation tools for hardware (FPGA/ASIC) implementations utilizing HLS technologies is also a part of future work.

5.6 Summary

In this work we present an automated threshold implementation design based on LLVM. The proposed scheme is independent on either the programming languages or the hardware architectures, and it does not require knowledge of the original cipher implementation. Our proposed method can generate secure TI solution for a given implementation automatically. We implement the generated TI on commercial hardware platform and the side-channel attacks results show that the generated TI design are secure against side-channel attacks.
CHAPTER 5. COMPILER ASSISTED THRESHOLD IMPLEMENTATION DESIGN

Figure 5.7: T-test results for the original and generated TI implementation of AES S-box
(a) T-test results for the original Keccak implementation

(b) T-test results for the nonuniform TI Keccak implementation

(c) T-test results for the re-masked TI Keccak implementation

Figure 5.8: T-test results for the original and generated TI implementation of Keccak
Chapter 6

Differential Fault Analysis of SHA-3

6.1 Introduction and Motivation

DFA is a powerful and efficient attack method, and it utilizes the dependency of the faulty output on the internal intermediate variables to recover the secret. DFA has been used to break many cryptographic algorithms. For example, it has been used to extract the secret key for symmetric ciphers. It was first introduced to conquer the Data Encryption Standard (DES) algorithm [16], and later it was used to break the Advanced Encryption Standard (AES) [17]. Many other ciphers have also been hacked by DFA, including CLEFIA [18], Mickey [19, 102] and Grain [20, 21].

Some existing hash standards have been evaluated against DFA attacks, including SHA-1 [22], Streebog [23], MD5 [24] and GrøStl [25]. For hash functions in general usage, DFA can be used to retrieve the original message. For example, it is found in [24] that on average 144 random faults are required for MD5 in order to discover the input message block. For GrøStl algorithm, on average 280 single-bit faults are needed to invert each compression step. When hash functions are used in the message authentication code (MAC) mode with a secret key, DFA also becomes a great threat to recover the key and then generate forgery messages and MAC against authentication. For example, in [22], the input of SHA-1, including the secret key and message, is fully extracted with about 1000 random faults. For Streebog, an average number of faults that varies between 338-1640 is required to recover the secret key for different MAC settings [23]. When GrøStl is used in a keyed hash function, about 300 faults are needed to retrieve the secret key [25].

Only one work exists for the differential fault analysis of SHA-3 [10]. A single-bit fault model is used in work [10], and only two modes of SHA-3, namely SHA3-384 and SHA3-512 have been discussed. Injection of single-bit faults into crypto systems requires laser or other invasive methods, which are costly and become ineffective as the technology scales down. While other non-invasive
fault injection methods, such as clock glitches and supply voltage variation, are more general and would affect a group of bits in intermediate states all together, i.e., inducing byte-level faults.

We propose a differential fault analysis method on all the four modes of SHA-3 under relaxed byte-level fault models. My work has a number of contributions as follows:

- We present the fault propagation of SHA-3 under relaxed fault models;
- We extend DFA to also SHA3-224 and SHA3-256;
- We propose an optimization method for the proposed attack to use the smallest number of faults to recover the whole state of $\chi_{i}^{22}$;
- We extend differential fault analysis on SHA-3 system with input message longer than bitrate.

We simulate all the proposed methods for all four modes of SHA-3. Results show that, for SHA3-384 and SHA3-512, about 120 random single-byte fault injected into the SHA-3 implementation are needed to recover an entire internal state, while about 500 single-bit faults are needed in previous work [10]. Our attack method can conquer SHA3-224 and SHA3-256 as well, while the numbers of required effective injected faults are about 250 and 150 respectively. We further propose a heuristic algorithm to search for the upper bound of the proposed attacks, and this algorithm can also be used to improve the efficiency of the DFA with more control on the faults injected. We simulate the proposed method on SHA3-384 and SHA3-512, and it shows that only 17 selected faults are enough to recover the internal state under the single-byte fault model, and 129 faults under the single-bit fault model.

The rest of this chapter is organized as follows. In Section 6.2, we present the fault models and notations used in this work. In Section 6.3, we analyze the fault propagation process in SHA-3 and construct the fault signatures. In Section 6.4, we present the attack on SHA3-384 and SHA3-512 using the proposed fault signature method under the relaxed fault models. In Section 6.5, we extend the attack to SHA3-224 and SHA3-256, and show the method to further improve the attack efficiency. In Section 6.6, we propose a heuristic algorithm to improve the efficiency of the DFA with more control of the faults injected, and derive the lower bound on the number of fault injections needed for the proposed attacks. In Section 6.7, we discuss attacks on SHA-3 systems with the input message size larger than the bitrate, and the protection of SHA-3 systems against DFAs.

6.2 Fault Model and Notations

All the above operations are reversible [1]. Thus if an internal state is recovered in SHA-3, the original message and all the other internal states can be recovered. We set our target as recovering

74
CHAPTER 6. DIFFERENTIAL FAULT ANALYSIS OF SHA-3

the entire internal state of $\chi^{22}_i$ (1,600 bits). We annotate the last two rounds of SHA-3 operations and important intermediate states in Figure 6.1, and use these notations in the rest of this work.

In this work, the fault injection point is $\theta^{22}_i$, and the attackers can only observe the clean and faulty digest, $H$ and $H'$, with length $d$ for Keccak-f[1600, $d$].

6.2.1 The Fault Model Used

Our DFA is based on byte-level faults. General fault injection methods, including clock glitches and supply voltage variations, will affect multiple bits in a data structure simultaneously. For example, one byte will be affected on 8-bit architectures, and one 16-bit word will be affected on 16-bit architectures. Multiple concurrent bit faults will interfere with each other during operations in the hash algorithm, and considering only individual independent single-bit faults does not address these interactions. To illustrate our proposed method, we use the fault model of single-byte as example:

- The attacker can inject faults into one byte of the penultimate round input $\theta^{22}_i$;
- The attacker has no control on either the position (which byte) or the value of the injected faults;
- The attacker can only observe the correct and faulty SHA-3 digest outputs, $H$ and $H'$, which are $d$-bit instead of 1600 bits;
- The attacker can inject multiple random faults into the same input message for different execution runs.

Besides single-byte fault model, we will also check our method under single-word (16-bit) fault model. As the single-bit fault model used in [10] is just a special case of the single-byte fault model, we will also check attack results under that model.
6.3 Fault Propagation and Fault Signature

Generally, because of confusion and diffusion properties in crypto operations, any bit flip at the input message will affect all the bits at the output under perfect randomness and the fault analysis would not work. For SHA-3, the path from the fault injection point ($\theta^{22}_i$) to the observable output ($H$) is not very long - only two rounds of operations, and therefore different faults injected will cause different patterns at the differential output $\Delta H = H \oplus H'$. We call such differential patterns as Fault Signature (FS) in this work. Different fault signatures can be used to identify the injected fault, and then recover the internal state bits.

Different from previous crypto systems like AES and DES, which are organized and operated in several modules (like 8-bit bytes), Keccak is organized at bit level and operations are performed on bits. For block ciphers like AES, the attacker can observe the differential output of 16 bytes to identify the injected fault positions and values. It has been shown in \[10\] that differential output of SHA-3 has much more complicated characteristics, and therefore DFA methods for previous block ciphers cannot be applied to Keccak based functions. In this section, we show the fault propagation process in SHA-3 and extract the fault signature for each possible injected fault.

6.3.1 Observable Hash Digest

For DFA, the first step is to select a comparison point (intermediate state of the algorithm), where information obtained from the differential outputs is used to match the various patterns of the fault propagation so as to identify the injected fault or recover secret message or key. In \[10\], the comparison point is picked at $\theta^{23}_o$ for SHA3-384 and SHA3-512 to identify the single-bit fault injected. For SHA3-384 and SHA3-512, a whole plane of 320 bits ($y = 0$, the bottom plane) at the output $H$ is observable. Because all the operations $\rho,$ $\pi,$ $\chi,$ and $\iota$ are reversible, the attacker can make use of this plane to recover the bottom plane (320 bits) of $\chi^{23}_i$:

$$\chi^{23}_i(X, 0, Z) = \chi^{-1} \circ (\iota^{23})^{-1}(H(y = 0)),$$

and the differential, $\Delta \chi^{23}_i(X, 0, Z)$, can be derived. When selecting the comparison point at $\chi^{23}_i$, we need to construct fault signatures at $\chi^{23}_i$, $FS_{\chi^{23}_i}$, for attacks in this work.

For SHA3-224 and SHA3-256, only a partial bottom plane of the output of the $f$ function is observed, and therefore $\chi^{23}$ cannot be inverted directly, according to (2.6). For SHA3-224 and SHA3-256, we present corresponding attack methods in Section 6.5.
6.3.2 Fault Signature Generation

For single-byte fault model, any internal state of Keccak-\(f[1600, d]\) is composed of 200 bytes \((0 \leq P < 200)\), and the fault value \((F, \text{defined as the differential of the state byte of the penultimate round input})\) ranges from 1 to 255, where for each bit of \(F\), 0 means the corresponding state bit does not change, and 1 means the state bit flips. For example, \(F = 1\) means the lowest bit of the state byte flips. For any possible fault \((F)\) at any one of the 200 positions \((P)\), we denote the corresponding fault signature at \(\chi_i^{23}\) as \(FS_{\chi_i^{23}}[P][F]\). Without further specification, all fault signatures are 1,600 bits, standing for the 1,600 differential bits of the state caused by the fault \(F\) injected at byte \(P\) of \(\theta_i^{22}\).

For faults injected at \(\theta_i^{22}\), it will propagate to \(\chi_i^{23}\) through the operations shown in Figure 6.1. We separate these operations into two categories:

- Operations that will not change bit values of fault signatures, including bit rotation operations \(\rho\) and \(\pi\) that only change the bit positions, and constant number addition operation \(\iota\).

- Operations that will change the bit values of fault signatures, which involve multiple bits to generate a single output bit, namely \(\theta\) and \(\chi\). There is also difference between these two operations, \(\theta\) is linear (only consisting of XOR operations) while \(\chi\) is non-linear (consisting of operations AND and NOT).

In the first kind of operations, for \(\rho\) and \(\pi\), faults at the input will go through the operation (position permutation) directly and propagate to the output, i.e., \(\Delta \rho_o = \rho(\Delta \rho_i)\) and \(\Delta \pi_o = \pi(\Delta \pi_i)\). For operation \(\iota\), the fault does not change at all, i.e., \(\Delta \iota_o = \Delta \iota_i\).

For the second kind of operations, one output bit is generated from multiple input bits. For \(\theta^{22}\) operation, one single-bit fault \(\Delta \theta_i^{22}(x, y, z)\) will propagate to 11 bits of \(\theta_o^{22}\), with their differential denoted as \(\Delta \theta_o^{23}(x, y, z)\), \(\Delta \theta_o^{22}(x + 1, Y, z)\) and \(\Delta \theta_o^{22}(x - 1, Y, z + 1)\), which are on the state bit and its two neighbor columns in different sheets. For the single-byte fault model, all the faulty bits are in the same lane of \(\theta_i^{22}\). With \(\theta\) operation, no \(\theta_o^{22}\) bit will involve more than one faulty bit. Thus, for \(\theta^{22}\), we have \(\Delta \theta_o^{22} = \theta(\Delta \theta_i^{22})\).

In this work, we use a single-bit fault at \(\theta_i^{22}(0, 0, 0)\) (\(\Delta \theta_i^{22}(0, 0, 0) = 1\) while all other bits of \(\Delta \theta_i^{22}\) are 0) as an example to demonstrate the fault propagation in SHA-3, and use it to explain the construction of fault signatures. According to the above analysis of fault propagation through different operations, the single-bit fault will be diffused to 11 bits after \(\theta^{22}\) operations, and then rotated into different lanes and rows through \(\rho\) and \(\pi\). The fault signature \(FS_{\chi_i^{23}}\) at the input of \(\chi^{22}\) for this single-bit fault is shown in Figure 6.2.
CHAPTER 6. DIFFERENTIAL FAULT ANALYSIS OF SHA-3

Due to the linear properties of θ, ρ and π operations, each bit of $FS_{\chi^{22}_i}$ will be either 0 or 1, depending on the value and position of the injected faults only. As the fault keeps propagation, two important processes, $\chi^{22}$ and $\theta^{23}$, will determine the fault signature $FS_{\chi^{23}_i}$. If we denote the fault propagation of $\chi^{22}$ as $FP_{\chi}$, and the fault propagation of $\theta^{23}$ as $FP_{\theta}$, the corresponding fault signature at $\chi^{23}_i$ can be denoted as follows (note that operation $\iota$ does not change the propagation of the faults):

$$FS_{\chi^{23}_i} = \pi \circ \rho \circ FP_{\theta} \circ FP_{\chi}(\Delta \chi^{22}_i). \hspace{1cm} (6.1)$$

We next analyze fault propagation of $\chi^{22}$ and $\theta^{23}$ in following section respectively.

6.3.2.1 Fault Propagation in $\chi^{22}$ - $FP_{\chi}$

$\chi$ is the only nonlinear operation in Keccak, and its AND operation leaks information of its input state bits with fault(s) on $\chi_i$. Under the single-bit fault model in [10], no more than one bit will be polluted in each row of $\chi^{22}_i$, as also shown in Figure 6.2 for vectors $\Delta \chi^{22}_i(X,y,z)$. For the

Figure 6.2: Fault signature at $\chi^{22}_i$ for the example single-bit fault injected at $\theta^{22}_i$
CHAPTER 6. DIFFERENTIAL FAULT ANALYSIS OF SHA-3

relaxed models used in this work, multiple bits may be polluted in one row of $\chi_i^{22}$. In this section, we present the general fault propagation of multi-bit faults in $\chi$ operation.

Denote five bits in one row of $\chi$ input as \{a_i, b_i, c_i, d_i, e_i\}, then five bits of corresponding $\chi_o$ output row can be denoted as $a_o = a_i \oplus (b_i \cdot c_i)$, $b_o = b_i \oplus (c_i \cdot d_i)$, $c_o = c_i \oplus (d_i \cdot e_i)$, $d_o = d_i \oplus (e_i \cdot a_i)$ and $e_o = e_i \oplus (a_i \cdot b_i)$.

We take $a_o$ as an example to demonstrate the fault propagation in $\chi$ operation. Bit $a_o$ is affected by bits $a_i$, $b_i$ and $c_i$:

1. With a single-bit fault on $a_i$ ($\Delta a_i = 1$), $\Delta a_o = \Delta a_i = 1$.
2. With a single-bit fault on $b_i$ ($\Delta b_i = 1$), $a_o' = a_i \oplus (\bar{b}_i \cdot c_i)$, and then $\Delta a_o = \Delta b_i \cdot c_i = c_i$, which leaks the internal state $c_i$ information.
3. With a single-bit fault on $c_i$ ($\Delta c_i = 1$), $a_o' = a_i \oplus (\bar{b}_i \cdot c_i')$, and then $\Delta a_o = (1 \oplus b_i) \cdot \Delta c_i = \bar{b}_i$.
4. With a two-bit fault on $a_i$ and $b_i$ ($\Delta a_i = \Delta b_i = 1$), $a_o' = a_i' \oplus (\bar{b}_i \cdot c_i)$, and then $\Delta a_o = \Delta b_i \cdot c_i \oplus (1 \oplus b_i) \cdot \Delta c_i = b_i \oplus c_i$.
5. With a two-bit fault on $b_i$ and $c_i$ ($\Delta b_i = \Delta c_i = 1$), $a_o' = a_i \oplus (\bar{b}_i \cdot c_i')$, and then $\Delta a_o = \Delta b_i \cdot c_i \oplus (1 \oplus b_i) \cdot \Delta c_i = b_i \oplus c_i$.
6. With a two-bit fault on $a_i$ and $c_i$ ($\Delta a_i = \Delta c_i = 1$), $a_o' = a_i' \oplus (\bar{b}_i \cdot c_i')$, and then $\Delta a_o = \Delta a_i \oplus (1 \oplus b_i) \cdot \Delta c_i = b_i$.
7. With a three-bit fault ($\Delta c_i = \Delta b_i = \Delta c_i = 1$), $a_o' = a_i' \oplus (\bar{b}_i \cdot c_i')$, and thus $\Delta a_o = \Delta a_i \oplus \Delta b_i \cdot c_i \oplus (1 \oplus b_i) \cdot \Delta c_i = \bar{b}_i \oplus c_i$.

In summary, we can denote the fault signature for bit $\chi_o^{22}(x, y, z)$ as in TABLE 6.1.

According to the above analysis, we can see that the nonlinear $\chi$ operation may cause leakage of some $\chi_i^{22}$ bits in the differential $\chi^{22}$ output. We present the whole fault pattern at $\chi_o^{22} = 0, 0, 0$ as in Figure 6.3 in which $\Delta \chi_o^{22}(x, y, z)$ is denoted as $C(x, y, z)$ for simplicity, and the same single-bit fault $\Delta \theta_i^{22}(0, 0, 0) = 1$ example is assumed.

In Figure 6.3, each differential bit $\Delta \chi_o^{22}(x, y, z)$ takes a value of ‘0’, ‘1’ or ‘x’, in which 1 (0) means this corresponding output bit flips (does not flip) with the specific fault injected, respectively, regardless of the internal states. However, ‘x’ at a bit position means that the corresponding $\Delta \chi_o^{22}$ bit value depends on some $\chi_i^{22}$ bit(s), and it can be ‘0’ or ‘1’. For example, we denote $\Delta \chi_o^{22}(0, 0, 44)$ as ‘x’, because $\Delta \chi_o^{22}(0, 0, 44) = \chi_i^{22}(2, 0, 44)$ under the fault injected ($\Delta \theta_i^{22}(0, 0, 0) = 1$), and $\chi_o^{22}(0, 0, 44)$ would flip if $\chi_i^{22}(2, 0, 44) = 1$, otherwise it remains unchanged if $\chi_i^{22}(2, 0, 44) = 0$. Thus, if the attacker has knowledge of $\Delta \chi_o^{22}(0, 0, 44)$ and the injected fault, he can construct the corresponding fault signature and then recover the bit $\chi_i^{22}(2, 0, 44)$.
Table 6.1: Fault propagation of operation $\chi^{22}$

<table>
<thead>
<tr>
<th>Fault at $\chi$ input</th>
<th>Fault signature at $\chi$ output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Delta \chi_i^{22}(\lfloor x : x + 2 \rfloor, y, z)$</td>
<td>$FS_{\chi_i^{22}}(x, y, z)$</td>
</tr>
<tr>
<td>[1,0,0]</td>
<td>1</td>
</tr>
<tr>
<td>[0,1,0]</td>
<td>$\chi^{22}_i(x + 2, y, z)$</td>
</tr>
<tr>
<td>[0,0,1]</td>
<td>$\chi^{22}_i(x + 1, y, z)$</td>
</tr>
<tr>
<td>[1,1,0]</td>
<td>$\chi^{22}_i(x + 2, y, z)$</td>
</tr>
<tr>
<td>[0,1,1]</td>
<td>$\chi^{22}_i(x + 1, y, z) \oplus \chi^{22}_i(x + 2, y, z)$</td>
</tr>
<tr>
<td>[1,0,1]</td>
<td>$\chi^{22}_i(x + 1, y, z)$</td>
</tr>
<tr>
<td>[1,1,1]</td>
<td>$\chi^{22}_i(x + 1, y, z) \oplus \chi^{22}_i(x + 2, y, z)$</td>
</tr>
</tbody>
</table>

\[ \Delta \theta^{23}_o(x, y, z) = \Delta \theta^{23}_i(x, y, z) \oplus (\oplus_{y=0}^{4} \Delta \theta^{23}_i(x - 1, y, z)) \oplus (\oplus_{y=0}^{4} \Delta \theta^{23}_i(x + 1, y, z - 1)). \] (6.2)

Thus the fault propagation function $FP_{\theta}$ can be denoted as follows:

\[ FS_{\theta^{23}} = \theta(FS_{\chi^{22}}). \] (6.3)

Each bit of $FS_{\chi^{22}}$ can be denoted as 0, 1, or a function of certain $\chi^{22}_i$ bits. For each bit of $FS_{\theta^{23}}$, some of the corresponding 11 $FS_{\chi^{22}}$ bits may depend on the same $\chi^{22}_i$ bits, and therefore some dependencies will be eliminated with the operation of XOR. This is a key insight for our byte-level (multiple bit) fault propagation analysis. For example, in the interleaved implementation, when fault $F = 65$ is injected at $P = 16$, $\Delta \theta^{23}_i(4, 4, 3) = \chi^{22}_i(0, 4, 3)$ and $\Delta \theta^{23}_i(3, 4, 3) = \chi^{22}_i(0, 4, 3)$. $\Delta \theta^{23}_o(4, 4, 3)$, which involves the two input bits $\theta^{23}_i(4, 4, 3)$ and $\theta^{23}_i(3, 4, 3)$, will not depend on $\chi^{22}_i(0, 4, 3)$ anymore because the dependencies get canceled out by XOR between the two input bits. Eventually, the fault signature at the $\theta^{23}$ output, $FS_{\theta^{23}}$, has a similar format as $FS_{\chi^{22}}$, with each bit being 0, 1, or an odd or even function (XOR) over some $\chi^{22}_i$ bits and constant one.

As $\Delta \chi^{23}_i = \pi \circ \rho(\Delta \theta^{23}_o)$, it is easy to build the fault signature at $\chi^{23}_i$ with $FS_{\theta^{23}}$ constructed from the above analysis, thus we show $FS_{\chi^{23}}$ directly here. We use the same example to show how
CHAPTER 6. DIFFERENTIAL FAULT ANALYSIS OF SHA-3

The single-bit fault at \( \theta_{i}^{22}(0, 0, 0) \) propagates to \( \chi_{i}^{23} \). For SHA3-224 and SHA3-256, only partial bottom plane (less than 320 bits) of the output state \( H \) will be observable. Nevertheless Figure 6.4 presents the fault signature in the whole bottom plane of \( FS_{\chi_{i}^{23}} \), in which we denote \( \Delta \chi_{i}^{23}(x, 0, z) \) as \( E(x, z) \) for simplicity.

With the observed bits of \( \Delta \chi_{i}^{23} \) and the fault signatures, the attacker can work on equations which involve only one bit of \( \chi_{i}^{22} \) to recover the \( \chi_{i}^{22} \) bits, and then plug them back into equations which involve more than one \( \chi_{i}^{22} \) bit to recover the remaining \( \chi_{i}^{22} \) bits. For example, as shown in Figure 6.4 with the single-bit fault injected at \( \theta_{i}^{22}(0, 0, 0) \), the attacker can use \( FS_{\chi_{i}^{23}}(1, 0, 24) \) to recover \( \chi_{i}^{22}(2, 0, 44) \) first. Then replace \( \chi_{i}^{22}(2, 0, 44) \) in \( FS_{\chi_{i}^{23}}(0, 0, 44) \) to recover \( \chi_{i}^{22}(0, 0, 44) \).

In this section, we showed the fault propagation process in SHA-3, and analyzed the composition of fault signatures at \( \chi_{i}^{23} \). In next two sections, we will show how to use the constructed fault signatures to conquer all four modes of SHA-3.
Chapter 6. Differential Fault Analysis of SHA-3

We separate the 320 observable $\Delta \chi_i^{23}$ bits (five lanes) into two groups:

- $\Delta \chi_i^{23}.white$ contains the bits $(x, y, z)$ of $\Delta \chi_i^{23}$ with $\Delta \chi_i^{23}(x, y, z) = 0$, which means that these bits are not flipped under the injected fault;

- $\Delta \chi_i^{23}.black$ contains the bits $(x, y, z)$ of $\Delta \chi_i^{23}$ with $\Delta \chi_i^{23}(x, y, z) = 1$, which means these bits are flipped under the injected fault.

We would like to use the observed $\Delta \chi_i^{23}$ to infer the fault injection position (at byte $P_0$) and the fault value ($F_0$). For any fault $F$ at position $P$, the fault signature at the bottom plane of $\chi_i^{23}$ consists of five lanes, and we can separate the 320 bits of $FS_{\chi_i^{23}}[P][F](x, y, z)$ into three groups:

- $FS_{\chi_i^{23}}[P][F].white$ contains the bits $(x, y, z)$ with $FS_{\chi_i^{23}}[P][F](x, y, z) = 0$, i.e., the injected fault does not affect these state bits.

Figure 6.4: Fault signature at $\chi_i^{23}$ (Bottom plane)

6.4 Differential Fault Analysis of SHA3-384 and SHA3-512

In this section, we use the constructed fault signatures in the previous section to conquer SHA3-384 and SHA3-512. In Section 6.4.1, we will present the method to identify the injected fault, including the position $P$ and value $F$, using the constructed fault signatures. Then we show how to recover some $\chi_i^{23}$ bits using the identified fault in Section 6.4.2.

6.4.1 Fault Position $P$ and Value $F$ Recovery

We separate the 320 observable $\Delta \chi_i^{23}$ bits (five lanes) into two groups:

- $\Delta \chi_i^{23}.white$ contains the bits $(x, y, z)$ of $\Delta \chi_i^{23}$ with $\Delta \chi_i^{23}(x, y, z) = 0$, which means that these bits are not flipped under the injected fault;

- $\Delta \chi_i^{23}.black$ contains the bits $(x, y, z)$ of $\Delta \chi_i^{23}$ with $\Delta \chi_i^{23}(x, y, z) = 1$, which means these bits are flipped under the injected fault.

We would like to use the observed $\Delta \chi_i^{23}$ to infer the fault injection position (at byte $P_0$) and the fault value ($F_0$). For any fault $F$ at position $P$, the fault signature at the bottom plane of $\chi_i^{23}$ consists of five lanes, and we can separate the 320 bits of $FS_{\chi_i^{23}}[P][F](x, y, z)$ into three groups:

- $FS_{\chi_i^{23}}[P][F].white$ contains the bits $(x, y, z)$ with $FS_{\chi_i^{23}}[P][F](x, y, z) = 0$, i.e., the injected fault does not affect these state bits.
CHAPTER 6. DIFFERENTIAL FAULT ANALYSIS OF SHA-3

- $FS_{\chi_i^{23}}[P][F].black$ contains the bits $(x, y, z)$ with $FS_{\chi_i^{23}}[P][F](x, y, z) = 1$, which are for sure to flip when the fault is injected.

- $FS_{\chi_i^{23}}[P][F].grey$ contains the bits $(x, y, z)$ with $FS_{\chi_i^{23}}[P][F](x, y, z)$ as a function dependent on some bits of $\chi_i^{22}$, i.e., they can leak some internal state bits information, and can be 0 or 1.

For the correct fault $F_0$ injected at the correct position $P_0$, the following relationships should hold:

- For any bit in $FS_{\chi_i^{23}}[P_0][F_0].white$, this bit should be in $\Delta\chi_i^{23}.white$;
- For any bit in $FS_{\chi_i^{23}}[P_0][F_0].black$, this bit should be in $\Delta\chi_i^{23}.black$;
- For any bit in $FS_{\chi_i^{23}}[P_0][F_0].grey$, it can be in $\Delta\chi_i^{23}.white$ or $\Delta\chi_i^{23}.black$, depending on some internal state bits.

We summarize the above relationships as following set relations:

$$
\begin{align*}
FS_{\chi_i^{23}}[P][F].white & \subseteq \Delta\chi_i^{23}.white \\
FS_{\chi_i^{23}}[P][F].black & \subseteq \Delta\chi_i^{23}.black \\
\Delta\chi_i^{23}.white & \subseteq \{FS_{\chi_i^{23}}[P][F].white \cup FS_{\chi_i^{23}}[P][F].grey\} \\
\Delta\chi_i^{23}.black & \subseteq \{FS_{\chi_i^{23}}[P][F].black \cup FS_{\chi_i^{23}}[P][F].grey\}.
\end{align*}
$$ (6.4)

By checking relationships in (6.4), the attacker can exclude many positions and fault values. If only one position with one fault value satisfies these relationships, the injected fault is discovered. All the $FS_{\chi_i^{23}}[P_0][F_0].grey$ bits now are mapped to either zero (white) or one (black) in the observed differentials, and therefore the internal state bits can be recovered.

We implement the attacks on both interleaved and non-interleaved versions of Keccak implementations in C++ [73], and run all the fault signature generation and mapping between the observed differential and the hypothesized signatures on a workstation, which consists of an Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz and 8GB memory. Results show that our one-time fault signature generation algorithm takes about 14.5s for offline execution, and it takes less than 0.3 ms to find the correct injected fault and recover the internal state bits from one fault.

For the non-interleaved version of SHA-3, using our algorithm, the attacker can find the unique injected fault with probability 99.13% under single-byte fault model. For the rest 0.87% probability, more than one faults satisfy the relationships in (6.4). For interleaved version, the attacker can find the unique fault with 100% probability under one single-byte fault injection. We define such unique
faults as effective faults. As only effective faults are useful for identifying state bits, the higher percentage of the effective fault, the more efficient the attacks will be.

We also check our algorithms under other fault models. For the single-bit fault model used in [10], our algorithms find the unique faults with 100% probability. For the single-word (16-bit) fault model, the effective fault rate is about 40% for both interleaved and non-interleaved implementations. This is because for single-word faults, most bits of the final digest \( H \) (and internal state \( \theta_{23}^{23} \)) will be polluted (more confusion), and therefore the difference between the signatures of two faults is less distinct.

As the results for non-interleaved and interleaved implementations are similar, and the methods for fault identification and \( \chi_{22}^{22} \) bits recovery are the same for these two implementations, the rest of this work presents only results on non-interleaved implementation, and focuses on the single-byte fault model.

6.4.2 \( \chi_{22}^{22} \) Bits Recovery

Previous sections introduce the algorithms to derive fault signatures and use the signatures to infer the injected fault information. In this section, we describe the algorithms to recover bits of the internal state \( \chi_{22}^{22} \).

As demonstrated in Figure 6.4 and Section 6.3.2.2, each bit of \( FS_{\chi_{22}^{22}} \) may involve one or several bits of \( \chi_{22}^{22} \). Once the unique fault value at a certain position is identified, all the ‘x’ bits in the \( FS_{\chi_{22}^{22}} \) are known to be zero or one. First, those bits that only depend on a single bit of \( \chi_{22}^{22} \) are checked to recover the corresponding \( \chi_{22}^{22} \) bits. Then these newly recovered \( \chi_{22}^{22} \) bits are used in those signature bits that depend on multiple \( \chi_{22}^{22} \) bits to recover other bits.

Note that for each single-bit fault injected at \( \theta_{i}^{22} \), 22 bits of \( \chi_{i}^{22} \) can be recovered. With a multi-bit fault (\( n \)-bit) injected at \( \theta_{i}^{22} \), up to \( 22 \times n \) bits can be recovered. However, the \( \theta^{23} \) operation may cancel some \( \chi_{i}^{22} \) bits by the XOR operation, and the number of \( \chi_{i}^{22} \) bits that can be recovered by an \( n \)-bit fault is at most \( 22 \times n \).

For fault \( 1 \leq F \leq 255 \) on a byte, the distribution of the number of flipped \( \theta_{i}^{22} \) bits (i.e., \( n \)) is shown in Figure 6.5(a). We conduct an experiment to find the average number of bits recovered by each fault injected. We randomly generate \( 10^5 \) messages, and inject random faults at random positions, and count the recovered \( \chi_{i}^{22} \) bits for each fault injected. The corresponding results are shown in Figure 6.5(b), in which the x-axis is number of recovered \( \chi_{i}^{22} \) bits, and the y-axis is the corresponding ratio of faults among all 255 faults.

We define the average number of recovered bits for each injected fault as \( \alpha \), and the simulation results of Figure 6.5(b) show that \( \alpha \) is about 74.97 for randomly injected single-byte faults. We
assume the \( l \)-th fault can recover \( \omega_l \) new bits of \( \chi_i^{22} \) that have not been recovered by the previous \( l - 1 \) faults. We denote the total number of \( \chi_i^{22} \) bits recovered by the first \( l \) injected faults as \( \Omega_l \):

\[
\Omega_l = \sum_{j=1}^{l} \omega_j. \quad (6.5)
\]

For the \( l \)-th fault, the previous \( l - 1 \) faults have already recovered \( \Omega_{l-1} \) bits of \( \chi_i^{22} \), thus the ratio of unrecovered bits can be denoted as \( \frac{1600 - \Omega_{l-1}}{1600} \). For random messages and randomly injected faults, we can assume that these \( 1600 - \Omega_{l-1} \) bits are randomly distributed in the 1,600-bit \( \chi_i^{22} \) state. For SHA-3, we can simplify this problem by assuming that the probability for each bit to be recovered is equal. Thus, for the \( \alpha \) bits of \( \chi_i^{22} \) recovered by the \( l \)-th fault, the number of \( \chi_i^{22} \) bits that have not been recovered by previous \( l - 1 \) faults can be denoted as:

\[
\omega_l = \frac{1600 - \Omega_{l-1}}{1600} \cdot \alpha \quad (6.6)
\]

For the first injected fault, there will be no collision and thus \( \omega_1 = \alpha \), which is 74.97 for randomly injected single-byte fault according to the results in Figure 6.5. Thus we can plug this result into (6.5) and (6.6) to emulate the attack process and show the theoretical result in Figure 6.6, where the x-axis is the number of random faults injected, and the y-axis is the corresponding total number of \( \chi_i^{22} \) bits recovered. To simulate the attacks on SHA-3, we randomly generate \( 10^5 \) messages, and inject 200 random faults at random positions for each message. The experimental attack results are also shown in Figure 6.6.

Figure 6.6 shows that the theoretical result matches the experimental result very well. It also shows that by injecting random errors into \( \theta_i^{22} \), the attacker can recover all the 1,600 bits of \( \chi_i^{22} \)
using about 120 random faults. Comparing with the attacks on SHA3-384/512 proposed in [10], our method needs much smaller number of faults to recover the whole internal state under relaxed fault models. We also test the theoretical model under random single-bit fault model used in [10], and the experimental result and theoretical result match very well, shown in Figure 6.12 in Section 6.6.

6.5 Differential Fault Analysis of SHA3-224 and SHA3-256

Section 6.4 shows the method to identify the injected faults and to recover the internal state $\chi_2^{22}$ bits for SHA3-384 and SHA3-512. However, these methods cannot be applied to SHA3-224 and SHA3-256 directly, where the $\chi^{23}$ operation cannot be inverted to obtain the input rows with each output row only partially known. In this section, we show the method to recover $\chi_3^{23}$ bits first, and then use these intermediate state bits to recover $\chi_2^{22}$ of SHA3-224 and SHA3-256. We propose two improved attacks, presented in Section 6.5.2 and Section 6.5.3 correspondingly.

6.5.1 Basic Attacks of SHA3-224 and SHA3-256

For SHA3-224 and SHA3-256, only partial bottom plane of the hash output is observable, i.e., no more than four bits in each row of $\chi_0^{23}$ on the bottom plane are known. In Section 6.5.1.1 we show that with limited information, part of $\chi_3^{23}$ on the bottom plane can still be recovered from the observable digest. Attack details using the recovered $\chi_3^{23}$ information will be presented in Section 6.5.1.2.
CHAPTER 6. DIFFERENTIAL FAULT ANALYSIS OF SHA-3

6.5.1.1 \( \chi_i^{23} \) Bits Recovery from the Observable Digest

For simplicity, we use one row in \( \chi_i^{23} \) operation as an example here. We express the input bits \((a_i, b_i, c_i, d_i, e_i)\) as functions over the output bits \((a_o, b_o, c_o, d_o, e_o)\) through \( \chi^{-1} \) operation as:

\[
\begin{align*}
    a_i &= a_o \oplus b_o \cdot (e_o \oplus c_o \oplus e_o \cdot d_o) \\
    b_i &= b_o \oplus c_o \cdot (a_o \oplus d_o \oplus a_o \cdot e_o) \\
    c_i &= c_o \oplus d_o \cdot (b_o \oplus e_o \oplus b_o \cdot a_o) \\
    d_i &= d_o \oplus c_o \cdot (d_o \oplus b_o \oplus d_o \cdot c_o) \\
    e_i &= e_o \oplus c_o \cdot (d_o \oplus b_o \oplus d_o \cdot c_o)
\end{align*}
\] (6.7)

For SHA3-256, for each row, bit \( e_o \) is unknown while \((a_o, b_o, c_o, d_o)\) are observable by the attacker; for SHA3-224, bit \( e_o \) is unknown for the first 32 rows while both \( d_o \) and \( e_o \) are unknown for the remaining 32 rows. For the equations in (6.7), we identify the cases where the input bits are independent of the unknown output bits, and have the following observations for SHA3-256:

- For \( a_i \), if \( d_o = 1 \), \( a_i = a_o \oplus b_o \cdot c_o \); if \( b_o = 1 \), \( a_i = a_o \). For both situations, the attacker can retrieve \( a_i \) without knowledge of \( e_o \). The probability of \( d_o = 1 \) and the probability of \( b_o = 1 \) are 0.5 respectively, and thus the total probability of \( d_o = 1 \) or \( b_o = 1 \) is 0.75, which means that the value of \( a_i \) can be recovered with a probability of 0.75.

- For \( b_i \), if \( a_o = 0 \), \( b_i = b_o \oplus c_o \cdot d_o \); if \( c_o = 1 \), \( b_i = b_o \). Similarly, the probability of recovering \( b_i \) with unknown \( e_o \) is also 0.75.

- For \( c_i \), if \( d_o = 1 \), \( c_i = c_o \), and the probability of recovering \( c_i \) is 0.5.

- For \( d_i \), if \( c_o \oplus a_o \oplus c_o \cdot b_o = 0 \), \( d_i = d_o \), and the probability of recovering \( d_i \) is 0.5.

- The value of \( e_i \) always depends on \( e_o \), and the attacker cannot retrieve \( e_i \) without knowledge of \( e_o \).

In conclusion, for SHA3-256, the attacker can recover the bits in the first and second lanes of the bottom plan of \( \chi_i^{23} \) with 0.75 probability, and the bits in the third and fourth lane with 0.5 probability. In total, the attacker can recover 160 bits of \( \chi_i^{23} \) theoretically. Similarly, for SHA3-224, the attacker can use the same method to recover 112 bits of \( \chi_i^{23} \) theoretically.

We propose a practical method to recover \( \chi_i^{23} \) bits. For the same example shown in (6.7), while \( a_o, b_o, c_o, d_o \) are observable by the attacker, the unknown \( e_o \) can only be either 0 or 1, then we can make guesses of both situations and write them as \( \text{row}_o^0 = \{a_o, b_o, c_o, d_o, 0\} \) and \( \text{row}_o^1 = \{a_o, b_o, c_o, d_o, 1\} \).
For both situations, we can calculate the corresponding input \( r_{0i}, r_{1i} \) using \( \chi \) inversion operation:

\[
\begin{align*}
\{a_0^i, b_0^i, c_0^i, d_0^i, e_0^i\} &= \chi^{-1}\{a_0, b_0, c_0, d_0, 0\} \\
\{a_1^i, b_1^i, c_1^i, d_1^i, e_1^i\} &= \chi^{-1}\{a_0, b_0, c_0, d_0, 1\}.
\end{align*}
\] (6.8)

Take bit \( a_i \) as an example here, the value of \( a_i \) can only be \( a_0^i \) or \( a_1^i \):

1. If \( a_0^i = a_1^i \), then the value of \( a_i \) does not depend on the value of \( e_o \) and this is the correct recovered value for \( a_i \);
2. If \( a_0^i \neq a_1^i \), then the value of \( a_i \) depends on the value of \( e_o \), and attacker cannot recover \( a_i \) without knowing \( e_i \).

We implement this method for both SHA3-224 and SHA3-256 in C++, and randomly generate \( 10^5 \) input messages for each of them. Results show that the proposed algorithm can correctly recover 160.12 bits of \( \chi_2^{23} \) for SHA3-256 and 111.84 bits of \( \chi_2^{23} \) for SHA3-224 on average for these \( 10^5 \) trials, which conform to the theoretical results given in the previous section.

Using the above method, the attacker can recover part of the \( \chi_2^{23} \) bits in the bottom plane from the original digest \( H \), and faulty \( \chi_2^{23} \) bits for faulty digest \( H' \). Using the recovered \( \chi_2^{23}(X, 0, Z) \) and \( \chi_2^{23}(X, 0, Z) \) bits, the attacker can calculate the corresponding \( \Delta\chi_2^{23}(X, 0, Z) \) bits. Note that for SHA3-256, the attacker can recover about 160 bits of both \( \chi_2^{23}(X, 0, Z) \) and \( \chi_2^{23}(X, 0, Z) \) on average, but the recovered \( \chi_2^{23} \) and \( \chi_2^{23} \) may have different locations, and therefore the attacker will recover fewer than 160 differential bits of \( \Delta\chi_2^{23}(X, 0, Z) \). The simulation results show that attacker can recover 136.42 bits of \( \Delta\chi_2^{23} \) for SHA3-256, and 93.68 bits for SHA3-224 on average for \( 10^5 \) trials, with more details presented in Table 6.2.

### 6.5.1.2 Injected Fault Identification and \( \chi_2^{22} \) Bits Recovery

Using the algorithms in Section 6.5.1.1, the attacker recovers some bits of \( \Delta\chi_2^{23}(X, 0, Z) \) from the observable output, instead of 320 bits as in other two modes, SHA3-384 and SHA4-512. We use the same method presented before in Section 6.4.1 to match the recovered \( \Delta\chi_2^{23}(X, 0, Z) \) bits against the fault signatures, and apply the same constraints of (6.4) to identify the injected fault. Results show that for SHA3-256, the attacker can uniquely identify the fault with a probability 66.61\%. For SHA3-224, the effective fault ratio is 30.67\% instead. The results are shown in Table 6.2.

With the injected fault identified, we use the same method to recover the internal state bits of SHA3-224/256. Simulation results are in Figure 6.7. It shows that attacks on SHA3-224 and SHA3-256 are much less efficient than attacks on the other two modes with the digest length higher.
Table 6.2: Simulation results for SHA3-224 and SHA3-256 with fault injected at $\theta^2_i$

<table>
<thead>
<tr>
<th></th>
<th>Number of bits recovered</th>
<th>Effective fault ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\chi^2_1$</td>
<td>$\Delta\chi^2_1$</td>
</tr>
<tr>
<td>SHA3-224</td>
<td>111.84</td>
<td>93.68</td>
</tr>
<tr>
<td>SHA3-256</td>
<td>160.12</td>
<td>136.42</td>
</tr>
<tr>
<td>SHA3-384/512</td>
<td>320</td>
<td>320</td>
</tr>
</tbody>
</table>

than 320. The shorter the digest length, the fewer bits of $FS_{\chi^2_1}$ and $\Delta\chi^2_1$ are available for internal bits recovery, and therefore each effective fault injection recovers fewer bits of $\chi^2_1$. Nevertheless, with more effective faults injected to recover groups of bits one by one, these two modes are also susceptible to DFA.

Figure 6.7: Number of recovered $\chi^i_2$ bits for different number of effective faults

In next sections, we propose two methods to improve the fault analysis attacks of SHA3-224 and SHA3-256.

6.5.2 Improve the Attacks by Using both $FS_{\chi^2_1}$ and $FS_{\chi^2_2}$

Previous methods all select $\chi^2_1$ as the comparison point for fault signatures and observable differential digest. However, for SHA3-224 and SHA3-256 ($d = 224$ and 256 respectively), fewer than $d$ bits of $\Delta\chi^2_1$ are available to the attacker and this will cause information loss in attacks. We find that the fault signature at $\chi^2_1$, $FS_{\chi^2_1}$, if used, can help to extract extra $\chi^2_1$ bits. This is because all $d$ bits of $\Delta\chi^2_0$ are available for SHA3-$d$ function, $FS_{\chi^2_2}$ can be used with $\Delta\chi^2_0$ to
CHAPTER 6. DIFFERENTIAL FAULT ANALYSIS OF SHA-3

build constraints similar as in (6.4). We show in this section that with the introduction of $FS_{\chi^2_3}$ into attacks, both the effective fault ratio and the number of $\chi^2_2$ bits recovered by each effective fault increase.

We first present the construction of $FS_{\chi^2_3}$ in Section 6.5.2.1 and then give the results of fault identification and internal state recovery in Section 6.5.2.2.

6.5.2.1 Fault Signature $FS_{\chi^2_3}$ Construction

We have known that the fault propagation at $\chi^2_3$, $FS_{\chi^2_3}$, are dependent on both the fault and internal state bits. Through one more step of $\chi^2_3$ operation on rows, $a_o = a_i \oplus (\overline{b_i} \cdot c_i)$, the fault propagation function is:

$$
\Delta a_o = \Delta a_i \oplus \Delta b_i \cdot c_i \oplus (1 \oplus b_i) \cdot \Delta c_i \oplus \Delta b_i \cdot \Delta c_i. \tag{6.9}
$$

Thus $FS_{\chi^2_3}(x, y, z)$ can be denoted as (6.10), which involves $\chi^2_3$ bits in addition to fault signature bits $FS_{\chi^2_3}$.

$$
FS_{\chi^2_3}(x, y, z) = FS_{\chi^2_3}(x, y, z) \oplus FS_{\chi^2_3}(x+1, y, z) \cdot \chi^2_3(x+2, y, z) \\
\quad \oplus (1 \oplus \chi^2_3(x+1, y, z)) \cdot FS_{\chi^2_3}(x+2, y, z) \\
\quad \oplus FS_{\chi^2_3}(x+1, y, z) \cdot FS_{\chi^2_3}(x+2, y, z) \tag{6.10}
$$

Bits in $FS_{\chi^2_3}$ can be 0, 1, or a function over $\chi^2_2$ and $\chi^2_3$ bits. When comparing $\Delta \chi^2_0$ ($d$-bits) against $FS_{\chi^2_3}$, more constraints are considered so as to help identify the fault and recover the internal bits. This improves the attacks in two ways. First, some faults that cannot be identified by the basic attack before can now be uniquely identified, i.e., the effective fault rate increases. Second, with a certain effective fault, more $\chi^2_2$ bits can be recovered because $\Delta \chi^2_0$ and $FS_{\chi^2_3}$ are used.

Note here we cannot use $\Delta \chi^2_0$ and $FS_{\chi^2_3}$ only for attacks while excluding using $\Delta \chi^2_3$ and $FS_{\chi^2_3}$, this is because only part of $FS_{\chi^2_3}$ bits (instead of $d$ bits) are available. The construction of $FS_{\chi^2_3}$ requires knowledge of $\chi^2_3(x, y, z)$ bits which are only partially known. Thus attacker should combine information at $\chi^2_3 (\Delta \chi^2_3, FS_{\chi^2_3})$ and $\chi^2_0 (\Delta \chi^2_0, FS_{\chi^2_3})$ together for analysis.

6.5.2.2 Simulation Results

We construct the fault signature $FS_{\chi^2_3}$ and run simulations of the improved attacks. Simulation results show that the effective fault rate rises from 30.67% to 49.12% for SHA3-224, and from 53.28% to 78.73% for SHA3-256. The results of internal state recovery for SHA3-224 and SHA3-256 are shown in Figure 6.8.
6.5.3 Further Improve the Attacks by Injecting Faults at $\theta_i^{23}$

If the attacker can inject faults at multiple points, the attack will achieve higher efficiency. In previous sections, we assume that the attacker injects only faults at $\theta_i^{22}$. In this section, we explore another fault injection point, the last round input $\theta_i^{23}$, and use more faults to recover internal state bits more efficiently.

For the method proposed in Section 6.5.2, the effective fault ratio and the number of recovered $\chi_i^{22}$ bits by using the same number of effective injected faults are still lower than attacks on SHA3-384/512. The reason lies in the fact that the attacker can recover fewer bits of $\chi_i^{23}$ and $\Delta\chi_i^{23}$ for SHA3-224 and SHA3-256 than SHA3-384/512 (320 bits). To improve the attack efficiency, we propose to recover more $\chi_i^{23}$ bits first, by injecting faults at the last round input $\theta_i^{23}$ of SHA3-224 and SHA3-256.

We first present the details of recovering $\chi_i^{23}$ bits in Section 6.5.3.1 and then present the recovery of $\chi_i^{22}$ bits for this improved attack in Section 6.5.3.2.
6.5.3.1 Recovering More $\chi^{23}_i$ by Injecting Faults at $\theta^{23}_i$

To recover $\chi^{23}_i$ bits by injecting faults at $\theta^{23}_i$ and comparing fault signature and differential fault at $\chi^{23}_o$, we need to calculate the fault propagation from $\theta^{23}_i$ to $\chi^{23}_o$. These faults will propagate through $\theta$, $\rho$, $\pi$ and $\chi$ operations. The fault propagation process is exactly the same as in the penultimate round (from $\theta^{22}_i$ to $\chi^{22}_o$) as presented in Section 6.3.2.1. We denote the fault signature at $\chi^{23}_o$ for faults injected at $\theta^{23}_i$ as $FS^{*}\chi^{23}_o$ in this section.

Using the faults injected at $\theta^{23}_i$, the attacker can recover some remaining bits of $\chi^{23}_i$ on the bottom plane that have not been recovered using the algorithm in Section 6.5.1.1. Note here that for SHA3-256 and the first 32 rows of SHA3-224 (with four bits out of five bits of each output row on the bottom plane known and some input bits (not all the five) recovered), if the attacker recovers one bit $\chi^{23}_i (x, 0, z)$ that has not been recovered using the algorithm in Section 6.5.1.1, he can recover all the other unknown bits in this input row. For example, we assume $a^0_i \neq a^1_i$ in (6.8) and this bit has been recovered by injecting faults at $\theta^{23}_i$, then the attacker can know which assumption of $e_o$ is correct, and then recover all the five bits in this row. This method can be used for all 64 rows in the bottom plane of SHA3-256 and the first 32 rows of SHA3-224. In SHA3-224, the remaining rows ($\chi^{23}_i (X, 0, z), 32 \leq z < 63$) have two bits unknown, and these two bits can only be recovered by injecting faults at $\theta^{23}_i$ separately.

We use both fault signatures at $\chi^{23}_i (FS^{*}\chi^{23}_i)$ and $\chi^{23}_o (FS^{*}\chi^{23}_o)$ to identify the faults injected at $\theta^{23}_i$. Results show that for both SHA3-224 and SHA3-256, we can identify the correct fault injected at $\theta^{23}_i$ with about 20% probability. After identifying the correct faults injected at $\theta^{23}_i$, we can recover all 320 bits in the bottom plane of $\chi^{23}_i$ with multiple faults, and we present the recovery process in Figure 6.9.

![Figure 6.9: Recovery of $\chi^{23}_i$ bits in the bottom plane by injecting faults at $\theta^{23}_i$](image-url)
CHAPTER 6. DIFFERENTIAL FAULT ANALYSIS OF SHA-3

Figure 6.9 shows that for SHA3-256, the attacker can recover about 244 bits of $\chi_i^{23}$ using only five effective faults, compared with 160 bits when no faults injected at $\theta_i^{23}$ (Table 6.2). Similar results for SHA3-224 are also presented in Figure 6.9. Note that the attacker does not need to recover all the bits of $\chi_i^{23}$ in the bottom plane, he can recover part of $\chi_i^{23}$ to improve the efficiency of recovering $\chi_i^{22}$. We will show how the recovery of $\chi_i^{22}$ changes with the number of $\chi_i^{23}$ bits recovered in next section.

6.5.3.2 Simulation Results

With more $\chi_i^{23}$ bits recovered, the attacker can recover and construct more bits of $\Delta \chi_i^{23}$ and $FS_{\chi_i^{23}}$, and use them for attacks. For simplicity, we make the following assumptions for the attacker:

- The attacker first recovers part of $\chi_i^{23}$ bits using algorithm presented in Section 6.5.1.1 (with no fault injected at $\theta_i^{23}$).
- The attacker then injects faults at $\theta_i^{23}$ to recover the remaining bits of $\chi_i^{23}$.

For simplicity, we assume the attacker can randomly recover from 0 to 64 rows of $\chi_i^{23}$ using the algorithm in Section 6.5.1.1. For each number of recovered rows, we inject random faults at $\theta_i^{22}$ to calculate the effective fault ratio and show the results in Figure 6.10.

![Figure 6.10: Effective fault ratio with different number of $\chi_i^{23}$ rows recovered](image_url)

Figure 6.10 shows that with more rows of $\chi_i^{23}$ recovered, the attacker can identify the faults injected at $\theta_i^{22}$ with higher rate. For example, for SHA3-224, the effective fault ratio is 53.28% when only part of $\chi_i^{23}$ bottom plane has been recovered using the algorithm in Section 6.5.1.1 and this ratio rises to 88.34% when all 64 rows (320 bits) of $\chi_i^{23}$ bottom plane are recovered. Therefore,
by recovering more bits of $\chi_{i}^{23}$ in the bottom plane, the effective fault ratio will increase for both SHA3-224 and SHA3-256.

With knowledge of more $\chi_{i}^{23}$ bits, the attacker can build more equations for $\chi_{i}^{22}$ bits like in Figure 6.4 then the attacker can recover more $\chi_{i}^{22}$ bits for each injected fault on average. To verify the assumption, we assume the attacker can recover from 0 to 64 rows of $\chi_{i}^{23}$, and we run attacks on SHA3-224 and SHA3-256 to recover all the bits of $\chi_{i}^{22}$. We present the attack results on SHA3-224 with different numbers of rows recovered in Figure 6.11. For SHA3-256, the results are similar, and we will not present the details here.

![Figure 6.11: Number of recovered $\chi_{i}^{22}$ bits for different number of effective faults with a number of $\chi_{i}^{23}$ rows recovered, SHA3-224](image)

Figure 6.11 shows that the attacker needs smaller number of effective faults to recover all the bits of $\chi_{i}^{22}$ if he has recovered more rows of $\chi_{i}^{23}$. For example, if he has knowledge of the whole bottom plane of $\chi_{i}^{23}$, he can recover 1590 bits of $\chi_{i}^{22}$ using 110 effective random faults on average. For attacker who cannot inject fault into the last round input, using the improved method in Section 6.5.2.1, he can only recover about 1,412 bits (using 110 effective faults) instead.

In conclusion, by injecting faults at $\theta_{i}^{23}$ to recover more state bits of $\chi_{i}^{23}$, the attacker can identify the faults injected at $\theta_{i}^{22}$ with a higher rate. Consequently, the attacker needs a smaller number of effective faults to recover the same number of $\chi_{i}^{22}$ bits.

As the improved attack method in Section 6.5.2 does not require extra knowledge of the target system, it should be applied to DFA directly, while the improvement proposed in this section can be applied if the attacker has the ability to inject extra faults at $\theta_{i}^{23}$.
CHAPTER 6. DIFFERENTIAL FAULT ANALYSIS OF SHA-3

6.6 Optimized Attacks with Chosen Faults

In this section, we show an algorithm to optimize the attack if the injected faults (in terms of faulty byte location and fault value) can be controlled.

As shown in the previous sections, injecting an n-bit fault into $\theta_i^{22}$ can recover up to $22 \times n$ bits of $\chi_i^{22}$, and different faults may have overlapping $\chi_i^{22}$ bits. The most efficient attack method should avoid such overlap and use the smallest number of faults to recover the entire internal state. In this section, we show an optimized attack with the smallest number of faults into $\theta_i^{22}$ to recover all the bits of $\chi_i^{22}$. This optimization is based on SHA3-384/512 for simplicity, and it can be easily extended to SHA3-224 and SHA3-256.

We formulate this problem into a set covering problem [103]. We assume the attacker can inject $n$ different faults into $\theta_i^{22}$, denote them as $F = \{f_1, f_2, \cdots, f_n\}$, and the bits to recover is a universe $U$ for all the $\chi_i^{22}$ bits. For each fault $f_i$, it can be used to recover a subset of $U$ and we denote it as $U_i$, where $U_1, U_2, \cdots \subseteq U$, and we denote the cost to inject fault $f_i$ as $c_i$. The problem is to find a set of $I \subseteq \{1, 2, \cdots, n\}$ that minimize the total cost $\Sigma_{i \in I} c_i$ and $\cup_{i \in I} U_i = U$. The set covering problem is an NP-hard problem. We adopt a greedy heuristic to find solutions, given in Algorithm 4.

**Algorithm 4** Optimization of the fault injection attacks

**Input:** Element set $U$, subset set $U = \{U_1, U_2, \cdots, U_n\}$ and their costs

**Output:** Set cover $C$ with the minimum cost

1: $C \leftarrow \emptyset$
2: while $C \neq U$ do
3:     for all $i = 1$ to $n$ do
4:         if $U_i \in C$ then
5:             $\alpha_i = 0$;
6:         else
7:             $\alpha_i = \frac{|U_i - C|}{c_i}$
8:         end if
9:     end for
10:    Choose $k$ s.t. $\alpha_k = \text{MAX}(\{\alpha_i\})$;
11:    $C \leftarrow C \cup U_k$;
12: end while

Each time, the algorithm chooses a set that gives the highest gain (number of new bits), and we assume that the cost for each set/fault injection is the same (all $c_i = 1$). We use this algorithm to find the “best” fault injection solutions under our byte-level fault model, and the result is shown in Figure 6.12(a). It demonstrates that for DFA with byte-level faults, the attacker needs to inject at
least 17 faults to recover all the 1,600 bits of $\chi_{i}^{22}$, while random fault injection attacks require about 120 faults to retrieve all the bits for SHA3-384/512.

Figure 6.12: Optimized fault injection attacks at byte level

We also use the method and algorithm to analyze the DFA on SHA-3 under single-bit fault model used in [10]. Three curves are presented in Figure 6.12(b): the random fault injection experimental result, the random fault injection theoretical prediction, and the optimized attack using the greedy heuristic. Under the single-bit fault model, each fault injected at $\theta_{i}^{22}$ can be used to recover 22 bits of $\chi_{i}^{22}$, and thus $\alpha = 22$ for the theoretical result. Figure 6.12(b) shows that the theoretical result matches the simulation result for random fault injection very well. It also shows that the attacker needs to inject at least 129 single-bit selected faults to recover all the 1,600 bits of $\chi_{i}^{22}$, while he needs about 500 single-bit random faults to recover the state.

The optimized attack we present in this section is the upper bound of the proposed differential fault attacks. For SHA3-224 and SHA3-256, similar method can be used to find the upper bound of the attacks.
6.7 Discussions

6.7.1 SHA-3 Systems with Long Input Message

In previous sections, we simplify the analysis by assuming that only one $f$ function is involved for absorbing and squeezing. In this section, we extend the proposed attack by assuming that the size of the input message can be larger than the bitrate $r$. Meanwhile, we assume that the digest size can be larger than $r$, instead of just $d$ bits for SHA3-$d$ function. Therefore, multiple $f$ functions may be involved for absorbing and squeezing.

First of all, we assume that the digest size is still $d$-bit for SHA3-$d$ function:

- There are $n$ ($n \geq 1$) $f$ functions ($f_0 \cdots f_{n-1}$) involved for absorbing;
- There are $m$ ($m \geq 0$) $f$ functions ($f_n \cdots f_{n+m-1}$) involved for squeezing, as $r > d$ for all four SHA3-$d$ functions, no extra $f$ function will be involved for squeezing ($m = 0$);
- The attacker has control of the input message $P_0 \cdots P_{n-1}$ in MAC mode, while he has no access to the messages when SHA-3 is used in hash mode;
- The attacker can observe the digest $z_0 \cdots z_m$ in both hash and MAC mode.

For MAC-Keccak system, the attacker can inject faults into the penultimate round input of the last permutation $f_{n-1}$, and use the proposed method in this work to recover the input of $f_{n-1}$, $f_{n-1}(in)$. Combining with $P_{n-1}$, the attacker can recover $f_{n-2}(in)$. There are two possible situations for the key length $l_k$ here:

- If $l_k \leq r$, the attacker can iteratively recover $P_0$ which contains the key used for MAC, and therefore recover the secret key.
- If $l_k > r$, for example, all the bits of $P_0$ and part of $P_1$ bits are key bits, then the attacker can recover $f_1(in)$, which is

$$f_1(in) = f(P_0||0^c) \oplus (P_1||0^c), \quad (6.11)$$

then the attacker will be unable to recover the key bits contained in $P_0$ and $P_1$ directly. The attacker needs to make assumption of the key bits in $P_1$ so as to recover the key bits in $P_0$. The difficulty increases rapidly with the number of the key bits in $P_1$.

As the key size in MAC system is usually much smaller than $r$ (for example, 128 bits or 256 bits), the proposed DFA is a great threat for SHA-3 based MAC systems. We propose to increase the length of MAC key for higher resilience against fault injection attacks.
For SHA-3 system in general hash mode, the attacker will have no access to the input message $P_0 \cdots P_{n-1}$. Therefore, after recovering $f_{n-1}(in)$, he will be unable to further recover $f_{n-2}(out)$ without knowledge of $P_{n-1}$. Thus the attacker will be unable to recover the original input message in hash mode directly if the message length is greater than bitrate $r$.

For modified SHA-3 system which allows the digest size larger than $d$, there may be extra $f$ functions involved for squeezing, and the digest comes from multiple $z_i$. As all the $z_i$ are observable, $z_0$ will be used to attack $f_{n-1}$. The length of $z_0$ will be $r$-bit, which is 1,152, 1,088, 832 and 576-bit for SHA3-224, SHA3-256, SHA3-384 and SHA3-512 respectively. More bits than original (224, 256, 384 and 512-bit) are available, and the attacks become easier. In conclusion, the proposed DFA method is still applicable for SHA-3 systems which involve extra $f$ functions for squeezing.

### 6.7.2 Countermeasure against Differential Fault Analysis

As unprotected SHA-3 systems are vulnerable to DFAs, countermeasures should be added into SHA-3 systems to improve their resilience against DFA.

One method is to implement detection of system disturbance which are used to inject faults. For example, a clock glitch and power supply disturbance detection module \textsuperscript{[104,105]} can be inserted into SHA-3 implementations to detect fault injections.

Another kind of widely used countermeasures relies on some redundancy to check the integrity of the intermediate results, such that errors caused by injected faults will be detected. For example, parity checking codes are used to detect the injected faults in SHA-3 \textsuperscript{[106]}. Similarly, another copy of modified Keccak implementation can be introduced into the system for error detection \textsuperscript{[107,65]}. Such schemes can detect errors in the system caused by injected faults, and thus the attacker will have no access to the faulty results to conduct DFA.

### 6.8 Summary

In this chapter, we propose efficient DFA methods for all four modes of SHA-3 functions under relaxed single-byte fault models. Results show that our method can effectively identify the injected faults, and then recover the corresponding internal state bits for all four SHA-3 functions. Meanwhile, we also present the upper bound of the proposed attacks and extend the attacks to systems with input message longer than bitrate $r$. 

98
Chapter 7

Algebraic Fault Analysis of SHA-3

7.1 Introduction and Motivation

In last chapter, details of our DFA attacks on SHA-3 are provided. In this chapter, I will improve the DFA on SHA-3 by introducing algebraic techniques into analysis, and this attack method is Algebraic Fault Analysis (AFA) [108, 109, 110, 111, 112].

AFA converts the cipher operations and faults into algebraic equations, formulates the searching for the secret into a satisfiability problem, and recovers the secret using automatic tools such as SAT solvers with no need of complex fault propagation analysis. It has been used to improve the fault attacks on DES [113], Trivium [114], AES [115], LED [109, 110], and Piccolo [110].

In this work, we show that comparing with DFA on SHA-3, AFA on SHA-3 has the following advantages:

- **Efficiency.** With AFA, much fewer fault injections are needed to recover a whole internal state than with DFA.

- **Automation.** Unlike previous work of DFA on SHA-3, the AFA method does not require the tedious and error-prone manual fault propagation analysis.

- **Extensibility.** Our AFA method can be easily extended for any fault models (by changing the constraints in the formulation only), such as single word 16-bit and 32-bit fault models.

Comparing with our DFA work in previous chapter, the contribution of this work is as following:

- We present and optimize the SAT equations construction of SHA-3.

- We introduce the algebraic fault analysis to conquer all SHA-3 functions.
We further propose two novel optimization methods for AFA on SHA-3, striving for higher efficiency under more relaxed fault models.

The rest of this chapter is organized as follows. In Section 7.2, we present the fault models used in this work. In Section 7.3, we present the details of basic AFA on SHA-3. In Section 7.4, we extend the attack method to more relaxed fault models by involving more than one faulty digests every time to achieve tighter constraints. In Section 7.5, we further extend the proposed attacks by injecting faults at multiple locations, and significantly improves the effectiveness of the AFA method. In Section 7.6, we summarize this work and give insights about future works. Finally, we conclude this paper in Section 7.7.

7.2 Fault Models

We start with the same attack setting as DFA, in which the fault injection point is $\theta_2^2$, and the attack target is recovering the entire internal state of $\chi_2^2$ (1,600 bits). Specifically, the fault model adopted for AFA work is:

1. Depending on the data structure of the underlying architecture, the attacker can inject faults into one unit of the penultimate round input $\theta_2^2$, where the unit size can be one-byte, 16-bit, or 32-bit.
2. The attacker has no control on either the position (on which unit) or the value of the injected faults.
3. The attacker can observe only the correct and faulty SHA-3 digests, $H$ and $H'$, which are $d$ bits for SHA3-$d$ function.
4. The attacker can inject different random faults into the same input message to get different faulty outputs.

7.3 Algebraic Fault Analysis of SHA-3

7.3.1 Overview of Algebraic Fault Analysis on SHA-3

The key idea of AFA is to transform the internal state recovery problem into a satisfiability problem, representing the target algorithms and operations using Boolean equations, and then use a SAT solver to find solutions for the variables which contain secret information.
CHAPTER 7. ALGEBRAIC FAULT ANALYSIS OF SHA-3

First, we build a set of equations for the hash function for both correct and faulty executions. With the input of the penultimate round input $\theta_{22}^i$, the correct hash digest is:

$$H = \iota^{23} \circ \chi \circ \pi \circ \rho \circ \theta \circ \iota^{22} \circ \chi \circ \pi \circ \rho \circ \theta(\theta_{22}^i). \tag{7.1}$$

Denote the fault injected at $\theta_{22}^i$ as $\Delta \theta_{22}^i$, then the faulty digest is:

$$H' = \iota^{23} \circ \chi \circ \pi \circ \rho \circ \theta \circ \iota^{22} \circ \chi \circ \pi \circ \rho \circ \theta(\theta_{22}^i \oplus \Delta \theta_{22}^i). \tag{7.2}$$

Both $H$ in (7.1) and $H'$ in (7.2) have $d$ bits for SHA3-$d$ function. For DFA in [10] and [116], the attacker uses the differential of $H$ and $H'$ and reverses the $\chi$ operation to get the differential of $\chi^{23}_i$, $\Delta \chi^{23}_i$, for attack. In this work, with algebraic techniques we are able to make use of both $H$ and $H'$ directly, i.e., with more information.

We first build the equation set for (7.1) and (7.2), and construct constraints for the injected fault $\Delta \theta_{22}^i$. Then we show the recovery of $\chi^{22}_i$ bits using a SAT solver in Section 7.3.3. In this work, we use C++ API of CryptoMiniSat for the SAT problem formulation and solving [117]. All the simulations are run on a Ubuntu 14.04.3 system, with an Intel i7-6700 CPU and 32 GB memory.

7.3.2 SAT Problem Formulation

In this section, we formulate fault attack on SHA-3 into an SAT problem.

7.3.2.1 Construction of the Equation Set for Keccak

Thanks to the clear algebraic operations of Keccak, the construction of equations is straightforward for SHA-3. We can use 1,600 single-bit variables to denote the input $\theta_{22}^i$, and another 1,600 variables for the differential input (fault injected) $\Delta \theta_{22}^i$. Then we build equations for the operations in the last two rounds of SHA-3 for both the correct and faulty hashing processes based on (7.1) and (7.2). Finally two additional sets of $d$ equations can be used to represent the observed value of digest $H$ and $H'$, respectively.

For $\theta$ step, the only operation is XOR of state bits, which can be easily expressed in CryptoMiniSat. We introduce 320 variables to denote the $\theta$ compression results $\theta_c$, and then use another 1,600 bits to denote the $\theta$ operation output. Both $\rho$ and $\pi$ are bit permutation operations, which can be simply denoted in SAT. The step $\chi$ involves XOR, NOT and AND operations, and can be represented as:

$$\chi_o(x, y, z) = \chi_i(x, y, z) \oplus \chi_i(x + 2, y, z) \oplus \chi_i(x + 1, y, z) \cdot \chi_i(x + 2, y, z). \tag{7.3}$$
CHAPTER 7. ALGEBRAIC FAULT ANALYSIS OF SHA-3

We introduce another 1,600 variables \( \chi_{and}(x, y, z) \) to denote \( \chi_i(x + 1, y, z) \cdot \chi_i(x + 2, y, z) \). Then \( \chi \) operation can be represented as XOR of three variables. Finally, \( \iota \) can be denoted using XOR operation directly.

Some optimization can be introduced into the equation set construction. Such optimization can decrease the number of variables and thus increase the speed of SAT solver. We have the following optimization in this work:

- As \( \rho \) and \( \pi \) are bit permutation operations, instead of introducing new variables for outputs of operations \( \rho \) and \( \pi \), we combine the operations of \( \theta \), \( \rho \), and \( \pi \), and therefore eliminate the 3200 variables of \( \theta_o \) and \( \rho_o \).

- Since the attacker can only observe \( d \)-bit digests, (\( H \) and \( H' \)), there are only partial \( \chi_{23}^i \) state bits that would be involved to generate the digests. So for the last round \( \chi \) operation, we only construct equations corresponding to these bits. For example, only 640 bits of \( \chi_{23}^i \) are involved for digest generation for SHA3-512, while only 320 bits are involved for SHA3-224.

- Due to the linearity of operations \( \theta \), \( \rho \), and \( \pi \), we only write equations for the fault propagation process by the first three operations in the penultimate round, i.e., generating \( \Delta \chi_{22}^i \) from variables \( \Delta \theta_{22}^i \). The last two rounds would be represented to start from \( \chi_{22}^i \) and \( \Delta \chi_{22}^i \) as following:

\[
\begin{align*}
    H &= \iota_{23}^i \circ \chi \circ \pi \circ \rho \circ \theta \circ \iota_{22}^i \circ \chi (\chi_{22}^i) \\
    H' &= \iota_{23}^i \circ \chi \circ \pi \circ \rho \circ \theta \circ \iota_{22}^i \circ \chi (\chi_{22}^i \oplus \Delta \chi_{22}^i), \\
    \Delta \chi_{22}^i &= \pi \circ \rho \circ \theta (\Delta \theta_{22}^i)
\end{align*}
\]  

instead of (7.1) and (7.2).

We present all the variables for SAT equation construction of SHA3-512 in Table 7.1. For example, there are 1600 variables required for the clean \( \chi_{22}^i \), independent 1600 fault variables \( \Delta \chi_{22}^i \), and corresponding 1600 variables for the faulty state \( \chi_{22}^i = \chi_{22}^i \oplus \Delta \chi_{22}^i \). Only part of state bits, \( \chi_{23}^i \), \( \chi_{23, and}^i \) and \( \iota_{23, o}^i \), are required due to the limited digests. Overall, there are 17,088 variables in total, comparing with 44,480 variables without optimization. Thus the above optimization method saves 61.6% variables in total. Similarly, for other three modes of SHA-3, different number of variables are saved for SAT equations.

Note that even without the above optimization, the AFA method proposed in this section needs only several seconds to recover the \( \chi_{22}^i \) bits under single-byte fault model. The proposed optimization will help to improve the speed of SAT solver under more relaxed fault models, which will be presented in following sections.
### Chapter 7: Algebraic Fault Analysis of SHA-3

#### Table 7.1: Variables for SAT equations of SHA3-512

<table>
<thead>
<tr>
<th>Name</th>
<th>$x_i^{22}$</th>
<th>$x_{\text{and}}^{22}$</th>
<th>$\theta_i^{22}$</th>
<th>$\theta_i^{23}$</th>
<th>$x_i^{23}$</th>
<th>$x_{\text{and}}^{23}$</th>
<th>$\iota_o^{23}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO.</td>
<td>1600</td>
<td>1600</td>
<td>1600</td>
<td>320</td>
<td>640</td>
<td>512</td>
<td>512</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>$x_i^{22}$</th>
<th>$x_{\text{and}}^{22}$</th>
<th>$\theta_i^{23}$</th>
<th>$\theta_i^{23}$</th>
<th>$x_i^{23}$</th>
<th>$x_{\text{and}}^{23}$</th>
<th>$\iota_o^{23}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO.</td>
<td>1600</td>
<td>1600</td>
<td>1600</td>
<td>320</td>
<td>640</td>
<td>512</td>
<td>512</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>$\Delta \theta_i^{22}$</th>
<th>$\Delta x_i^{22}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO.</td>
<td>1600</td>
<td>320</td>
</tr>
</tbody>
</table>

#### 7.3.2.2 Constraints for the Injected Fault

The fault model we adopt will be represented as constraints in the SAT problem. Under the single-byte fault model, we use $D = \{d_1, d_2, \ldots, d_{200}\}$ to denote $\Delta \theta_i^{22}$, which is 200 bytes, and $d_i$ ($1 \leq i \leq 200$) is one byte of $\Delta \theta_i^{22}$. We use $d_i^j$ to represent the $j^{th}$ bit of $d_i$, in which $j \in \{1, 2, \ldots, 8\}$.

A one-bit variable $c_i$ is introduced to represent whether the $i^{th}$ byte of $\theta_i^{22}$ is corrupted or not:

$$c_i = d_i^1 \lor d_i^2 \lor \cdots \lor d_i^8.$$  \hspace{1cm} (7.5)

We have the following statement based on the assumption of a random byte-based fault model:

1. One and only one byte out of 200 bytes of $D$ has non-zero value, because only one byte of $\theta_i^{22}$ is corrupted.

2. If $d_i$ is corrupted, $c_i = 1$, otherwise $c_i = 0$.

Then we have the following constraints which will make sure that one and only one byte of $\theta_i^{22}$ is corrupted:

$$\begin{cases}
    c_1 \lor c_2 \cdots \lor c_{200} = 1 \\
    \bar{c}_i \lor \bar{c}_j = 1, 1 \leq i < j \leq 200
\end{cases}.$$  \hspace{1cm} (7.6)

Another 1600 variables are introduced to represent the constraints in (7.5) and (7.6) under single-byte fault model. The constraints for other fault models, such as single word 16-bit fault model and 32-bit fault model, are similar as described above.

#### 7.3.3 Recovery of Internal State Bits

After constructing the formulations of SHA-3, the next step is to use an SAT solver to find the solution under the constraints. The goal of our attack is to recover the total 1600 bits of the internal...
CHAPTER 7. ALGEBRAIC FAULT ANALYSIS OF SHA-3

state $\chi_{22}$. Previous work show that each fault can be used to recover only part of $\chi_{22}^i$, instead of the whole internal state \[116\]. In \[116\], the authors need to build complex fault signatures and identify the injected fault first, and then locate the corresponding target bits in $\chi_{22}^i$. In this work, we rely on an SAT solver to recover $\chi_{22}^i$ bits directly without figuring out the fault.

We denote the set of all the 1600 bits of $\chi_{22}^i$ as $\{\chi_{22}^i\}$, and the set of $\chi_{22}^i$ bits that can be recovered as $T$. Thus elements in $T$ should only have one solution, while other elements in $\{\chi_{22}^i\}$ can have multiple solutions under the constraints. We denote the solution by SAT solver as $S$.

Without knowledge of the injected fault and complex fault propagation details, we can use the following method to find $T$ and recover the corresponding $\chi_{22}^i$ bits:

1. Initialize $T$ as the set of all 1600 $\chi_{22}^i$ bits, $\{\chi_{22}^i\}$.
2. Use the SAT solver to find a solution for all elements in $T$.
3. Block the above solution and add this constraint into the SAT solver.
4. Use the SAT solver to find another solution for the elements in $T$.
5. For all the elements in $T$, if they have different values from previous solution, remove these elements from $T$.
6. Repeat step 3-5 until there is no new solutions, and the remaining bits in $T$ are recoverable while their values given by the SAT solver.

We present the above algorithm in Algorithm 5.

In this work, we denote the above method as Method I. We check Method I on all four SHA-3 functions under single-byte fault model and 16-bit fault model, and present their results in Figure 7.1. The proposed method works for all the four modes of SHA-3 under the single-byte fault model. For 16-bit fault model, the method succeeds for SHA3-384 and SHA3-512 while cannot apply to the other two modes with shorter digest, SHA3-224 and SHA3-256, effectively. Note DFA does not work at all for the 16-bit fault model.

Figure 7.1a and Figure 7.1c show the distribution of number of $\chi_{22}^i$ bits recovered by one fault injection under different fault models and for different modes of SHA-3. As the digest size, $d$, increases, the distribution moves towards right, i.e., more bits recovered on average. This is because the larger $d$, more bits of clean and faulty digest for SHA-3 function are observable, which provide tighter constraints for the SAT solver. As each fault recovers more bits, the SHA3-$d$ function with greater $d$ needs fewer faults to recover the whole internal state, as shown in Figure 7.1b and Figure 7.1d.
CHAPTER 7. ALGEBRAIC FAULT ANALYSIS OF SHA-3

Algorithm 5 Internal State Bits Recovery

1: \( S = \emptyset, S' = \emptyset, T = \{ \chi_{i}^{22} \} \)
2: while (solver.solve()==true) do
3: \( S\).clear();
4: For all \( \chi_{i}^{22} \) bits in \( T \), store their newly found results in \( S \)
5: for all \( \chi_{i}^{22} \in S \) and \( S' \) do
6: \( \text{if } S \text{ and } S' \text{ have different value for } \chi_{i}^{22} \text{ bit then} \)
7: Delete this \( \chi_{i}^{22} \) bit from \( T \)
8: Delete this \( \chi_{i}^{22} \) bit value from \( S \)
9: \( \text{end if} \)
10: end for
11: Block(\( S \));
12: \( S' = S \);
13: end while
14: number of recovered bits = size(\( T \))
15: Recovered \( \chi_{i}^{22} \) values given by \( S \)

We also compare the performance results of AFA with DFA in Table 7.2 in terms of number of required faults and attacking time. The DFA results are from [116]. It shows that AFA requires much fewer faults than DFA to recover \( \chi_{i}^{22} \), and the time consumption of AFA is less than four minutes for all four SHA-3 functions under the single-byte fault model, and SHA3-384 and SHA3-512 under the 16-bit fault model.

Table 7.2: Compare Attach Method I with DFA

<table>
<thead>
<tr>
<th></th>
<th>8-bit fault model</th>
<th>16-bit fault model</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SHA3-224</td>
<td>SHA3-256</td>
</tr>
<tr>
<td>AFA Time</td>
<td>232s</td>
<td>184s</td>
</tr>
<tr>
<td>Faults</td>
<td>69</td>
<td>65</td>
</tr>
<tr>
<td>DFA Time</td>
<td>&gt; 60s</td>
<td>34s</td>
</tr>
<tr>
<td>Faults</td>
<td>&gt; 400</td>
<td>194</td>
</tr>
</tbody>
</table>

The results demonstrate that the AFA attack method proposed in this work has the following advantages:

- It does not require knowledge of complex fault propagation in SHA-3, and therefore can be algorithm agnostic.
It makes use of every single injected fault for internal bits recovery, while the previous attacks can only make use of effective faults that can be uniquely identified based on the fault signatures. For example, in DFA the effective fault ratio is about $49.12\%$ for SHA3-224 and $78.7\%$ for SHA3-256. While in AFA, our method can make use of every single injected fault for bits recovery. This affects the attack efficiency.

Our proposed method needs much fewer faults to recover the internal state of $\chi_i^{22}$ for all four SHA-3 functions. For example, the AFA attack needs about only 69 faults for 1600 bits, while the previous DFA can recover about 818 bits using 69 faults (34 effective faults with effective fault ratio 49.12%) \[116\].

Our attack method scales under the 16-bit fault model for SHA3-384 and SHA3-512, while the computational complexity for DFA is too high to succeed. For 16-bit faults (6,553,500 different faults), the memory usage and computation time for fault propagation are beyond the
CHAPTER 7. ALGEBRAIC FAULT ANALYSIS OF SHA-3

\[ \mathbb{H}'[k] = \iota^{23} \circ \chi \circ \pi \circ \rho \circ \theta \circ \iota^{22} \circ \chi \circ \pi \circ \rho \circ \theta (\theta_i^{22} \oplus \Delta \theta_i^{22}[k]) \quad (7.7) \]

power of our workstation.

Note here Method I is no longer effective for SHA3-224 and SHA3-256 under 16-bit fault model, and all the four SHA-3 modes under 32-bit or more relaxed fault models. This is because the constraints will be looser under more relaxed fault models, and also for SHA-3 functions with shorter digest. This happens for DFA too, where the attackers can no longer identify the injected faults effectively [116]. The reason lies in the fact that for more relaxed fault models, more digest bits will be affected by the injected fault, and the fault signatures between different faults will have smaller difference. Thus the attacker will be no longer able to identify the injected faults effectively.

To apply AFA under more relaxed fault models, we need to devise effective schemes to construct tighter constraints for the SHA-3 SAT problem. Next we propose two improvement methods.

7.4 Improvement - Utilize Multiple Faults Together for Tighter Constraints

In this section, we propose to utilize multiple faults and the group of faulty digests \((\Delta \theta_i^{22}, H')\) to constraint the target system further for higher effectiveness.

7.4.1 Tighter Constraints Construction

For the original input message, the attacker can inject different faults at \(\theta_i^{22}\) to get multiple faulty digest \(H'\). Instead of using a pair of \(\{H, H'\}\) to recover internal state bits each time independently, now we are using the original \(H\) and a tuple of \(n H'\) (assume we consider \(n\) fault injections together). Denote the set of \(H'\) as \(\mathbb{H}'\), and the corresponding set of \(\Delta \theta_i^{22}\) as \(\{\Delta \theta_i^{22}\}\). Then for the \(k\)-th fault injection, the fault propagation can be denoted as \((7.7)\).

For each bit of \(\chi_i^{22}\), it may not be uniquely defined by the constraints of a single fault injection \((\Delta \theta_i^{22}, H')\). We find that when multiple faults \((\Delta \theta_i^{22}, H')\) work together to constraint the internal state bits as in \((7.7)\), some \(\chi_i^{22}\) bits can be uniquely identified and recovered under more relaxed fault models. In this work, we denote the improved method proposed in this section as Method II.

7.4.2 Attack Results

For the proposed improvement, a group of fault injections \((\Delta \theta_i^{22}, H')\) will be considered as constraints at the same time, and this will cause a significant increase in the number of variables,
equations, and therefore the SAT solver running time. We only report the attack results for the combinations of SHA-3 functions and fault models that can be solved within reasonable time in this work, for example, the attack results of SHA3-256 under 16-bit fault model and SHA3-512 under 32-bit fault model. We can expect better results with further attack method improvement, shown in next section, and also better SAT solvers and more computation power in the future.

How many fault injections should be considered together to recover internal bits remains a question. There is a balance between the SAT solver running time and number of recovered \( \chi_i^{22} \) bits here. If the number of faults (\( \Delta \theta_i^{22}, H' \)) in a group is small, the constraints are still not tight enough and no bits are recovered. However, with a large number of (\( \Delta \theta_i^{22}, H' \)), the chance of recovering bits is higher, but the number of variables increases rapidly, which may thwart the SAT solver from finding a solution in reasonable time.

Our simulation results suggest that a group of four faults (\( \Delta \theta_i^{22}, H' \)) together works for SHA3-256 under the 16-bit fault model, and a group of two faults together for SHA3-512 under the 32-bit fault model. The results of distributions of the number of recovered \( \chi_i^{22} \) bits are presented in Figure 7.2. It shows that not every trial of the proposed attacks can effectively recover \( \chi_i^{22} \) bits under more relaxed fault models. For example, the proposed method can recover more than 200 \( \chi_i^{22} \) bits for SHA3-256 under 16-bit fault model with about 36.04% probability; and it can recover more than 200 \( \chi_i^{22} \) bits for SHA3-512 under 32-bit fault model with about 54.14% probability. As one trial of the proposed method does not guarantee to give useful results for 100% probability, we can run several trials of the above attacks in parallel with each trial involving different groups of (\( \Delta \theta_i^{22}, H' \)) for constraints, such that at least one trial will recover some \( \chi_i^{22} \) bits with a high probability. For example, if we run five different attacks on SHA3-256 in parallel, the probability of recovering more than 200 \( \chi_i^{22} \) bits will increase significantly to about 90%.

After recovering the first group of \( \chi_i^{22} \) bits using multiple faults, we can add the recovered \( \chi_i^{22} \) bits into constraints to facilitate the SAT solver seeking for the rest of state bits. For example, for SHA3-256, the first group of constraints can help to recover 467 \( \chi_i^{22} \) bits on average. With these recovered bits added into the constraints, the attacker can recover the whole \( \chi_i^{22} \) internal state within two minutes. Similarly, the first group of constraints can help to recover 486 \( \chi_i^{22} \) bits on average for SHA3-512 under 32-bit fault model. Note here after recovering the first group of \( \chi_i^{22} \) bits with multiple faults and adding them into constraints, the attacker can either involve one fault or multiple faults each time to recover the remaining \( \chi_i^{22} \) bits. For example, for SHA3-256 under the 16-bit fault model, our attack starts with four faults, and then every time two additional faults. For SHA3-512, the attack starts with two faults, and then one fault a time. The recovery process of the above attacks is shown in Figure 7.3. Attack Method II breaks SHA3-256 under 16-bit fault model using only 16
CHAPTER 7. ALGEBRAIC FAULT ANALYSIS OF SHA-3

Figure 7.2: Distribution of the number of recovered $\chi^2_i$ bits for SHA-3 functions using Method II faults, and the time consumption is about 608 seconds. It also breaks SHA3-512 under the 32-bit fault model using only seven faults, and the time consumption is 317 seconds.

Figure 7.3: Attack results under Method II

Figure 7.3 also shows that the attacker needs fewer faults to recover the whole internal state under more relaxed fault models. For example for SHA3-512, the attacker needs about 45 faults under the single-byte fault model, fewer than 10 faults for the 32-bit fault model. The reason is more bits are flipped at the fault injection point with the more relaxed fault model, which can can flip more internal state bits, and thus help to recover more $\chi^2_i$ bits on average.
CHAPTER 7. ALGEBRAIC FAULT ANALYSIS OF SHA-3

7.5 Further Improvement - Multiple Fault Injection Points

In Section 7.4, we extend the attack on SHA3-256 to the 16-bit fault model, and on SHA3-512 to the 32-bit fault model. The Method II proposed in Section 7.4 needs groups of \((\Delta \theta_i^{22}, H')\) for constraints at the same time. However, under the 16-bit fault model, it cannot break SHA3-224 yet; and under the 32-bit fault model, it does not work for the other three modes of SHA-3. What’s more, with multiple faults being considered simultaneously in the constraints for the SAT solver, the time consumption for the solving process increases rapidly. In this section, we will further improve the fault analysis on SHA-3 and extend it to more cases.

For both Method I and Method II, we choose only one fault injection point, \(\theta_i^{22}\), for internal state bits recovery. It has been shown that injecting faults at \(\theta_i^{22}\) can recover the whole internal state of \(\chi_i^{22}\), while faults at \(\theta_i^{23}\) can only recover part of \(\chi_i^{23}\) bits. However, these recovered \(\chi_i^{23}\) bits, once added into the constraints, can help to recover \(\chi_i^{22}\) more efficiently. We denote the attack method proposed in this section as Method III in this work.

7.5.1 Recovery of \(\chi_i^{23}\)

In previous sections, we recover \(\chi_i^{22}\) bits by injecting faults at \(\theta_i^{22}\) and representing faults as constraints for algorithm operations (7.1) and (7.2). As demonstrated in [116], by making use of \(H\), theoretically the attackers can recover 112 bits of \(\chi_i^{23}\) for SHA3-224, 160 bits of \(\chi_i^{23}\) for SHA3-256, and 320 bits for SHA3-384/512, all on the bottom plane by each fault injection. When we add an additional fault injection point \(\theta_i^{23}\), similarly some \(\chi_i^{23}\) bits can be recovered. We next examine the effectiveness of recovering \(\chi_i^{23}\) bits with faults injected at \(\theta_i^{23}\).

We revisit the fault propagation of \(\chi\) operation. It has been shown in [116] that by observing the clean and faulty output at \(\chi_o^{23}(x, y, z)\), the attacker can get knowledge of \(\chi_i^{23}(x + 1, y, z)\) and \(\chi_i^{23}(x + 2, y, z)\). We have the following observations:

- For SHA3-224 and SHA3-256, the attacker can recover the whole bottom plane of \(\chi_i^{23}\).
- For SHA3-384, the attacker can observe the bottom plane of the output and one lane of the next plane, \(\chi_o^{23}(0, 1, z)\), both clean and faulty ones. Therefore, it can recover two lanes on the second plane, \(\chi_i^{23}(1, 1, z)\) and \(\chi_i^{23}(2, 1, z)\), in addition to the bottom plane of \(\chi_i^{23}\). Then he can further combine the three lanes including the output \(\chi_o^{23}(0, 1, Z)\) and input \(\chi_i^{23}(1, 1, Z)\) and \(\chi_i^{23}(2, 1, Z)\) of the \(\chi\) operation, to recover \(\chi_i^{23}(0, 1, Z)\). Thus the attacker can recover at most 512 bits of \(\chi_i^{23}\) in total.
CHAPTER 7. ALGEBRAIC FAULT ANALYSIS OF SHA-3

For SHA3-512, the attacker can recover two planes (640 bits) of $\chi^{23}_i$, namely $\chi^{23}_i(X, 0, Z)$ and $\chi^{23}_i(X, 1, Z)$.

For any of the four SHA-3 mode, the attacker can recover the whole plane of $\chi^{23}_i(X, 0, Z)$ and part of $\chi^{23}_i(X, 1, Z)$. To implement the attacks with algebraic techniques, we build constraints for the fault $\Delta \theta^{23}_i$ injected at $\theta^{23}_i$, shown in (7.8), with the corresponding faulty digest denoted as $H^*$.

$$\begin{align*}
H &= \iota^{23} \circ \chi \circ \pi \circ \rho \circ \theta^{23}_i \\
H^* &= \iota^{23} \circ \chi \circ \pi \circ \rho \circ \theta^{23}_i \oplus \Delta \theta^{23}_i
\end{align*}$$ (7.8)

In this section, we are trying to extend AFA to SHA3-224/256 under 16-bit fault model, and SHA3-384/512 under 32-bit fault model using the above method. Based on the new constraints defined in (7.8), we run fault injection attacks on all four SHA-3 functions. In Table 7.3 we present the results of attack time and number of faults required to recover all the $\chi^{23}_i$ bits that can be recovered with the injected faults.

<table>
<thead>
<tr>
<th>$\chi^{23}_i$ bits</th>
<th>16-bit fault model</th>
<th>32-bit fault model</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHA3-224</td>
<td>320</td>
<td>512</td>
</tr>
<tr>
<td>SHA3-256</td>
<td>320</td>
<td>640</td>
</tr>
<tr>
<td>SHA3-384</td>
<td>512</td>
<td>8.63s</td>
</tr>
<tr>
<td>SHA3-512</td>
<td>640</td>
<td>8.63s</td>
</tr>
</tbody>
</table>

Table 7.3 shows that the numbers of $\chi^{23}_i$ bits the attacker can recover by injecting faults at $\theta^{23}_i$ conform with the analysis above. Because only one round of SHA-3 operations are involved in (7.8), the number of variables is much smaller than the previous two-round process. The time consumption for recovering $\chi^{23}_i$ is much smaller than injecting faults at $\theta^{22}_i$ and recovering $\chi^{22}_i$ bits for four SHA-3 functions. We show the recovery process in details in Figure 7.4.

Table 7.3 and Figure 7.4 show that the attacker needs a small number of faults to recover all the recoverable $\chi^{23}_i$ bits. For example, the attacker needs about only 22 faults to recover all 640 $\chi^{23}_i$ bits of SHA3-512 under 32-bit fault model, and he needs about 45 faults to recover 512 $\chi^{23}_i$ bits for SHA3-384. Thus if the attacker has the ability to inject faults at $\theta^{23}_i$, the cost of recovering the $\chi^{23}_i$ bits is not very high. In next section, we will show how the recovered $\chi^{23}_i$ bits can help SAT solver to recover the $\chi^{22}_i$ bits under more relaxed fault models.
CHAPTER 7. ALGEBRAIC FAULT ANALYSIS OF SHA-3

7.5.2 Attack Results

In this section, we will show that after adding the recovered $\chi_i^{23}$ bits into constraints, the attack efficiency will increase significantly. In this section, we target SHA3-224 and SHA3-256 under 16-bit fault model, and SHA3-384 and SHA3-512 under 32-bit fault model.

First, similar as the attacks proposed in Section 7.4, we combine several faults ($\Delta \theta_i^{22}, H'$) together to recover the first batch of $\chi_i^{22}$ bits. For SHA3-224 under the 16-bit fault model, we need a group of five fault injections ($\Delta \theta_i^{22}, H'$) at the same time for $\chi_i^{22}$ bits recovery. For SHA3-256, we can use a group of four fault injections to recover the first group of $\chi_i^{22}$ bits, the same as in Method II. For SHA3-512 under 32-bit fault model, we need two faults ($\Delta \theta_i^{22}, H'$). Note here that even with 512 $\chi_i^{23}$ bits recovered, the proposed method cannot recover $\chi_i^{22}$ bits for SHA3-384 under 32-bit fault model in reasonable time. We present the distribution of the number of recovered $\chi_i^{23}$ bits for SHA3-256 under 16-bit fault model and SHA3-512 under 32-bit fault model in Figure 7.5.

Comparing with the results shown in Figure 7.2, the probability for SHA3-256 to recover more than 200 $\chi_i^{22}$ bits rises from 36.04% to 100%, and this probability rises from 54.14% to 100% for SHA3-512. For SHA3-224, the probability that the attacker can recover more than 100 $\chi_i^{22}$ bits is 20.63%.

After recovering the first group of $\chi_i^{22}$ bits, the attacker can add them into constraints and recover the remaining bits for the whole $\chi_i^{22}$ internal state. The detailed results are shown in Figure 7.6. For SHA3-224 and SHA3-256, two faults are considered together each time for the remaining bits recovery.

It shows that the improved attack method in this section can effectively break SHA3-224 and
SHA3-256 under the 16-bit fault model, with only 29 faults and 14 faults, respectively. It also breaks SHA3-512 under the 32-bit fault model, with six faults. What’s more, the improved method in this section needs much fewer time than the method in Section 7.4. For example, the method in Section 7.4 needs about 608 seconds to break SHA3-256 while the improved method in this section needs 532 seconds instead. Similarly, the time consumption decreases from 317 seconds to 23 seconds for SHA3-512. Detailed results are shown in Table 7.4.

Note that we make a balance between the number of faults and the time consumption for bits recovery here. For example, if we use a group of two \((\Delta \theta_{i}^{22}, H')\) to get the first group of \(\chi_{i}^{22}\) bits for SHA3-256 using Method III, the time consumption will decrease to 30 seconds, while the total number of faults will increase to more than 20.

Although the improved method proposed in this section still cannot break SHA3-384 and under modes with the 32-bit fault model in reasonable time, this is a complexity issue and more advanced SAT solvers techniques and computation power can help pushing the coverage of cases by the AFA method.

7.6 Discussion

7.6.1 Result Discussion

We summarize the attack results of the proposed three methods on all four SHA-3 functions in Table 7.4.
From Table 7.4, we have the following conclusions:

- The larger $d$, the more vulnerable the SHA-3 function to fault analysis attack. For example, for SHA3-512 under the 32-bit fault model, Method II and Method III both break it, while the other three modes of SHA-3 are not broken yet by any method. The basic method, Method I, breaks SHA3-512 and SHA3-384 under the 16-bit fault model, but not the other two modes with shorter digests.

- For the same attack method under the same fault model, attacker needs less time and fewer faults to break SHA-3 system with larger $d$. This can be seen in the first row for the 8-bit model case.

- For the same SHA-3 function, the attacker needs fewer faults to break it under more relaxed fault models. For example for SHA3-512, the attacker needs 45 faults under single-byte fault model, while only 23 faults under 16-bit fault model, and 6 faults under 32-bit fault model.

This is because for SHA-3 functions with larger $d$, more bits of $H$ and $H'$ are observable by attackers, which means that more bits of digests can be transformed into constraints and this will help SAT solver to find solutions. With tighter constraints, the attacker will need less time to break the whole system and extend the attacks to more relaxed fault models. For more relaxed fault models, the constraints of (7.5) and (7.6) will be looser, and thus the SAT solver will need more time to find the solution.
CHAPTER 7. ALGEBRAIC FAULT ANALYSIS OF SHA-3

Table 7.4: Attack results of proposed attack methods

<table>
<thead>
<tr>
<th>Fault Model</th>
<th>Result</th>
<th>SHA-3 Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>SHA-3-224</td>
</tr>
<tr>
<td>8-bit</td>
<td>Method I</td>
<td>I</td>
</tr>
<tr>
<td></td>
<td>Time</td>
<td>232s</td>
</tr>
<tr>
<td></td>
<td>Faults</td>
<td>69</td>
</tr>
<tr>
<td>16-bit</td>
<td>Method II/III</td>
<td>I</td>
</tr>
<tr>
<td></td>
<td>Time</td>
<td>1347s</td>
</tr>
<tr>
<td></td>
<td>Faults</td>
<td>29</td>
</tr>
<tr>
<td>32-bit</td>
<td>Method II/III</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Time</td>
<td>317s/23s</td>
</tr>
<tr>
<td></td>
<td>Faults</td>
<td>7/6</td>
</tr>
</tbody>
</table>

7.6.2 Countermeasure and Future Work

In this work we present the AFA attack on SHA-3 and propose improved methods to push it towards its limit. We relaxed the fault models to 16-bit and 32-bit fault models. The first line of future work would be pushing AFA onto the remaining three modes, SHA3-224, SHA3-256, and SHA3-384, under the 32-bit fault model. We will also obtain insights on possibly extending to 64-bit fault model, which is the upper limit of data structures for software implementations. We will tap into advanced SAT solvers and more optimization of the attack method.

This work shows that SHA-3 implementation is very vulnerable to fault analysis attacks. With the broad usage of SHA-3 and other Keccak-based crypto applications, the protection of SHA-3 against fault injection attacks is of vital importance. Some work has been done to detect the fault injections (errors) in SHA-3 systems, using either error detection codes [118] or exploiting the properties of Keccak functions [118, 107]. However, it has been shown in a recent work [119] that most of the existing concurrent error detection schemes with high fault coverage are not secure enough against DFA attacks. Thus, efficient and effective methods to protect SHA-3 systems against fault injection attacks should be a part of the future work.

7.7 Summary

In this work, we present several AFA attack methods on SHA-3 functions under relaxed fault models. The methods we present in this work push AFA on SHA-3 towards its limit of fault model
CHAPTER 7. ALGEBRAIC FAULT ANALYSIS OF SHA-3

relaxation. Results show that AFA is more efficient than DFA in fault analysis of SHA-3 in terms of the solving time and number of faults required to recover the entire internal state of $\chi^2_{i}$. Meanwhile, AFA is more effective than DFA on SHA-3, as it break conquer SHA-3 under more relaxed fault models. The proposed methods in this work can break all the four modes of SHA-3 in several minutes under 16-bit fault model, and break SHA3-512 under the 32-bit fault model in several seconds.
Chapter 8

Protect SHA-3 Against Fault Attacks

8.1 Introduction and Motivation

Cryptographic systems are sensitive to random errors caused by aging, ambient environment such as temperature and X-ray radiation \[34\], \[38\]. For cryptographic systems used for encryption, authentication and integrity checking, etc., the random errors will cause incorrect results and make the cryptographic systems unreliable. Attackers can also inject faults temporarily to the system to retrieve the secret key or state. Analyzing the correct output and faulty output is called differential fault analysis (DFA), which has been shown to be very effective in cracking block ciphers including Data Encryption Standard (DES) and Advanced Encryption Standard (AES) \[46\], \[47\]. Another attack method, Fault Sensitivity Analysis (FSA), shows that attackers even do not need the ciphertext for attacks, only the correctness of the computation result is enough for attacks \[120\], \[121\], \[122\], \[123\]. Thus different kinds of fault injection attacks are serious threat to the security of cryptographic systems. Previous chapters also show that SHA-3 is vulnerable to fault injection attacks.

To protect cryptographic systems against random errors and injected faults, different error detection methods have been adopted in cryptographic systems \[33\], \[34\], \[35\], \[36\], \[124\], \[37\], \[38\]. Many schemes are based on redundancy - with another copy of the original implementation for outputs comparison so as to detect error occurring in either copy. To further improve the reliability, the second copy can have different implementation from the first copy \[33\]. To reduce the resource overhead, error detection coding is also widely used. Parity checking code has been used in the protection of AES \[38\] because of its efficient design and high error coverage. Besides parity checking, non-linear codes are also introduced to further improve system reliability with higher fault coverage \[35\].

Another method to detect disturbance used for fault injection has been devised \[39\], \[40\], \[41\], \[42\]. Commonly used system disturbance for fault injection are system clock glitch \[125\], \[126\], power
CHAPTER 8. PROTECT SHA-3 AGAINST FAULT ATTACKS

supply disturbance [127], and electromagnetic (EM) [128], etc.. By detecting the disturbance injected into crypto systems, fault injection attacks can be avoided.

In this chapter, we introduce two schemes to protect SHA-3 systems against fault injection attacks. In Section 8.2 we design efficient error detection method for SHA-3 based on the properties of Keccak function. In Section 8.3 we focus on the detection of glitches in system clock and power disturbance such that to avoid fault injections in SHA-3 system.

8.2 Protect SHA-3 Using Error Detection Codes

8.2.1 Introduction and Motivation

The reliable design of SHA-3 against injected faults and random errors is almost blank. To the best of our knowledge, only one work about reliable Keccak design has been published [65]. In [65], the authors find a property of Keccak algorithm: each lane of the Keccak state can be rotated by a random number before each round operation, and then shifted back after Keccak operations without changing the results. Based on this property, they implement another copy of Keccak with this kind of rotation for comparison. Any error can cause output mismatch of the two copies and thus can be detected. With different implementations in the two copies, it is unlikely that same errors will appear in the two copies. However, this method has high resource overhead due to the extra copy.

In this work, we propose a simple and efficient parity checking based error detection method for Keccak. Optimized designs are also proposed to further improve the efficiency of the proposed scheme. We implement the proposed scheme in VHDL and simulate fault injection at gate level to get the fault coverage of the proposed scheme. Results show that our scheme has a high fault coverage for hardware implementations with very small resource overhead.

The rest of this work is organized as follows. In Section 8.2.2 preliminaries of error detection are introduced. In Section 8.2.3 the mathematical property of Keccak is analyzed and our parity checking scheme for Keccak operations is proposed. In Section 8.2.4 implementation and fault injection simulation results of the proposed schemes are given.

8.2.2 Preliminary of Error Detection

Reliable cryptographic modules rely on error detection to detect random errors and injected faults in systems. The basic structure of error detection is shown in Figure 8.1 in which the cryptographic module is protected by introducing another module, Protector. Protector is composed of three modules, which are Predictor, Compressor and Comparator.
CHAPTER 8. PROTECT SHA-3 AGAINST FAULT ATTACKS

Figure 8.1: Conception of error detection

Predictor and Compressor read the input and output of cryptographic module respectively, and generate some intermediate computation results. Comparator compares the results of Predictor and Compressor for error detection. If errors happen in cryptographic module, results of Predictor and Compressor may not match, and errors will be detected. Design of predictor and compressor will significantly determine the resource overhead and fault coverage. For example, in redundancy-based error detection, predictor is another copy of the original implementation and there is no compressor.

8.2.3 Parity Checking of Keccak

In this section, we will propose our parity checking based concurrent error detection scheme for Keccak. We take each operation of Keccak as a cryptographic module shown in Figure 8.1, and the goal is to design efficient and effective predictor and compressor suitable for each operation.

8.2.3.1 Analysis of the Protection of Keccak

Sponge function involves a lot of simple bitwise operations such as XOR, AND, NOT etc., rather than complex nonlinear operations such as S-box in block ciphers. Such bit-wise operations can be combined according to Boolean algebra for efficient predictor and compressor design. For each 3-D input and output state of a cryptographic operation, parity checking can be implemented at different granularities. For example, it can be implemented in row, column, or lane, or in 2-D entities, in unit of slice, sheet, or plane. This will result in different compression ratio, yielding different error coverage and resource overhead.
CHAPTER 8. PROTECT SHA-3 AGAINST FAULT ATTACKS

For concurrent error detection, usually some steps can be combined to achieve higher efficiency. For example, the error detection of ShiftRows, MixColumns and AddRoundKey of AES can be combined together to achieve lower overhead and higher efficiency. Instead of protecting each operation of Keccak separately, we propose to combine the protections of some operations to improve the efficiency and save resource. For example, \( \iota \) is a binary XOR operation with a constant number, and thus the protection of \( \iota \) can be efficiently combined with the protection of \( \chi \). In this section, we show how to make use of the property of Keccak and combine parity checking of some steps to achieve higher efficiency.

The overall protection scheme is shown in Figure 8.2. The protections of \( \chi \) and \( \iota, \rho \) and \( \pi \), are combined respectively for higher efficiency, with the combined operations denoted as \( \chi' \) and \( \rho' \).

![Figure 8.2: Error detection structure of Keccak](image)

Note that depending on the implementation of Keccak, the \( \rho' \) protector may be optional. This is because both \( \rho \) and \( \pi \) are permutation operations and only change the position of bits without changing the values. If no storage (state register) is used in these operations, they will be synthesized as wires in FPGA or ASIC. Any errors (stuck-at-zero or stuck-at-one) will be detected by the previous \( \theta \) protector or the following \( \chi' \) protector. Nevertheless, we present protection on \( \rho \) and \( \pi \) in this section at the algorithm level, and leave the implementation to the next section.
CHAPTER 8. PROTECT SHA-3 AGAINST FAULT ATTACKS

8.2.3.2 Parity Checking of $\theta$

8.2.3.2.1 Parity Checking of $\theta$ in 1-D Entities

As shown in formula (2.2), each $\theta$ operation works on one input bit $\theta_i$ and two nearby columns (each five bits). Operations on five bits in the same column of $\theta_i$ involve the same two nearby columns. Thus we propose to implement parity checking of $\theta$ along the $y$ axis (in column). For each column of $\theta_o$, the parity checking is as follows:

$$4 \oplus y = 0 \theta_o(x, y, z) = 4 \oplus y = 0 \theta_i(x, y, z) \oplus (4 \oplus 4 \theta_i(x + 1, y, z - 1)) \oplus (4 \oplus 4 \theta_i(x - 1, y, z)).$$

Due to the property of XOR operation ($a \oplus 0 = a$, $a \oplus a \oplus a = a$), we can simplify the above equation as:

$$4 \oplus y = 0 \theta_o(x, y, z) = 4 \oplus y = 0 \theta_i(x, y, z) \oplus (4 \oplus 4 \theta_i(x + 1, y, z - 1)) \oplus 4 \oplus 4 \theta_i(x - 1, y, z)). \quad (8.1)$$

Where the parity checking of each column of $\theta_o$ ($\theta_o(x, Y, z)$ on the left hand) is the parity checking of three $\theta_i$ columns ($\theta_i(x, Y, z)$, $\theta_i(x - 1, Y, z)$ and $\theta_i(x + 1, Y, z - 1)$).

We denote the parity checking result of $\theta_i$ in each column as $P[\theta_i](x, z)$, which means to compress $\theta_i$ along the $y$ direction. The parity checking of the input state is a plane ($X = [0 : 4]$, $Z = [0 : 63]$). Similarly, we denote the parity checking result of $\theta_o$ in each column as $P[\theta_o](x, z)$.

We can represent (8.1) as follows:

$$P[\theta_o](x, z) = P[\theta_i](x, z) \oplus P[\theta_i](x + 1, z - 1) \oplus P[\theta_i](x - 1, z). \quad (8.2)$$

The parity checking can also be done along the row. Similarly, we have:

$$P[\theta_o](y, z) = P[\theta_i](y, z) \oplus P[\theta_i](z) \oplus P[\theta_i](z - 1), \quad (8.3)$$

in which $P[\theta_i](z)$ stands for the compression of each slice of $\theta_i$ and therefore the input state is compressed to a parity lane, and $P[\theta_i](y, z)$ is the parity of a row. Compared to (8.2), it involves multiple parity generations (both in row and in slice).

8.2.3.2.2 ParityChecking of $\theta$ in Slice

Equation (8.2) shows that parity check in each column involves nearby columns (with one on the same slice). Equation (8.3) shows that parity checking in each row already involves two slices. We examine parity checking in 2-D entities too. Parity of each slice is denoted as:

$$4 \oplus 4 \oplus \theta_o(x, y, z) = (4 \oplus 4 \theta_i(x, y, z)) \oplus (4 \oplus 4 \theta_i(x - 1, y, z))$$

$$\oplus (4 \oplus 4 \theta_i(x + 1, y, z - 1)). \quad (8.4)$$
CHAPTER 8. PROTECT SHA-3 AGAINST FAULT ATTACKS

Due to the property of XOR operation, we find a unique property for slice based parity checking of $\theta$ operation:

$$\theta_o(x, y, z) = \frac{4}{x=0} \oplus \frac{4}{y=0} \theta_i(x + 1, y, z - 1).$$

(8.5)

It means that the parity of each slice of $\theta_o$ is the parity of a nearby slice of $\theta_i$. For the 1, 600-bit state of Keccak, there are 64 slice-based parity checking bits for $\theta_i$ and $\theta_o$ respectively. Define $P[\theta_o](Z)$ and $P[\theta_i](Z)$ as following:

$$P[\theta_i](z) = \frac{4}{x=0} \oplus \frac{4}{y=0} \theta_i(x, y, z)$$

$$P[\theta_o](z) = \frac{4}{x=0} \oplus \frac{4}{y=0} \theta_o(x, y, z)$$

then (8.5) can be represented as:

$$P[\theta_i](z) = P[\theta_o](z + 1), \ Z = [0 : 63],$$

(8.6)

which means that the slice-based parity lane of $\theta_o$ is a round shift of the parity lane of $\theta_i$. This makes the slice-based parity checking for $\theta$ very efficient.

For the three parity generation and checking schemes introduced in Section 8.2.3.2.1 and Section 8.2.3.2.2, we compare their resource overhead in terms of XOR gates. Take parity checking in slice as an example, the predictor compresses every slice into one bit, requiring 24 XOR gates, and the predictor needs 1536 (24 * 64) XOR gates in total. The XOR gates consumption of different protection schemes of $\theta$ are listed in Table 8.1.

Table 8.1: XOR gates overhead of $\theta$ protection

<table>
<thead>
<tr>
<th></th>
<th>Predictor</th>
<th>Compressor</th>
<th>Comparator</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Column</td>
<td>1920</td>
<td>1280</td>
<td>639</td>
<td>3839</td>
</tr>
<tr>
<td>Row</td>
<td>2176</td>
<td>1280</td>
<td>639</td>
<td>4095</td>
</tr>
<tr>
<td>Slice</td>
<td>1536</td>
<td>1536</td>
<td>127</td>
<td>3199</td>
</tr>
<tr>
<td>Duplicate</td>
<td>3520</td>
<td>0</td>
<td>3199</td>
<td>6719</td>
</tr>
</tbody>
</table>

For schemes with parity generation in row and column, they protect $\theta$ for every five bits, while the slice based scheme protect $\theta$ operation for every 25 bits. Thus the row and column based schemes have higher error coverage than the slice based scheme. Table 8.1 shows that the slice based parity checking scheme has lower area overhead than the other two schemes. A balance should be made between error coverage and resource overhead during design. Meanwhile, comparing with the proposed parity checking schemes, duplication based error detection requires much higher resource overhead. In this work, we use slice-based parity generation to implement the parity checking of $\theta$ operation.
8.2.3.3 Parity Checking of $\rho$ and $\pi$

As discussed in Section 8.2.3.1, $\rho$ and $\pi$ can be left unprotected if they are implemented using wires in circuit. For implementations in which $\rho$ and $\pi$ use registers (pipelined design, for example), the protections of $\rho$ and $\pi$ can be implemented either separately or combined together.

For $\rho$, the permutation is along z-axis, thus we propose to compress $\rho$ operation along each lane, and the protection function is as following:

$$P[\rho_i](x, y) = P[\rho_o](x, y). \quad (8.7)$$

In (8.7), the left side is the predictor while the right side is the compressor design. Similarly, while $\pi$ permutes the state bits inside each slice, we propose to compress the bits inside each slice:

$$P[\pi_i](z) = P[\pi_o](z). \quad (8.8)$$

The protection of $\rho$ and $\pi$ can both be implemented efficiently because of their simple operations. To further improve the implementation efficiency, we propose to combine $\rho$ and $\pi$ as a new operation $\rho'$ and protect it instead. For the protection of $\rho'$, parity checking can be efficiently implemented in z-axis. The protector of $\rho'$ can be designed as following:

$$\pi(P[\rho'_i](x, y)) = P[\rho'_o](x, y), \quad (8.9)$$

in which the left-hand side is predictor and the right-hand side is for compressor design. For predictor design, the 1,600-bit state is firstly compressed to a slice $P[\rho'_i](x, y)$, and this slice is then permuted according to $\pi$ operation. For this protection scheme, the predictor and compressor both require 1,575 XOR gates, and the comparator needs 49 XOR gates. Although $\theta$ can be also combined with $\rho'$, the efficiency is not improved significantly, thus we protect them separately in this work.

8.2.3.4 Parity Checking of $\chi$ and $\iota$

8.2.3.4.1 What Should We Avoid in Protection of $\chi$?

$\chi$ is the only non-linear step in Keccak and it involves both NOT and AND operations. This makes the parity checking of $\chi$ step different from previous operations. In this section, we first show a pitfall for the protection of $\chi$ which should be avoided in practical design.

We take one row of $\chi_i$ as an example here, we denote the five bits of this row as $\{a, b, c, d, e\}$. Then five bits of corresponding $\chi_o$ output row can be denoted as $a \oplus (\bar{b} \cdot c), b \oplus (\bar{c} \cdot d), c \oplus (\bar{d} \cdot e), d \oplus (\bar{e} \cdot a)$ and $e \oplus (\bar{a} \cdot b)$. The parity bit of this single row is:

$$a \oplus b \oplus c \oplus d \oplus e \oplus (\bar{b} \cdot c) \oplus (\bar{c} \cdot d) \oplus (\bar{d} \cdot e) \oplus (\bar{e} \cdot a) \oplus (\bar{a} \cdot b)$$

$$= (a \cdot b) \oplus (b \cdot c) \oplus (c \cdot d) \oplus (d \cdot e) \oplus (e \cdot a). \quad (8.10)$$
CHAPTER 8. PROTECT SHA-3 AGAINST FAULT ATTACKS

For equation (8.10), if one bit out of five bits in this row is flipped, the final result of equation (8.10) may not change. Take bit \( a \) as an example, the change of \( a \) will affect both \((e \cdot a)\) and \((a \cdot b)\). According to De Morgan’s laws:

\[
(e \cdot a) \oplus (a \cdot b) = a \cdot (b \oplus e).
\] (8.11)

When \( a \) flips, if \( b \oplus e \) is already 0, the above result does not change, then the result of (8.10) will not change either. Assume all bits are independent, the probability of \( b \oplus e = 0 \) is 50\% and therefore the fault coverage of this scheme is 50\% for single bit errors. So parity checking in each row of \( \chi \) should be avoided because of the non-linearity of \( \chi \) operation, and thus parity checking in each slice will not be applicable for \( \chi \) operation either.

8.2.3.4.2 Parity Checking of \( \chi \) in Each Lane

In this section, we show how to build practical error detection module for \( \chi \) operation. As x-axis compression of \( \chi \) is not a good choice, we considering compressing \( \chi \) results along either z-axis or y-axis. For parity generation along z-axis, 64 bits in each lane are compressed to one bit and thus the 1,600 bits are compressed to one slice. The parity generation can be denoted as following:

\[
P[\chi_o](x, y) = \oplus_{z=0}^{63} \chi_o(x, y, z).
\] (8.12)

According to the definition of \( \chi \) operation, we have:

\[
P[\chi_o](x, y) = \oplus_{z=0}^{63} (\chi_i(x, y, z) \oplus \chi_i(x + 2, y, z) \oplus \chi_i(x + 1, y, z) \cdot \chi_i(x + 2, y, z))
\]

\[
= P[\chi_i](x, y) \oplus P[\chi_i](x + 2, y) \oplus P[\chi_{and}](x, y),
\]

in which \( \chi_{and}(x, y, z) = \chi_i(x + 1, y, z) \cdot \chi_i(x + 2, y, z) \). Thus the predictor design of parity checking for \( \chi \) at z-axis is also easy to implement. It involves AND operations first, then it compresses the data in z-axis to generate the parity for checking.

Meanwhile, the parity of \( \chi \) can also be generated in each column, along y-axis direction. For this scheme, five bits in each column are compressed to one bit, and the 1,600-bits state is compressed to one plane. The compressor works as follows:

\[
P[\chi_o](x, z) = P[\chi_i](x, z) \oplus P[\chi_i](x, z) \oplus P[\chi_{and}](x, z).
\]

In this scheme, the overhead is higher than the z-axis compression scheme due to its lower compression ratio, while it has higher fault coverage. Thus designers can choose the best scheme according to the system requirement. In this work, we implement the protection of \( \chi \) operation along the z-axis direction (lane) for fault injection simulation and resource overhead evaluations.
CHAPTER 8. PROTECT SHA-3 AGAINST FAULT ATTACKS

8.2.3.4.3 Combination of $\chi$ and $\iota$

While $\iota$ only adds a constant number to $\chi$ result, it can be easily combined with $\chi$ as discussed in previous section. If $\chi$ is checked at z-axis, the combined parity checking is

$$P[\iota_c](x, y) = P[\chi](x, y) \oplus P[\chi](x + 2, y) \oplus P[\chi](x, y) \oplus P[\iota_c](x, y), \quad (8.13)$$

and parity checking of $\chi$ combining with $\iota$ at other directions are similar. Note here that $P[\iota_c](x, y)$ is computed at design stage to avoid computations for each run. This protection scheme requires 3,200 XOR gates and 1,600 AND gates for the predictor, 1,575 XOR gates for the compressor, and 49 XOR gates for the comparator.

8.2.4 Implementation and Fault Injection Results

8.2.4.1 Implementation Results

To evaluate the proposed scheme, we implement the unprotected Keccak implementation (referring to the official implementation provided online [56]) and the proposed scheme in Figure 8.2. We implement three variants of the proposed scheme:

- **Proposed** combines the protection of $\chi$ and $\iota$ as described in Section 8.2.3.4.3 and leaves $\rho$ and $\pi$ unprotected;
- **Design 2** protects $\chi$ and $\iota$ separately, and leaves $\rho$ and $\pi$ unprotected;
- **Design 3** combines the protection of $\chi$ and $\iota$ as described in Section 8.2.3.4.3 and protects $\rho$ and $\pi$ together referring to (8.9).

For the above three designs, we implement the protection of $\theta$ using the scheme in Section 8.2.3.2.2, which is to implement parity checking in each slice. Each protector is composed of three parts, as described in Figure 8.1. All the designs have five steps in each round within one clock cycle using combinational circuits. For the proposed schemes, we implement the original circuit and the protection circuits in parallel and they work simultaneously.

For integrated circuit resource evaluation, the implementations (with and without protections) are modeled in VHDL and synthesized in Cadence Encounter RTL Compiler with a 45nm Opencell library (NanGate FreePDK45 v1.3 v2009.07). The designs were placed and routed using Cadence Encounter. The power and area overhead of the protection schemes were estimated using Concurrent Current Source (CCS) model under typical operation conditions assuming a supply voltage of 1.1V and a temperature of 25 Celsius degree. The results including area, timing delay and power consumption are shown in Table 8.2.
CHAPTER 8. PROTECT SHA-3 AGAINST FAULT ATTACKS

Table 8.2: Resource overhead and error coverage results

<table>
<thead>
<tr>
<th></th>
<th>Area (um²)</th>
<th>Timing (ns)</th>
<th>Power (mW)</th>
<th>Error coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Normalized</td>
<td>Normalized</td>
<td>Normalized</td>
<td></td>
</tr>
<tr>
<td>Original</td>
<td>41611.7</td>
<td>3.892</td>
<td>17.95</td>
<td>0%</td>
</tr>
<tr>
<td>Proposed</td>
<td>52867.2</td>
<td>4.500</td>
<td>26.69</td>
<td>83.60%</td>
</tr>
<tr>
<td>Design 2</td>
<td>62429.4</td>
<td>5.476</td>
<td>41.78</td>
<td>83.34%</td>
</tr>
<tr>
<td>Design 3</td>
<td>66621.0</td>
<td>4.381</td>
<td>44.07</td>
<td>89.89%</td>
</tr>
</tbody>
</table>

Table 8.2 shows that Proposed has much higher performance while lower area and power consumption than Design 2 and Design 3.

Comparing with original implementation, Proposed has about 27.05% area resource overhead. Meanwhile, our proposed scheme maintains high performance because it has the protection modules works concurrently with Keccak module. What’s more, the proposed scheme combines $\rho$ and $\pi$, $\chi$ and $\iota$ together respectively, thus it has small timing delay overhead.

Comparing with Proposed, Design 2 does not optimize the protections of $\chi$ and $\iota$ and protect these two modules separately. It shows that Design 2 has much higher area resource overhead than the proposed design because of this separate protection design. Meanwhile, it has larger timing delay and power consumption than the proposed design as well. Design 3 protects $\rho$ and $\pi$ together as described in Section 8.2.3.4.3, results show that it has much higher area resource overhead than the proposed scheme, as well as larger timing delay and power consumption. Thus the proposed optimization methods can help to save resources in the protection of Keccak.

In conclusion, the proposed scheme has acceptable area resource overhead and high performance at the same time. The proposed protection optimization methods can help to save area resource and improve performance. Comparing with some protection techniques generic for encryption algorithms, the overhead of our proposed scheme is very small. Some schemes introduce another copy of the original design or rotated design for error detection [37, 65], such scheme will have much higher resource overhead than our proposed design.

8.2.4.2 Fault Coverage Analysis

Theoretically, for parity checking schemes, if odd number bits are flipped, the errors will be detected with 100% probability; while if even number bits are flipped, the errors will be always undetected. For real hardware systems, it will be very difficult for the attackers to precisely control the numbers and positions of faulty gates in the circuit [129]. What’s more, the errors in the circuit...
CHAPTER 8. PROTECT SHA-3 AGAINST FAULT ATTACKS

will randomly propagate and cause different numbers of faulty bits in the output [130]. Thus, fault injection simulation results at gate level are required for error coverage evaluation.

In this work, we randomly inject one to ten stuck-at-0 and stuck-at-1 faults into Keccak circuit for error coverage simulation. To get the fault coverage result, we give random plaintext input for one round of Keccak, then randomly inject one to ten stuck-at-0 or stuck-at-1 faults into the system. We check the results of the protected implementation and the alarm signals to see if we miss any errors. For each design, we run about $10^8$ fault injection trials and the error coverage results are shown in Table 8.2.

Table 8.2 shows that the proposed scheme has error coverage about 83.60%. Meanwhile, results show that separate protections of $\chi$ and $\iota$ in Design 2 will not increase the error coverage. This is because $\iota$ module is very small and only occupies very small ratio of gates in the design. The probability of errors happening in $\iota$ is very small, and the error coverage of Design 2 is almost the same as the proposed design.

For Design 3, it has a large part of gates used for the protection of $\rho$ and $\pi$, and errors happen in $\rho'$ module with high probability. In such case, we can assume that a large part of errors are injected into the protection module of $\rho'$ and this part of errors will be detected with high probability. Results show that the error coverage will increase to 89.89% for Design 3. Thus for pipelined designs which use registers to store the results of $\rho$ and $\pi$, the protection method proposed in Section 8.2.3.3 should be implemented for higher error coverage.

In conclusion, according to the synthesis results in Section 8.2.4.1 and fault injection simulation results in Section 8.2.4.2 the proposed scheme has a small resource overhead and high performance, and it can detect the injected faults with a high probability. Thus our proposed scheme strikes a good balance between resource overhead and error coverage, and can be efficiently implemented for the protection of Keccak implementations.

8.3 Protect SHA-3 by Detecting System Disturbance

8.3.1 Introduction

In this work, we devise a method to detect clock glitches using ring oscillator (RO) in FPGA, and RO and voltage-controlled oscillator (VCO) in ASICs. We implement Advanced Encryption Standard (AES) protected with the proposed scheme in a Virtex-5 FPGA, we inject glitches into the system clock and measure the detection rate of the proposed scheme. Results show that the proposed scheme can detect very high frequency clock glitches with a high detection rate. Meanwhile, it can be easily implemented in both FPGAs and ASICs with very small resource overhead.
CHAPTER 8. PROTECT SHA-3 AGAINST FAULT ATTACKS

The rest of this work is organized as follows. Preliminaries of fault injection using clock glitches are given in Section 8.3.2, followed by introduction of previous works. In Section 8.3.3, we introduce the proposed method used to detect clock glitch, and analysis about the proposed scheme will be given. In Section 8.3.4, we present the clock glitch detection results of the proposed scheme, and the implementation overhead results.

8.3.2 Fault Injection Using Clock Glitch

Clock glitches are widely used to inject faults into crypto hardware systems. In crypto systems, each round of the crypto algorithm needs some amount of time to generate the stable results, and we denote this time as $t_{\text{setup}}$. If the clock cycle is smaller than $t_{\text{setup}}$, the computation result will be passed to next round before it’s stable, thus errors will happen in next round input. Shown in Figure 8.3, attackers can inject glitch with cycle $\Delta t$ ($\Delta t < t_{\text{setup}}$) into system clock to generate incorrect computation results.

![Figure 8.3: Fault injection using clock glitch](image)

Comparing with other fault injection methods, fault injection based on clock glitch is easy to implement and it’s easy to control the width of glitch $\Delta t$. Thus besides DFA attacks [126, 131], clock glitch based fault injection has also been widely used in FSA attacks which require precise control of the strength (width) of the disturbance (clock glitch) [120, 122, 123].

8.3.2.1 Related Works

In this work, we focus on the detection of clock glitches. In [40, 41], the authors propose to insert a delay buffer chain into an ASIC. The delay chain consists of a series of buffers followed by an inverter, and is inserted between two registers, such that the source register receives the complement of its current value every clock cycle. The latency of this delay buffer chain should be a little greater than the delay of the crypto module, such that if the clock cycle is smaller than the critical path delay, the complement relationship will be violated and thus the clock glitch will be detected.
CHAPTER 8. PROTECT SHA-3 AGAINST FAULT ATTACKS

In [39], a time-to-delay converter is implemented in a 45 nm microprocessor to monitor the system variations. Its fast time-to-digital converter enables quick characterization of transients and therefore the method can detect even nanosecond-scale glitches. Previous papers also find that the frequency of ring oscillators change with the supply power voltage, and this has been used to find the inserted resistors used for power acquisition [132]. However, this property has not been discussed for clock glitch detection.

Not much work has done on the detection of clock glitch in crypto systems. Meanwhile, as FPGAs have been widely used in crypto applications, protection methods which require analog modules are not suitable for FPGA devices. Good protection scheme should make use of existing resource in FPGAs, and should be easy to implement in both FPGAs and ASICs. In this work, we design efficient clock glitch detection module which can be easily implemented in both FPGAs and ASICs.

8.3.3 Clock Glitch and Power Disturbance Detection

In this work, we propose to monitor the system clock signal clock using a higher frequency clock signal clk. We propose to generate this high frequency signal clk inside cryptographic systems using available resources in FPGA and ASIC. In FPGA, we generate clk using RO which requires only inverters and buffers, and we propose to generate clk using either RO or VCO in ASIC. We will present the glitch detection method assuming high frequency signal clk is available in this section, detailed implementation methods will be given in next section.

8.3.3.1 System Clock Glitch Checking

We assume that the generated clock signal clk has higher frequency than clock, and we propose to measure the width of clock using clk for cross-checking.

Firstly, we consider system clock signal with unequal high and low pulse, which means that the duty cycle is not 50%. Correspondingly, we have two separate width counters for the high pulse and low pulse, respectively. For two continuous cycles, the counter results should be the same. The width results of first cycle \( (n^L_0, n^H_0) \) and result in the second cycle \( (n^L_1, n^H_1) \) are shown in Figure 8.4.

If clock and clk are both glitch free, the width of the previous pulse should be equal with the current pulse, which means:

\[
\begin{align*}
  n^L_0 &= n^L_1, \\
  n^H_0 &= n^H_1.
\end{align*}
\]  

(8.14)

If the clock is not constant, the width of logic ‘0’ or logic ‘1’ of clock is varying. Such difference between two consecutive cycles can be used to detect glitches in the system clock. If a glitch happens

129
in system clock clock, the width of clock in terms of number of clk will change suddenly, and then (8.14) will not hold. We assume that there is a glitch in system clock as Figure 8.5.

Then the width results will be different and the glitch will be detected by the comparators and the Alarm signal is triggered:

\[
\begin{align*}
& n_0^L \neq n_1^L, n_1^L \neq n_2^L, \\
& n_0^H \neq n_1^H, n_1^H \neq n_2^H,
\end{align*}
\]  

(8.15)

By comparing the measured width values continuously, glitches in clock signal should be detected. We note here that for clock with 50% duty cycle, \(n_0^L, n_0^H, n_1^L\) and \(n_1^H\) should be all equal with each other, and they can be compared with each other for system monitoring. Based on the previous discussions, we propose a detection module with the structure shown in Figure 8.6.

The proposed structure has two width counters, one is used to measure the width of the low pulse while the other one is used to measure the width of the high pulse. The width counter result of current cycle is compared with previous cycle for glitch detection. Alarm will be triggered if glitch in clock is detected, and this will stop the output of faulty cryptographic results.

In the above discussions, we assume that the frequency of generated clock signal clk is higher than the frequency of clock, and use clk to measure the width of clock. Actually, in some crypto systems, clock can have very high frequency, and in such situation, we can use ROs and DCMs.
 CHAPTER 8. PROTECT SHA-3 AGAINST FAULT ATTACKS

8.3.3.2 Relaxation of the Design for Robustness

In previous section, we assume that both clock and clock are stable if no glitch exists in system clock. Actually, electronic parts are sensitive to environment variations like temperature. The output frequency of RO, VCO also changes with environmental variations, because the buffer delay is affected by temperature significantly. This is also a problem for the previously proposed scheme [41]. Meanwhile, the power supply may have small inherent variations due to noise or temperature variations, and system clock may have small jitters. Such variations should be differentiated from intentionally inserted glitches. In this section, we present methods to improve robustness of the proposed system to cancel the effect of such variations.

As environmental factors like temperature change slowly, the output frequency variations of VCO and RO caused by the environmental factors are very slow. While glitches in clock always change the frequency of clock abruptly, we can set a comparison threshold to improve the robustness of the proposed scheme. Previous works show that although ring oscillator output frequency changes with the temperature, the change is only several megahertz [133] when temperature changes 100 °C, which is very small comparing with the changes caused by clock glitches. The frequency change of clock caused by environmental variations is much smaller than the change caused by glitches.

Assume that clock or clock have slight variations and thus the width of clock is not constant, we set the comparator to tolerant difference between \( n_0^L \) and \( n_1^L \) (also the difference between \( n_0^H \) and

Figure 8.6: Clock width monitoring system

to generate clk with much lower frequency and use clock to measure the width of clk for system monitoring.
CHAPTER 8. PROTECT SHA-3 AGAINST FAULT ATTACKS

\( n^H_i \) below \( \lambda \), i.e., variation that is not caused by glitches in clock or power supply is acceptable:

\[
\begin{align*}
| n^L_0 - n^L_1 | & < \lambda \\
| n^H_0 - n^H_1 | & < \lambda
\end{align*}
\tag{8.16}
\]

In our design, the width of current clock cycle is compared with previous clock cycle instead of a constant value, slow changes in the system caused by environmental variations will not trigger the alarm. Thus environmental variations have little effect on the proposed scheme.

8.3.3.3 Upper bound of the Proposed Scheme

For clock glitch detection module, the most important thing is the detection range of glitches. Good detection scheme should have high detection rate for even high frequency injected glitches. We denote the frequency of \( \text{clk} \) as \( f_{\text{clk}} \), and the frequency of glitch as \( f_g \), then the proposed scheme can detect all injected glitches with \( f_g \) no higher than \( \frac{1}{2} f_{\text{clk}} \).

In Figure 8.7(a), we show that if \( f_g > \frac{1}{2} f_{\text{clk}} \), the high pulse (and low pulse) of clock glitch may be between two consecutive rising edges of \( \text{clk} \), and then the glitch will not be detected in this situation. Meanwhile, in Figure 8.7(b), \( f_g < \frac{1}{2} f_{\text{clk}} \), which means the cycle of clock glitch should be greater than two cycles of \( \text{clk} \). Then the high pulse (low pulse) of the glitch will always be detected by at least one cycle of \( \text{clk} \).

![Figure 8.7: Upper bound of the detection](image)

8.3.4 Glitch Detection and Implementation Results

We implement unprotected AES and AES protected with the proposed scheme on a Virtex-5 LX30 FPGA, on a SASEBO-GII board. We use a Rigol DG4162 arbitrary waveform generator as
system clock source to generate high frequency glitch signal. In this section, we present the detection results of injected clock glitches, and also the resource overhead of the proposed scheme.

8.3.4.1 Internal Clock Signal Generation

Different methods can be used to generate clocks in FPGA and ASIC implementations. For FPGA implementations, RO can be used to generate clock signals with different frequency. RO has been widely used for physical uncolonable function (PUF) and random number generator design in FPGA. In ASIC, VCO can be used to generate stable clock signals besides RO.

First of all, we measure the critical path delay of the AES module in Virtex-5 implementation. Results show that the critical path delay is about 129 MHz, which means that errors start to happen when the frequency of glitch is higher than 129 MHz. Meanwhile, all ciphertext bytes will be faulty if the clock cycle is higher than about 144.5 MHz when we inject clock glitch in the last round. Thus clock glitch used by attacker should be between this range (129 MHz to 144.5 MHz). According to the conclusion in Section 8.3.3.3, \( \text{clk} \) with frequency higher than 289 MHz will be able to detect all the injected glitches.

We implement five ROs in a Virtex-5 FPGA, they have three, five, seven, nine and eleven stages respectively. With default synthesis and mapping constraints, their frequency are 400 MHz, 305 MHz, 276 MHz, 167 MHz and 108.5 MHz respectively. So the first two ROs can be used for system clock glitch detection and they should be able to detect all the injected glitches for DFA and FSA attacks.

Meanwhile, Digital Clock Manager (DCM) and phase-locked loop (PLL) modules in FPGA can be used to generate stable signals with different frequency using existing clock signals. Thus RO and DCM (or PLL) can be used together to generate clock signals with different frequency. For example, DCM module can generate up to 450 MHz clock in Virtex-5 FPGAs. Thus ROs which have frequency not high enough can be used together with DCM and PLL in FPGA to generate high frequency clock signal to improve the glitch detection coverage.

8.3.4.2 Clock Glitch Detection Result

In the experiment, we use Rigol DG4162 to generate 18 MHz to 50 MHz clock signal, and use this signal to generate 0.5 time frequency \( \text{clock} \) for AES computation (9 MHz to 25 MHz). Meanwhile, we use a DCM module in the Spartan-3A FPGA on SASEBO-GII board to generate 5 times frequency glitch signal (90 MHz to 250 MHz). For each clock frequency, we run glitch injection at the 9\(^{th}\) round 10,000 times, and summarize the clock glitch detection rate as in Figure 8.8.
CHAPTER 8. PROTECT SHA-3 AGAINST FAULT ATTACKS

Figure 8.8 shows the clock glitch detection rate for two different ROs, RO₁ (414.1 MHz) and RO₂ (368.5 MHz). It shows that:

- RO₁ can detect up to about 207 MHz glitch with almost 100% rate, while RO₂ can detect up to about 185 MHz glitch frequency with 100% rate. This is the same as the conclusion in Section 8.3.3.3.

- Higher frequency clk has overall higher detection rate for injected clock glitches. For example, RO₁ has higher error detection than RO₂ based scheme.

To improve the clock glitch detection rate, we propose to combine the use of more than one detection modules, each with independent clk signal. Thus the system will miss the clock glitch only if all detection modules miss the clock glitch. Assume for one detection module, the detection rate is p, then for n detection module, the detection rate should be 1 − (1 − p)^n instead, which is much higher than p. Which means that the combination of multiple detection module will significantly improve the detection rate of the proposed scheme. We combine two detection modules for glitch detection and present the results in Figure 8.9.

Figure 8.9 shows the detection rate for two designs each with two detection modules, working with two different ROs respectively. Results show that the detection rates for these two designs improve significantly comparing with the single detection module design in Figure 8.8. For example, the design with 373.3 MHz plus 276.2 MHz ROs have much higher detection rate than the design with only 368.5 MHz RO. Thus more detection module can be combined together to further improve the detection rate. We will show that although this scheme can significantly improve the detection rate of clock glitches, the resource overhead will be still very small in Section 8.3.4.3.
8.3.4.3 Overhead Result of the Proposed Scheme

We summarize the resource overhead results of the proposed scheme in Table 8.3. Table 8.3 has four rows, which are for unprotected AES, AES protected with one detection module, two detection modules and three detection modules respectively. For FPGA resource evaluation, we use Xilinx ISE 14.6 with default settings for all the implementations. For integrated circuit resource evaluation, the implementations (with and without protections) are modeled in VHDL and synthesized in Synopsys Design Compiler using a 45-nm standard-cell library (FreePDK45). The power and area overhead of the protection schemes are estimated under typical operation conditions assuming a supply voltage of 1.2V and a temperature of 25 Celsius degree. The resource utilization results for both FPGA and ASIC implementations are shown in Table 8.3.

<table>
<thead>
<tr>
<th></th>
<th>FPGA</th>
<th>ASIC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Slice Regs</td>
<td>Slice LUTs</td>
</tr>
<tr>
<td>Unprotected</td>
<td>748</td>
<td>2,206</td>
</tr>
<tr>
<td>One RO</td>
<td>807</td>
<td>2,362</td>
</tr>
<tr>
<td>Two ROs</td>
<td>867</td>
<td>2,458</td>
</tr>
<tr>
<td>Three ROs</td>
<td>910</td>
<td>2,560</td>
</tr>
</tbody>
</table>

Table 8.3 shows that the proposed scheme has very small resource overhead. For example, for AES protected with one detection module, the design needs 7.9\% more slice registers, 7.1\% more...
slice lookup tables (LUTs), and 4.6% more LUT flip-flops (FFs) than the original design. For design with multiple detection modules, the overhead increase slightly. For example, the design with two detection modules needs 15.9%, 11.4% and 10.7% more slice registers, slice LUTs and LUT FFs overhead respectively. For ASIC implementations, the protected implementation with one protection module consumes 4.63% more area resource than the original design, and designs with two and three protection modules need 7.62% and 11.56% more area resource overhead respectively.

Results in Section 8.3.4.2 and Section 8.3.4.3 show that the proposed scheme has a high glitch detection rate for high frequency clock signals. Meanwhile, the proposed scheme can be easily implemented in both FPGA and ASIC with small resource overhead. The proposed scheme has high robustness against environmental variations, and more than one modules can be used together to further improve the clock glitch detection rate.

8.4 Summary

In this chapter, we propose two methods to protect SHA-3 implementations against random errors and fault injection attacks. The first method adopts the conception of error detection codes, and the second method detects system disturbance such as clock glitches and power disturbance such as to avoid fault injections. Both methods can highly improve the security level of SHA-3 implementations with limited resource overhead.
Chapter 9

Conclusion

In this dissertation, I investigate both side-channel security analysis and protection of SHA-3 implementation. Specifically, I consider side-channel power/EM attacks and fault analysis of SHA-3.

In Chapter 3, I present our side-channel power attacks methods on hardware implementation of SHA-3, and analyze the source of leakages in hardware implementations. Our work shows that the hardware implementations of SHA-3 are very vulnerable to side-channel power attacks.

In Chapter 4, I present the methods to mitigate side-channel power leakages of SHA-3 by using the properties of Keccak operations. Results show that the proposed methods can effectively mitigate the leakages of SHA-3 implementations with limited resource overhead. I adopt the open-source LLVM compilation platform for our implementation.

In Chapter 5, I implement TI in a compiler to automatically generate protected code against side-channel power analysis. As the compiler-assisted technique is algorithmic-agnostic, it can be applied to any crypto algorithm, including SHA-3 and AES. The generated designs are implemented on a commercial ARM Cortex-M3 processor, and the real attacks results show that the generated implementations are secure against side-channel power attacks.

In Chapter 6 and Chapter 7, I use DFA and AFA to break SHA-3 using injected faults respectively. For DFA, I present the details of fault propagation in SHA-3, and the method to identify the injected faults and then recover the internal state bits. For AFA, I extend fault analysis of SHA-3 by introducing algebraic techniques into analysis, to avoid the tedious fault propagation analysis and leverage the power of SAT/SMT solvers to recover the internal state bits. Results show that SHA-3 is very vulnerable to fault analysis, and attacker can recover the MAC key when SHA-3 is used as a MAC function.

To protect SHA-3 against fault analysis attacks presented in Chapter 6 and Chapter 7, I propose different countermeasures in Chapter 8. First, I use error detection codes to detect injected faults so
as to avoid faulty digest output. Second, I design a sensor for hardware implementations to detect
clock glitch and power disturbance which are widely used to inject errors in crypto systems. Results
show that the proposed methods can detect injected faults and disturbance with high rate, while the
resource overhead is very small.

In this dissertation, I analyze and proposed different side-channel attack methods for SHA-3,
and present corresponding schemes to protect SHA-3 implementations from such attacks. Our
analysis and results show that SHA-3 is very vulnerable to various side-channel attacks. Meanwhile,
we protect SHA-3 against these attacks by introducing various countermeasures, and our methods
can be used to improve the security level of SHA-3 systems significantly. We believe that our works
are important for the security analysis and protection of SHA-3.
Bibliography


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


[124] P. Luo, Y. Fei, L. Zhang, and A. Ding, “Side-channel power analysis of different protection schemes against fault attacks on AES,” in ReConFigurable Computing and FPGAs (ReConFig), 2014 International Conference on, 2014, pp. 1–6.


BIBLIOGRAPHY


My Publications

Conference Publications


14. **P. Luo**, Y. Fei, L. Zhang, and A. A. Ding, “Side-channel power analysis of different protection schemes against fault attacks on AES,” in *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*, Dec 2014, pp. 1–6.


**Journal Publications**


**Online Documents**


**Publication at Boston University**


BIBLIOGRAPHY

Publication at Chinese Academy of Sciences

