

# **SECURITY OF BLOCK CIPHERS**

**From Algorithm Design to  
Hardware Implementation**

KAZUO SAKIYAMA • YU SASAKI • YANG LI

**WILEY**



# **SECURITY OF BLOCK CIPHERS**



# **SECURITY OF BLOCK CIPHERS**

## **FROM ALGORITHM DESIGN TO HARDWARE IMPLEMENTATION**

**Kazuo Sakiyama**

*The University of Electro-Communications, Japan*

**Yu Sasaki**

*NTT Secure Platform Laboratories, Japan*

**Yang Li**

*Nanjing University of Aeronautics and Astronautics, China*

**WILEY**

This edition first published 2015  
© 2015 John Wiley & Sons Singapore Pte. Ltd.

Registered office  
John Wiley & Sons Singapore Pte. Ltd., 1 Fusionopolis Walk, #07-01 Solaris South Tower, Singapore 138628.

For details of our global editorial offices, for customer services and for information about how to apply for permission to reuse the copyright material in this book please see our website at [www.wiley.com](http://www.wiley.com).

All Rights Reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, scanning, or otherwise, except as expressly permitted by law, without either the prior written permission of the Publisher, or authorization through payment of the appropriate photocopy fee to the Copyright Clearance Center. Requests for permission should be addressed to the Publisher, John Wiley & Sons Singapore Pte. Ltd., 1 Fusionopolis Walk, #07-01 Solaris South Tower, Singapore 138628, tel: 65-66438000, fax: 65-66438008, email: [enquiry@wiley.com](mailto:enquiry@wiley.com).

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

Designations used by companies to distinguish their products are often claimed as trademarks. All brand names and product names used in this book are trade names, service marks, trademarks or registered trademarks of their respective owners. The Publisher is not associated with any product or vendor mentioned in this book. This publication is designed to provide accurate and authoritative information in regard to the subject matter covered. It is sold on the understanding that the Publisher is not engaged in rendering professional services. If professional advice or other expert assistance is required, the services of a competent professional should be sought.

**Limit of Liability/Disclaimer of Warranty:** While the publisher and author have used their best efforts in preparing this book, they make no representations or warranties with respect to the accuracy or completeness of the contents of this book and specifically disclaim any implied warranties of merchantability or fitness for a particular purpose. It is sold on the understanding that the publisher is not engaged in rendering professional services and neither the publisher nor the author shall be liable for damages arising herefrom. If professional advice or other expert assistance is required, the services of a competent professional should be sought.

*Library of Congress Cataloging-in-Publication Data*

Sakiyama, Kazuo, 1971-

Security of block ciphers : from algorithm design to hardware implementation / Kazuo Sakiyama, Yu Sasaki, Yang Li.

pages cm

Includes bibliographical references and index.

ISBN 978-1-118-66001-0 (cloth)

1. Computer security--Mathematics. 2. Data encryption (Computer science) 3. Ciphers. 4. Computer algorithms. I. Sasaki, Yu. II. Li, Yang, 1986-. III. Title.

QA76.9.A25S256 2015

005.8'2--dc23

2015019381

Typeset in 10/12pt, TimesLTStd by SPi Global, Chennai, India

# Contents

<b>Preface</b>	<b>xi</b>
<b>About the Authors</b>	<b>xiii</b>
<b>1 Introduction to Block Ciphers</b>	<b>1</b>
1.1 Block Cipher in Cryptology	1
1.1.1 <i>Introduction</i>	1
1.1.2 <i>Symmetric-Key Ciphers</i>	1
1.1.3 <i>Efficient Block Cipher Design</i>	2
1.2 Boolean Function and Galois Field	3
1.2.1 <i>INV, OR, AND, and XOR Operators</i>	3
1.2.2 <i>Galois Field</i>	3
1.2.3 <i>Extended Binary Field and Representation of Elements</i>	4
1.3 Linear and Nonlinear Functions in Boolean Algebra	7
1.3.1 <i>Linear Functions</i>	7
1.3.2 <i>Nonlinear Functions</i>	7
1.4 Linear and Nonlinear Functions in Block Cipher	8
1.4.1 <i>Nonlinear Layer</i>	8
1.4.2 <i>Linear Layer</i>	11
1.4.3 <i>Substitution-Permutation Network (SPN)</i>	12
1.5 Advanced Encryption Standard (AES)	12
1.5.1 <i>Specification of AES-128 Encryption</i>	12
1.5.2 <i>AES-128 Decryption</i>	19
1.5.3 <i>Specification of AES-192 and AES-256</i>	20
1.5.4 <i>Notations to Describe AES-128</i>	23
Further Reading	25
<b>2 Introduction to Digital Circuits</b>	<b>27</b>
2.1 Basics of Modern Digital Circuits	27
2.1.1 <i>Digital Circuit Design Method</i>	27
2.1.2 <i>Synchronous-Style Design Flow</i>	27
2.1.3 <i>Hierarchy in Digital Circuit Design</i>	29

2.2	Classification of Signals in Digital Circuits	29
2.2.1	<i>Clock Signal</i>	29
2.2.2	<i>Reset Signal</i>	30
2.2.3	<i>Data Signal</i>	31
2.3	Basics of Digital Logics and Functional Modules	31
2.3.1	<i>Combinatorial Logics</i>	31
2.3.2	<i>Sequential Logics</i>	32
2.3.3	<i>Controller and Datapath Modules</i>	36
2.4	Memory Modules	40
2.4.1	<i>Single-Port SRAM</i>	40
2.4.2	<i>Register File</i>	41
2.5	Signal Delay and Timing Analysis	42
2.5.1	<i>Signal Delay</i>	42
2.5.2	<i>Static Timing Analysis and Dynamic Timing Analysis</i>	45
2.6	Cost and Performance of Digital Circuits	47
2.6.1	<i>Area Cost</i>	47
2.6.2	<i>Latency and Throughput</i>	47
	Further Reading	48
<b>3</b>	<b>Hardware Implementations for Block Ciphers</b>	<b>49</b>
3.1	Parallel Architecture	49
3.1.1	<i>Comparison between Serial and Parallel Architectures</i>	49
3.1.2	<i>Algorithm Optimization for Parallel Architectures</i>	50
3.2	Loop Architecture	51
3.2.1	<i>Straightforward (Loop-Unrolled) Architecture</i>	51
3.2.2	<i>Basic Loop Architecture</i>	53
3.3	Pipeline Architecture	55
3.3.1	<i>Pipeline Architecture for Block Ciphers</i>	55
3.3.2	<i>Advanced Pipeline Architecture for Block Ciphers</i>	56
3.4	AES Hardware Implementations	58
3.4.1	<i>Straightforward Implementation for AES-128</i>	58
3.4.2	<i>Loop Architecture for AES-128</i>	61
3.4.3	<i>Pipeline Architecture for AES-128</i>	65
3.4.4	<i>Compact Architecture for AES-128</i>	66
	Further Reading	67
<b>4</b>	<b>Cryptanalysis on Block Ciphers</b>	<b>69</b>
4.1	Basics of Cryptanalysis	69
4.1.1	<i>Block Ciphers</i>	69
4.1.2	<i>Security of Block Ciphers</i>	70
4.1.3	<i>Attack Models</i>	71
4.1.4	<i>Complexity of Cryptanalysis</i>	73
4.1.5	<i>Generic Attacks</i>	74
4.1.6	<i>Goal of Shortcut Attacks (Cryptanalysis)</i>	77
4.2	Differential Cryptanalysis	78
4.2.1	<i>Basic Concept and Definition</i>	78

4.2.2	<i>Motivation of Differential Cryptanalysis</i>	79
4.2.3	<i>Probability of Differential Propagation</i>	80
4.2.4	<i>Deterministic Differential Propagation in Linear Computations</i>	83
4.2.5	<i>Probabilistic Differential Propagation in Nonlinear Computations</i>	86
4.2.6	<i>Probability of Differential Propagation for Multiple Rounds</i>	89
4.2.7	<i>Differential Characteristic for AES Reduced to Three Rounds</i>	91
4.2.8	<i>Distinguishing Attack with Differential Characteristic</i>	93
4.2.9	<i>Key Recovery Attack after Differential Characteristic</i>	95
4.2.10	<i>Basic Differential Cryptanalysis for Four-Round AES †</i>	96
4.2.11	<i>Advanced Differential Cryptanalysis for Four-Round AES †</i>	103
4.2.12	<i>Preventing Differential Cryptanalysis †</i>	106
4.3	<b>Impossible Differential Cryptanalysis</b>	110
4.3.1	<i>Basic Concept and Definition</i>	110
4.3.2	<i>Impossible Differential Characteristic for 3.5-round AES</i>	111
4.3.3	<i>Key Recovery Attacks for Five-Round AES</i>	114
4.3.4	<i>Key Recovery Attacks for Seven-Round AES †</i>	123
4.4	<b>Integral Cryptanalysis</b>	131
4.4.1	<i>Basic Concept</i>	131
4.4.2	<i>Processing <math>\mathcal{P}</math> through Subkey XOR</i>	132
4.4.3	<i>Processing <math>\mathcal{P}</math> through SubBytes Operation</i>	133
4.4.4	<i>Processing <math>\mathcal{P}</math> through ShiftRows Operation</i>	134
4.4.5	<i>Processing <math>\mathcal{P}</math> through MixColumns Operation</i>	134
4.4.6	<i>Integral Property of AES Reduced to 2.5 Rounds</i>	135
4.4.7	<i>Balanced Property</i>	136
4.4.8	<i>Integral Property of AES Reduced to Three Rounds and Distinguishing Attack</i>	137
4.4.9	<i>Key Recovery Attack with Integral Cryptanalysis for Five Rounds</i>	139
4.4.10	<i>Higher-Order Integral Property †</i>	141
4.4.11	<i>Key Recovery Attack with Integral Cryptanalysis for Six Rounds †</i>	143
	<b>Further Reading</b>	147
<b>5</b>	<b>Side-Channel Analysis and Fault Analysis on Block Ciphers</b>	149
5.1	<b>Introduction</b>	149
5.1.1	<i>Intrusion Degree of Physical Attacks</i>	149
5.1.2	<i>Passive and Active Noninvasive Physical Attacks</i>	151
5.1.3	<i>Cryptanalysis Compared to Side-Channel Analysis and Fault Analysis</i>	151
5.2	<b>Basics of Side-Channel Analysis</b>	152
5.2.1	<i>Side Channels of Digital Circuits</i>	152
5.2.2	<i>Goal of Side-Channel Analysis</i>	154
5.2.3	<i>General Procedures of Side-Channel Analysis</i>	155
5.2.4	<i>Profiling versus Non-profiling Side-Channel Analysis</i>	156
5.2.5	<i>Divide-and-Conquer Algorithm</i>	157
5.3	<b>Side-Channel Analysis on Block Ciphers</b>	159
5.3.1	<i>Power Consumption Measurement in Power Analysis</i>	160
5.3.2	<i>Simple Power Analysis and Differential Power Analysis</i>	163

5.3.3	<i>General Key Recovery Algorithm for DPA</i>	164
5.3.4	<i>Overview of Attack Targets</i>	169
5.3.5	<i>Single-Bit DPA Attack on AES-128 Hardware Implementations</i>	181
5.3.6	<i>Attacks Using HW Model on AES-128 Hardware Implementations</i>	186
5.3.7	<i>Attacks Using HD Model on AES-128 Hardware Implementations</i>	192
5.3.8	<i>Attacks with Collision Model</i> †	199
5.4	Basics of Fault Analysis	203
5.4.1	<i>Faults Caused by Setup-Time Violations</i>	205
5.4.2	<i>Faults Caused by Data Alteration</i>	208
5.5	Fault Analysis on Block Ciphers	208
5.5.1	<i>Differential Fault Analysis</i>	208
5.5.2	<i>Fault Sensitivity Analysis</i> †	215
	Acknowledgment	223
	Bibliography	223
<b>6</b>	<b>Advanced Fault Analysis with Techniques from Cryptanalysis</b>	<b>225</b>
6.1	Optimized Differential Fault Analysis	226
6.1.1	<i>Relaxing Fault Model</i>	226
6.1.2	<i>Four Classes of Faulty Byte Positions</i>	227
6.1.3	<i>Recovering Subkey Candidates of <math>sk_{10}</math></i>	228
6.1.4	<i>Attack Procedure</i>	230
6.1.5	<i>Probabilistic Fault Injection</i>	231
6.1.6	<i>Optimized DFA with the MixColumns Operation in the Last Round</i> †	232
6.1.7	<i>Countermeasures against DFA and Motivation of Advanced DFA</i>	236
6.2	Impossible Differential Fault Analysis	237
6.2.1	<i>Fault Model</i>	238
6.2.2	<i>Impossible DFA with Unknown Faulty Byte Positions</i>	238
6.2.3	<i>Impossible DFA with Fixed Faulty Byte Position</i>	244
6.3	Integral Differential Fault Analysis	245
6.3.1	<i>Fault Model</i>	246
6.3.2	<i>Integral DFA with Bit-Fault Model</i>	247
6.3.3	<i>Integral DFA with Random Byte-Fault Model</i>	251
6.3.4	<i>Integral DFA with Noisy Random Byte-Fault Model</i> †	254
6.4	Meet-in-the-Middle Fault Analysis	260
6.4.1	<i>Meet-in-the-Middle Attack on Block Ciphers</i>	260
6.4.2	<i>Meet-in-the-Middle Attack for Differential Fault Analysis</i>	263
	Further Reading	268
<b>7</b>	<b>Countermeasures against Side-Channel Analysis and Fault Analysis</b>	<b>269</b>
7.1	Logic-Level Hiding Countermeasures	269
7.1.1	<i>Overview of Hiding Countermeasure with WDDL Technique</i>	270
7.1.2	<i>WDDL-NAND Gate</i>	272
7.1.3	<i>WDDL-NOR and WDDL-INV Gates</i>	273
7.1.4	<i>Precharge Logic for WDDL Technique</i>	273
7.1.5	<i>Intrinsic Fault Detection Mechanism of WDDL</i>	276

---

7.2	Logic-Level Masking Countermeasures	277
7.2.1	<i>Overview of Masking Countermeasure</i>	277
7.2.2	<i>Operations on Values with Boolean Masking</i>	278
7.2.3	<i>Re-masking and Unmasking</i>	278
7.2.4	<i>Masked AND Gate</i>	279
7.2.5	<i>Random Switching Logic</i>	281
7.2.6	<i>Threshold Implementation</i>	283
7.3	Higher Level Countermeasures	285
7.3.1	<i>Algorithm-Level Countermeasures</i>	286
7.3.2	<i>Architecture-Level Countermeasures</i>	289
7.3.3	<i>Protocol-Level Countermeasure</i>	290
	Bibliography	291
	<b>Index</b>	<b>293</b>



# Preface

The main purpose of this book is to offer a fundamental understanding of security and its implementation of block ciphers. Nowadays, research fields in computer science and engineering have a vast scope and cryptology deals with various topics in information security. In order to understand the cutting-edge technology and science that underlies cryptology, block cipher is one of the best-suited targets both from theoretical and practical points of view. In order to offer the learning materials to fill the gap between theory and practice of the security of block ciphers, our focus goes to cryptanalysis, side-channel analysis, and fault analysis against block ciphers rather than covering all the security issues of block ciphers. AES is currently one of the most researched block ciphers in academia and widely used both in government and in commerce. Considering this fact, the explanations in this book are mainly oriented to the security of AES. In addition, AES is one of the best choices to build up all the discussions from algorithm design to hardware implementation, which is very helpful for readers to follow and to understand the basic ideas that can apply to other block ciphers.

## Book Organization

This book is intended as a textbook for undergraduate and graduate students to have a big picture understanding of block ciphers from algorithm to implementations. The contents also include essential knowledge that is useful for cryptographers who are not familiar with hardware, and hardware researchers who are not familiar with the security of block ciphers. This book consists of seven chapters, and each chapter is written by the main authors listed in Table 1.

**Table 1** Main Author

Chapter Number: Chapter Title	KS	YS	YL
1: Introduction to Block Ciphers	X	X	
2: Introduction to Digital Circuits	X		
3: Hardware Implementations for Block Ciphers	X		
4: Cryptanalysis on Block Ciphers		X	
5: Side-Channel Analysis and Fault Analysis on Block Ciphers	X		X
6: Advanced Fault Analysis with Techniques from Cryptanalysis		X	
7: Countermeasures against Side-Channel Analysis and Fault Analysis	X		X

For the purpose of helping readers to understand the chapters, we have prepared several exercises. Some exercises are easy, and suitable for testing the comprehension of each individual learner. Some exercises are moderately difficult, and therefore readers might consider working in a small group as they would on a mini project.

There are several (sub)sections whose titles have a mark “†” at the end. They require knowledge about advanced-level techniques to understand and implement the analysis methods. Readers who find it difficult to follow them are recommended to skip them at the first reading, and focus on understanding the essential concepts of cryptanalysis and side-channel analysis from other sections.

We hope that the readers will enjoy the world of block cipher security and open new horizons through this fantastic field of study.

*Kazuo Sakiyama*

*Yu Sasaki*

*Yang Li*

# About the Authors

**Kazuo Sakiyama** is currently a faculty member in the Department of Informatics at the University of Electro-Communications, Tokyo. He received his Ph.D. degree in electrical engineering from the Katholieke Universiteit Leuven, Belgium in 2007. From 1996 to 2004, he was with the Semiconductor and IC Division, Hitachi, Ltd., and engaged in designing system-on-chip LSIs. His current research interests include information security, hardware security, and security analysis of cryptographic modules.

**Yu Sasaki** received his Ph.D. degree in engineering from the University of Electro-Communications, Tokyo, in 2010. He is currently a member of NTT Secure Platform Laboratories. He has been working with NTT from 2005. His current research interests include cryptography, especially for design and security analysis of symmetric-key cryptography.

**Yang Li** received his Ph.D. degree in engineering from the Faculty of Informatics and Engineering of the University of Electro-Communications, Tokyo, in 2012. He is currently an associated professor in College of Computer Science and Technology at Nanjing University of Aeronautics and Astronautics, China. His main research interests include security evaluation and improvement for cryptographic hardware and embedded systems.



# 1

## Introduction to Block Ciphers

### 1.1 Block Cipher in Cryptology

#### 1.1.1 Introduction

Information includes our private data that we desire to protect from unwilling leakage depending on the application. **Cryptology** is a field of research that offers appropriate solutions for the data protection by exploring how to construct a secure communication for fair information exchange. Modern cryptology often deals with digitalized data rather than analog data that cannot be expressed simply with a series of 0s and 1s. In our daily life, information is exchanged by digital devices such as radio frequency identification (RFID) tags, smart cards, and smart phones, where a computational resource is limited. Therefore, it is one of the most important challenges in cryptology to realize an efficient implementation of **cryptosystems**.

#### 1.1.2 Symmetric-Key Ciphers

There are various ways to realize **encryption** that is a kind of computational process for information to be protected. In a symmetric-key cipher, information is encrypted with a secret key, and it is expected that the owner of the secret key can decrypt the encrypted information correctly. For instance, let us see the situation, where Alice would like to send a **message** to Bob in a secure way. If the secret key,  $K$ , is shared only with Alice and Bob, only Bob can decrypt the message from the encrypted message. The original and the encrypted messages are called **plaintext** and **ciphertext**, respectively. Figure 1.1 illustrates the encryption and decryption processes.

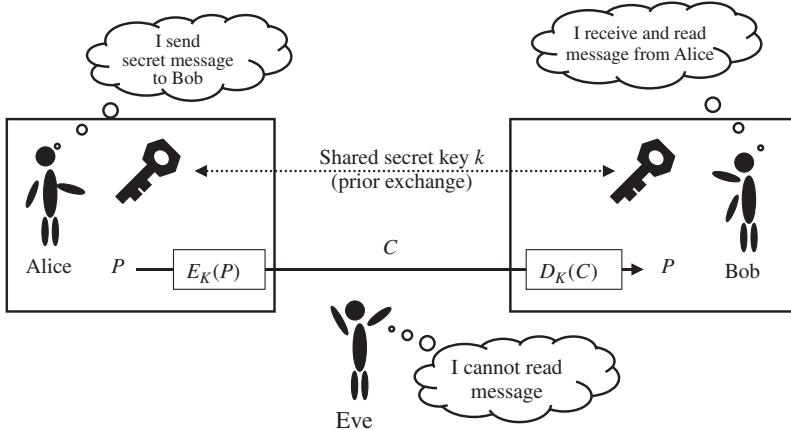
The encryption by Alice can be written as

$$C = E_K(P). \quad (1.1)$$

The ciphertext is decrypted by Bob as

$$P = D_K(C). \quad (1.2)$$

Only Bob can decrypt and read the message, and Eve, who does not own the secret key, cannot decrypt it.



**Figure 1.1** Basic model for a symmetric-key cryptosystem

Alice and Bob need to compute the cryptographic operations based on the functions,  $E_K(\cdot)$  and  $D_K(\cdot)$ . The simpler the functions are, the more efficiently they can compute. For instance, **Vernam cipher**, invented in 1917, uses just **XOR** operations as

$$C = P \oplus K, \quad P = C \oplus K \quad (1.3)$$

to convert plaintext and ciphertext. The XOR operation is explained in Section 1.2.1.

However, in order to guarantee the security, that is, in order that Eve cannot obtain any information of message from  $C$ , the secret key needs to be refreshed with a random number for each encryption/decryption. In other words, in order to communicate securely with the Vernam cipher, a very long key, which is the same size as  $M$ , is required. This is significantly inefficient. In general, encryption and decryption processes are based on the trade-offs between cost, performance, and security.

### 1.1.3 Efficient Block Cipher Design

The fundamental idea to achieve an efficient encryption scheme is designing a fixed-input size encryption scheme, and iteratively applying this scheme to encrypt arbitrary length messages. Such a fixed-input size encryption scheme is called **block cipher**, and the group of bits with the fixed-input size is called **block**. If the unit of operation is small enough, for example, 1 bit or 1 byte, such a symmetric-key cipher is called stream cipher. As block ciphers are expected to compute encryption and decryption efficiently, they have an iterated structure, and repeat the same function several times. Such a function is called **round function**. The iterated structure contributes to achieving a small program code in software and implementing a compact circuit design in hardware.

Modern block ciphers are mainly categorized into two kinds: Feistel structure and **substitution-permutation network (SPN)** structure. Feistel structure was employed in data encryption standard (DES) block cipher proposed in 1977. Including FEAL and Camellia, the Feistel structure has been employed by many block ciphers.

On the contrary, **Advanced Encryption Standard (AES)** employed SPN structure. AES is the main target of this book as it is one of the most widely used block ciphers, and it contains fundamental ideas of SPN structure. The basic mathematics to understand SPN structure and AES specification will be explained later in this chapter.

## 1.2 Boolean Function and Galois Field

**Boolean functions** are used in most of the block ciphers including AES. A Boolean function,  $f$ , is described as

$$f : \{0, 1\}^n \rightarrow \{0, 1\}, \quad (1.4)$$

where  $\{0, 1\}$  is called **Boolean domain** and  $\{0, 1\}^n$  is the set of all  $n$ -tuples  $(x_1, \dots, x_n)$ , where  $x_1, \dots, x_n$  are all in Boolean domain.<sup>1</sup>

### 1.2.1 INV, OR, AND, and XOR Operators

The most simple Boolean function is **inversion** or the **INV** operation that is a bit complement. It operates as

$$\neg x = \begin{cases} 1 & (x = 0), \\ 0 & (x = 1), \end{cases} \quad (1.5a)$$

$$(1.5b)$$

where  $\neg$  is used for representing the INV operation. Alternatively, the logic symbol,  $\bar{\phantom{x}}$ , is also used for INV. In this book, we allow both usage, that is,  $\neg x = \bar{x}$ .

For the case of  $n = 2$ , representative Boolean functions are **OR**, **AND**, and **XOR**. OR is defined as

$$x \vee y = \begin{cases} 0 & (x = y = 0), \\ 1 & (\text{else}). \end{cases} \quad (1.6a)$$

$$(1.6b)$$

Likewise, AND and XOR are defined, respectively, as

$$x \wedge y = \begin{cases} 1 & (x = y = 1), \\ 0 & (\text{else}), \end{cases} \quad (1.7a)$$

$$x \oplus y = \begin{cases} 0 & (x = y), \\ 1 & (x \neq y). \end{cases} \quad (1.8a)$$

$$(1.8b)$$

“ $\vee$ ,” “ $\wedge$ ,” and “ $\oplus$ ” are used for representing OR, AND, and XOR operations.

The **truth table** for OR, AND, and XOR is described in Table 1.1.

### 1.2.2 Galois Field

**Finite field** or **Galois field** deals with a finite number of elements. Over a Galois filed, addition, subtraction, multiplication, and division are defined. Galois field with the smallest order is

---

<sup>1</sup> For the case  $n = 0$ , Boolean function denotes a constant, 0 or 1.

**Table 1.1** Truth table for basic operators

$x$	$y$	$x \vee y$	$x \wedge y$	$x \oplus y$
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	0

**Table 1.2** Operations over  $GF(2)$ 

$x$	$y$	$x + y$	$x \times y$	$-x$	$x^{-1}$
0	0	0	0	0	–
0	1	1	0	0	–
1	0	1	0	1	1
1	1	0	1	1	1

called a binary field or  $GF(2)$ . For instance, addition, multiplication, **additive inverse**, and **multiplicative inverse** over  $GF(2)$  are defined in Table 1.2.

As can be found from Tables 1.1 and 1.2, addition and multiplication over  $GF(2)$  are realized, respectively, with XOR and AND.

**Exercise 1.1** Complete Table 1.3, that is, for addition, multiplication, additive inverse, and multiplicative inverse over  $GF(5)$ .

### 1.2.3 Extended Binary Field and Representation of Elements

**Binary field**,  $GF(2)$ , can be extended to a large field size called **extended binary field**,  $GF(2^n)$ , where  $n$  is a positive integer. Especially, in the case of AES, operations in  $GF(2^8)$  are of special interest. The number of elements of  $GF(2^n)$  is  $2^n$ . There are several different representations for the elements, which affect the cost and speed performance of software and hardware implementations.

#### 1.2.3.1 Polynomial Basis Representation

As the number of elements of  $GF(2^n)$  is a power of 2, each bit of the binary representation can be used for each coefficient of a polynomial whose degree is  $n - 1$ . Any element in  $GF(2^n)$  can be expressed with the so-called **polynomial basis** as

$$a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_0, \quad (1.9)$$

**Table 1.3** Operations over  $GF(5)$ 

$x$	$y$	$x + y$	$x \times y$	$-x$	$x^{-1}$
0	0	0	0	0	1
0	1	1	0	4	4
0	2	2	0	3	3
0	3	3	0	2	2
0	4	4	0	1	1
1	0	1	0	4	4
1	1	2	0	3	3
1	2	3	0	2	2
1	3	4	0	1	1
1	4	0	0	0	0
2	0	2	0	3	3
2	1	3	0	2	2
2	2	4	0	1	1
2	3	0	0	0	0
2	4	1	0	4	4
3	0	3	0	2	2
3	1	4	0	1	1
3	2	0	0	0	0
3	3	1	0	4	4
3	4	2	0	3	3
4	0	4	0	1	1
4	1	0	0	0	0
4	2	1	0	3	3
4	3	2	0	2	2
4	4	3	0	1	1

where  $a_i \in \{0, 1\}$ . For instance, 16 elements in  $GF(2^4)$  can be expressed with the binary representation,  $(a_3, a_2, a_1, a_0)_2$ . By assigning each bit to the coefficient of a polynomial of  $x$ , we have  $a_3x^3 + a_2x^2 + a_1x + a_0$ . Addition of two field elements, for example,  $(x + 1) + (x^3 + 1)$ , can be calculated as

$$(x + 1) + (x^3 + 1) = x^3 + x, \quad (1.10)$$

as  $1 + 1 = 0$  over  $GF(2)$ .

Multiplication of the two field elements, for example,  $(x + 1)(x^3 + 1)$ , needs modular reduction with an **irreducible polynomial**, for example,  $x^4 + x^3 + 1$ , which specifies the field.<sup>2</sup> Therefore, the multiplication result becomes as

$$(x + 1)(x^3 + 1) \equiv x^4 + x^3 + x + 1 \equiv x \pmod{(x^4 + x^3 + 1)}. \quad (1.11)$$

### 1.2.3.2 Normal Basis Representation

Alternatively, elements in  $GF(2^n)$  are described using **normal basis** as

$$b_{n-1}\alpha^{2^{n-1}} + b_{n-2}\alpha^{2^{n-2}} + \cdots + b_0\alpha^{2^0}, \quad (1.12)$$

---

<sup>2</sup> In this case, we also use the expression,  $GF(2)[x]/(x^4 + x^3 + 1)$ .

where  $b_i \in \{0, 1\}$  and  $\alpha$  are roots of an irreducible polynomial,  $P(x)$ , that is,

$$P(\alpha) = 0. \quad (1.13)$$

Furthermore,

$$\alpha^{2^n-1} \equiv 1 \pmod{P(\alpha)}. \quad (1.14)$$

This can be confirmed by Fermat little theorem.

For the case of  $GF(2^4)$ , suppose that  $P(x) = x^4 + x^3 + 1$ , that is,  $P(\alpha) = \alpha^4 + \alpha^3 + 1 = 0$ . Addition in the normal basis representation of  $\alpha^7 + \alpha^{11}$  can be calculated simply by XORing each coefficient of two elements in the form of Equation (1.12). That is,

$$\alpha^7 + \alpha^{11} = (\alpha^8 + \alpha^4) + (\alpha^4 + \alpha^2) = \alpha^8 + \alpha^2 = \alpha^{10}, \quad (1.15)$$

where the normal basis representations of  $\alpha^7$  and  $\alpha^{11}$  can be found in Table 1.4.

This is correct as  $\alpha^7 + \alpha^{11} = \alpha^7(1 + \alpha^4) = \alpha^{10}$ . By using the fact of  $\alpha^{15} = 1$ , multiplication in  $GF(2^4)$ , for example,  $\alpha^7\alpha^{11}$  is calculated as

$$\alpha^7\alpha^{11} = \alpha^{18} = \alpha^3. \quad (1.16)$$

The most advantageous point to use the normal basis representation lies in the fact that **squaring** is easy to compute in  $GF(2^n)$ . As can be found in Table 1.4, squaring for  $(b_3, b_2, b_1, b_0)$  is  $(b_2, b_1, b_0, b_3)$ . More precisely, in squaring, the elements in the normal basis representation are derived as

$$(b_{n-1}\alpha^{2^{n-1}} + b_{n-2}\alpha^{2^{n-2}} + \cdots + b_0\alpha^{2^0})^2 \quad (1.17)$$

$$= b_{n-1}\alpha^{2^n} + b_{n-2}\alpha^{2^{n-1}} + \cdots + b_0\alpha^{2^1} \quad (1.18)$$

$$= b_{n-2}\alpha^{2^{n-1}} + \cdots + b_0\alpha^{2^1} + b_{n-1}\alpha^{2^0}. \quad (1.19)$$

**Table 1.4** Representations of elements for irreducible polynomial  $x^4 + x^3 + 1$  in  $GF(2^4)$

Binary $(a_3, a_2, a_1, a_0)_2$	Bit concatenation	Hex.	Polynomial basis	Power of $\alpha$	Normal basis $(b_3, b_2, b_1, b_0)$
(0, 0, 0, 0)	0  0  0  0	0	0	0	(0, 0, 0, 0)
(0, 0, 0, 1)	0  0  0  1	1	1	1	(1, 1, 1, 1)
(0, 0, 1, 0)	0  0  1  0	2	$x$	$\alpha$	(0, 0, 0, 1)
(0, 1, 0, 0)	0  1  0  0	4	$x^2$	$\alpha^2$	(0, 0, 1, 0)
(1, 0, 0, 0)	1  0  0  0	8	$x^3$	$\alpha^3$	(1, 0, 1, 1)
(1, 0, 0, 1)	1  0  0  1	9	$x^3 + 1$	$\alpha^4$	(0, 1, 0, 0)
(1, 0, 1, 1)	1  0  1  1	b	$x^3 + x + 1$	$\alpha^5$	(0, 1, 0, 1)
(1, 1, 1, 1)	1  1  1  1	f	$x^3 + x^2 + x + 1$	$\alpha^6$	(0, 1, 1, 1)
(0, 1, 1, 1)	0  1  1  1	7	$x^2 + x + 1$	$\alpha^7$	(1, 1, 0, 0)
(1, 1, 1, 0)	1  1  1  0	e	$x^3 + x^2 + x$	$\alpha^8$	(1, 0, 0, 0)
(0, 1, 0, 1)	0  1  0  1	5	$x^2 + 1$	$\alpha^9$	(1, 1, 0, 1)
(1, 0, 1, 0)	1  0  1  0	a	$x^3 + x$	$\alpha^{10}$	(1, 0, 1, 0)
(1, 1, 0, 1)	1  1  0  1	d	$x^3 + x^2 + 1$	$\alpha^{11}$	(0, 1, 1, 0)
(0, 0, 1, 1)	0  0  1  1	3	$x + 1$	$\alpha^{12}$	(1, 1, 1, 0)
(0, 1, 1, 0)	0  1  1  0	6	$x^2 + x$	$\alpha^{13}$	(0, 0, 1, 1)
(1, 1, 0, 0)	1  1  0  0	c	$x^3 + x^2$	$\alpha^{14}$	(1, 0, 0, 1)

This merit is often used in both software and hardware implementations. However, in general, implementing modular multiplication in the normal basis requires more computation than that in the polynomial basis. Hereafter, we mainly use polynomial basis representation.

### 1.3 Linear and Nonlinear Functions in Boolean Algebra

#### 1.3.1 Linear Functions

Addition and multiplication by a constant are **linear functions** in  $GF(2^n)$ . Suppose that  $A(x) = a_{n-1}x^{n-1} + \dots + a_0$  and  $B(x) = b_{n-1}x^{n-1} + \dots + b_0$ , where  $a_i, b_i \in \{0, 1\}$ . Addition of  $A(x)$  and  $B(x)$  is

$$A(x) + B(x) = (a_{n-1} \oplus b_{n-1})x^{n-1} + \dots + a_0 \oplus b_0. \quad (1.20)$$

From the fact that  $a_i \oplus b_i \in \{0, 1\}$ , it is confirmed that addition in  $GF(2^n)$  is a linear function.

For multiplication by a constant  $B$ , there exist  $c_{n-1}, \dots, c_0 \in \{0, 1\}$  such that

$$A(x) \times B = c_{n-1}x^{n-1} + \dots + c_0. \quad (1.21)$$

Therefore, we know that such multiplication in  $GF(2^n)$  is also a linear function. It can be easily understood considering the fact that multiplication by a constant can be computed with multiple additions of  $A(x)$  in  $GF(2^n)$ .

**Exercise 1.2** Suppose that  $A(x) = x^3 + x^2$  and  $B(x) = x^3 + x$  are represented in the polynomial basis. Calculate  $A(x) + B(x)$ ,  $2A(x)$ , and  $3B(x)$  in  $GF(2^4)$  when the irreducible polynomial is  $x^4 + x^3 + 1$ . Note that 2 and 3 are hexadecimal representations of  $x$  and  $x + 1$ , respectively.

**Exercise 1.3** Confirm that modular additive inverse is a linear function.

#### 1.3.2 Nonlinear Functions

On the contrary, (normal) modular multiplication and multiplicative inverse in  $GF(2^n)$  are **nonlinear functions**. The AES block cipher uses a nonlinear function in a part of the design that is based on modular multiplicative inversion in  $GF(2)[x]/x^8 + x^4 + x^3 + x + 1$ . The multiplicative inverse computation can be done with Fermat's (little) theorem as

$$a^{-1} \equiv a^{2^8 - 2} \equiv a^{254}, \quad (1.22)$$

for  $a \neq 0$ . In AES, multiplicative inverse of 0 is mapped to 0.

One of the most optimal ways to compute the inversion is to find **addition chain**. On the basis of the Itoh–Tsujii algorithm, the computation can be performed with four multiplications and seven modular squarings as

$$\left\{ \begin{array}{l} (a)^2 = a^2, \\ a^2a = a^3, \end{array} \right. \quad (1.23a)$$

$$\left\{ \begin{array}{l} (a^3)^2 = a^{12}, \\ a^{12}a^3 = a^{15}, \end{array} \right. \quad (1.23b)$$

$$\left\{ \begin{array}{l} (a^3)^2 = a^{12}, \\ a^{12}a^3 = a^{15}, \end{array} \right. \quad (1.23c)$$

$$\left\{ \begin{array}{l} (a^{15})^2 = a^{240}, \\ a^{240}a^2a^{12} = a^{254}. \end{array} \right. \quad (1.23d)$$

$$\left\{ \begin{array}{l} (a^{15})^2 = a^{240}, \\ a^{240}a^2a^{12} = a^{254}. \end{array} \right. \quad (1.23e)$$

$$\left\{ \begin{array}{l} (a^{15})^2 = a^{240}, \\ a^{240}a^2a^{12} = a^{254}. \end{array} \right. \quad (1.23f)$$

Itoh–Tsujii algorithm utilizes the relationship of

$$a^{2^{2^t}-1} = (a^{2^{2^{t-1}}-1})^{2^{2^{t-1}}} (a^{2^{2^{t-1}}-1}). \quad (1.24)$$

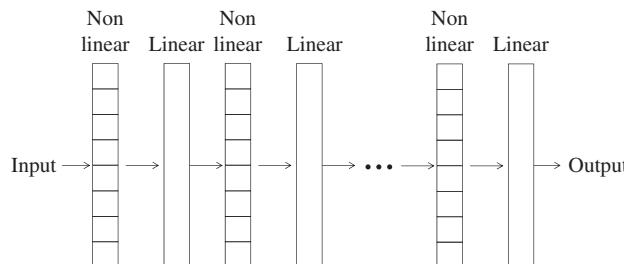
## 1.4 Linear and Nonlinear Functions in Block Cipher

As discussed in Section 1.3, logical operations are classified into linear operations and nonlinear operations. Composition of linear operations is also linear. Hence, if all the cipher's operations are linear, the resulting cipher is also linear, which is insecure. In order to break the linearity of the cipher, nonlinear operations need to be introduced. However, in general, the cost of implementing nonlinear operations is more expensive than the one for linear operations.

The strategy of the block cipher design is alternately applying nonlinear and linear operations several times. To avoid the heavy cost, nonlinear operation is designed to be weak but its cost is small. In many cases, a nonlinear operation is designed to be operated on a smaller size than the block size, and the operation is applied in parallel to all the data. Then, in order to compensate the weak nonlinear computations, a linear operation mixes the entire block. The strategy is depicted in Figure 1.2. In the following, each of the nonlinear layer and linear layer is further detailed.

### 1.4.1 Nonlinear Layer

In order to reduce the implementation cost, a nonlinear operation is designed to work on a fraction of the data. Typical choices of the size are 64 bits, 32 bits, 8 bits (called byte),



**Figure 1.2** Block cipher design strategy. Nonlinear operations and linear operations are alternately applied

**Table 1.5** An example of 4-bit to 4-bit S-box,  $S(\cdot)$ 

Input	$x$	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
Output	$S(x)$	c	0	f	a	2	b	9	5	8	3	d	7	1	e	6	4

All values are described in the hexadecimal format.

4 bits (called nibble), and 1 bit. The size of the nonlinear operation is determined depending on the following two aspects.

- type of nonlinear operation
- target platform in which the cipher is implemented.

#### 1.4.1.1 Modular Operation

When the cipher is designed for being used in high-end CPUs, the implementation cost is not a big issue but the operation should be optimized for instructions adopted in such a CPU. Currently, many CPUs operate on 64 or 32 bits, thus the size of the nonlinear operation is also adjusted to 64 or 32 bits. The high-end CPUs can perform the modular addition or subtraction efficiently. The nonlinearity is often introduced by addition or subtraction on modulo  $2^{64}$  or  $2^{32}$ .

#### 1.4.1.2 Substitution Table (S-box)

When the cipher is designed for more resource-constrained hardwares such as micro-controllers, the balance of the implementation cost and the computation efficiency is important. When the CPU register size is smaller than 32 bits, the 32- or 64-bit modular addition cannot be performed efficiently. The hardware implementation also faces some problems for those operations. Typical choices of the size of the nonlinear operation are 8 or 4 bits. Because the size is small, using the substitution table is a popular approach to introduce the nonlinearity. The **substitution table**, or **S-box**, is a pre-specified mapping from the input values to the output values. An example of 4-bit to 4-bit S-box is given in Table 1.5.

**Exercise 1.4** Answer the output value of the following computations.

1.  $S(2)$
2.  $S(a)$
3.  $S(2) \oplus a$
4.  $S(2 \oplus a)$
5.  $S(2) \oplus S(a)$
6.  $S(S(2) \oplus S(a))$

**Exercise 1.5** Prove that any 1-bit to 1-bit bijective S-box is a linear mapping rather than nonlinear mapping.

In this S-box, the input value 5 is transformed to b according to the table. A 4-bit to 4-bit S-box is implemented only with  $16 \times 4 = 64$  bits of memory, which is very small. An 8-bit to 8-bit S-box is implemented only with  $256 \times 8 = 2048$  bits of memory, which is bigger than the 4-bit to 4-bit S-box but can mix the data faster than the 4-bit to 4-bit S-box.

#### 1.4.1.3 Boolean Function

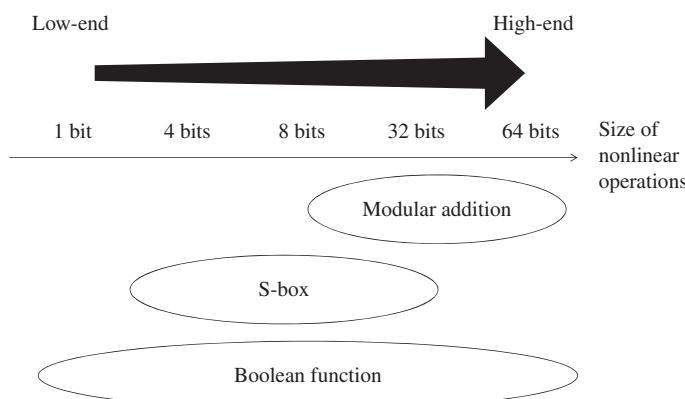
A Boolean function is the smallest tool to introduce the nonlinearity. By using an AND or OR operation, the nonlinearity is introduced in 1 bit. When the cipher is designed to be a very resource constraint environment such as RFID, a Boolean function is a typical choice as a source of the nonlinearity. A Boolean function can also fit the high-end CPUs. Thirty two-bit CPUs can operate bit-wise for each of the 32 bits in parallel. If this is combined with modular additions (not bit-wise), the nonlinearity can be introduced quickly.

It is also a popular approach to specify the input and output correspondence of some Boolean functions as an S-box. If the cipher is implemented with some memory, the S-box can be implemented, and the nonlinearity of several bits can be introduced with 1 table look-up. If the cipher is implemented with small hardware, the logic of the Boolean function is implemented to minimize the implementation cost.

#### 1.4.1.4 Balanced Choice

Unfortunately, there is no obvious choice that shows the overwhelming performance in any implementation environment. When the cipher is designed in multi-platforms, that is, both the high- and low-end environment, an S-box maybe chosen as the source of nonlinearity that shows a relatively good performance in both the environments. The popular choices of the nonlinear operations are summarized in Figure 1.3.

Note that the data is mixed by alternately applying a nonlinear operation and a linear operation. The choice of the nonlinear operation also depends on the choice of the linear operation.



**Figure 1.3** Substitution-permutation network. Popular choices of size and type of nonlinear operations

### 1.4.2 Linear Layer

The purpose of the linear layer is mixing all the output data from the nonlinear layer in which the data is updated in a small part independently. The linear layer is required to be performed efficiently and implemented lightly.

One of the simplest linear operations is XOR. A part of the nonlinear layer output is XORed to another part to mix the data from different parts. The XOR operation can be performed several times between different parts to mix the data more.

The bit-rotation and bit-shift are also simple linear operations. For example, by applying the 1-bit rotation to the entire data, 1-bit from each part will be moved to the next part. The XOR, bit-shift, and bit-rotation can be implemented efficiently in various platforms, thus they are suitable for the block cipher design.

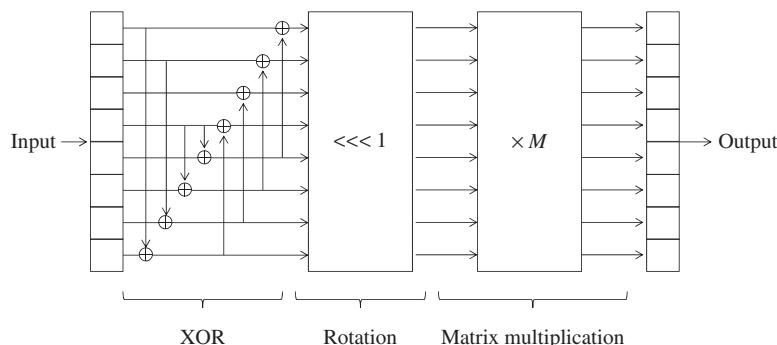
Another important example is a multiplication over a finite field or modular multiplication. Suppose that the size of the nonlinear operation is  $n$  bits and each bit of  $n$ -bit value represents each coefficient of a polynomial whose degree is  $n - 1$ . As explained in Section 1.3, multiplication over a finite field with some irreducible polynomial  $P(x)$  is a linear function. Suppose that the entire data consists of  $m$  parts of  $n$ -bit data, that is, its size is  $mn$  bits. The purpose of the linear function is mixing  $m$  independent outputs from the nonlinear layer. In order to mix all the  $m$  outputs,  $m \times m$  matrices are often used.

For instance, when  $m = 4$ , four  $n$ -bit values  $x_0, x_1, x_2, x_3$  are updated to four  $n$ -bit values  $y_0, y_1, y_2, y_3$  by the following matrix operation:

$$\begin{bmatrix} c_0 & c_4 & c_8 & c_{12} \\ c_1 & c_5 & c_9 & c_{13} \\ c_2 & c_6 & c_{10} & c_{14} \\ c_3 & c_7 & c_{11} & c_{15} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix}, \quad (1.25)$$

where each  $c_i$  is a constant number.

Any combination of linear operations is a linear operation. A popular design approach is combining different types of light linear operations to introduce a strong mixing effect. An example of the linear layer is depicted in Figure 1.4.



**Figure 1.4** An example of linear layer consisting of three linear operations. Nonlinear layer is supposed to update data in eight parts independently

#### 1.4.2.1 Maximum Distance Separable Matrix (MDS Matrix)

A **maximum distance separable** matrix (in short **MDS** matrix) is a matrix with some special property useful for **block cipher's design**. Considering the usage in block cipher AES, only the case with the **same input and output size is discussed here**. Let  $x$  be the  $m$ -component input to the matrix,  $M$ , and  $y$  be the  $m$ -component output from the matrix, that is,  $y = Mx$ . The matrix  $M$  is called **MDS** if no distinct input-output pairs  $(x, y)$  collide in  $m$  or more components.

For the application to cryptology, the fact that at least  $m + 1$  components differ in distinct pairs of  $(x, y)$  is important. In other words, the MDS matrix guarantees a certain amount of change in different input and output values. For instance, suppose that the value of  $x$  is slightly modified to  $x'$ , which differs only 1 bit from  $x$ , and the corresponding output value  $y'$  is computed. The multiplication by the MDS matrix can guarantee that **all the  $m$  components of the outputs  $y$  and  $y'$  have different values**, meaning that the 1-bit change of the input always changes all the  $m$  components of the output.

#### 1.4.3 Substitution-Permutation Network (SPN)

Substitution-permutation network, which is often called **SPN**, is a design approach to mix a fixed-length input data. SPN is a special form of the iterative application of nonlinear and linear computations.

The substitution layer (or S-layer), which applies a nonlinear operation, is supposed to be an S-box application in a small size. The permutation layer (or P-layer) applies a linear operation to mix the results of the S-layer efficiently.

The SPN structure is adopted in many block ciphers. AES, which is a main target of this book, also adopts the SPN structure.

### 1.5 Advanced Encryption Standard (AES)

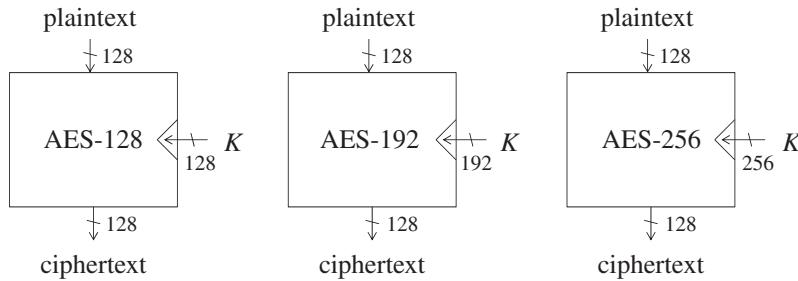
**AES** is the most widely used block cipher in present time in both governmental and commercial purposes. AES is standardized internationally, and a lot of academic researches and industrial developments have been proposed about AES. This section explains the specification of AES.

The block cipher AES supports three different key sizes: 128 bits, 192 bits, and 256 bits. The corresponding AES algorithms are called **AES-128**, **AES-192**, and **AES-256**, respectively. AES supports a fixed block size: 128 bits. That is to say, when the key is determined, AES provides a bijective map from 128-bit plaintext to 128-bit ciphertext, that is, for a key  $K$ ,  $\text{AES-128}_K$ ,  $\text{AES-192}_K$ ,  $\text{AES-256}_K : \{0, 1\}^{128} \rightarrow \{0, 1\}^{128}$  (Figure 1.5).

#### 1.5.1 Specification of AES-128 Encryption

In high level, the 128-bit key  $K$  is expanded to eleven 128-bit **subkeys**  $sk_0, sk_1, \dots, sk_{10}$  according to the **key schedule function**, or **KSF**.

1. The 128-bit key  $K$  is set to the first 128-bit subkey  $sk_0$ .
2. The KSF is computed to update 128-bit subkey  $sk_0$  to another 128-bit subkey  $sk_1$ .



**Figure 1.5** Three algorithms of AES

3. Similarly, the KSF is iterated nine times. In each time, 128-bit subkey  $sk_{i-1}$  is updated to another 128-bit subkey  $sk_i$  for  $i = 2, 3, \dots, 10$ .

Then, a plaintext is encrypted to a ciphertext as follows:

1. An XOR of the plaintext and the first subkey  $sk_0$  is computed, and this value is set to a 128-bit internal state value  $state_1$ . This operation is often called **whitening**.
2. The 128-bit internal state value  $state_1$  is updated to  $state_2$  by computing a round function, which also takes as input subkey  $sk_1$ . This operation is called **round 1 or the first round**.
3. The round function is iterated nine times to update the internal state value  $state_2$  to  $state_3, state_4, \dots, state_{11}$ . In round  $i$ , where  $i = 2, 3, \dots, 10$ , the round function takes as input  $(state_i, sk_i)$  and outputs  $state_{i+1}$ . Note that the round function in the last round is slightly different from the other rounds. The last state that is  $state_{11}$  is the ciphertext.

The computation structure of AES-128 in a function level is described in Figure 1.6.

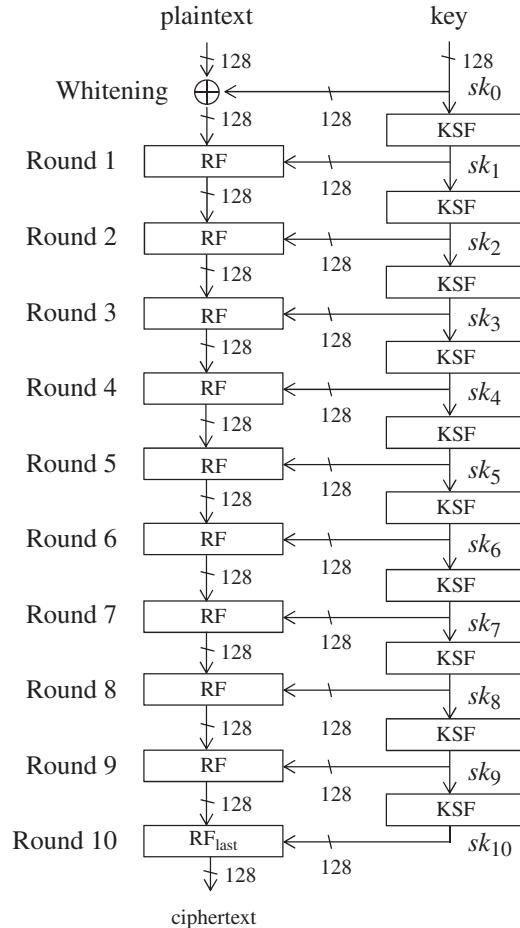
In practice, it is not necessary to compute all the 11 subkeys at the very beginning. For example, the last subkey will not be used until the very end of the encryption process. Thus, generating the last subkey and keeping it in a register is a waste of computation resource. In order to minimize the computation resource, the KSF and the round function updates are computed in parallel round by round. The AES-128 encryption algorithm in the function level can be described as Algorithm 1.1.

### 1.5.1.1 Preliminaries to Describe Computation Details

In AES, **byte** represents 8-bit values. AES is a byte-oriented cipher. All operations are defined at byte level. Let  $v$  be a byte value and  $v_7||v_6||v_5||v_4||v_3||v_2||v_1||v_0$  be its bit-wise representation, of which the corresponding vector representation is  $(v_7 v_6 v_5 v_4 v_3 v_2 v_1 v_0)_2$ . In AES, each bit of a byte represents coefficients of polynomial of  $GF(2^8)$ :

$$v_7x^7 + v_6x^6 + v_5x^5 + v_4x^4 + v_3x^3 + v_2x^2 + v_1x + v_0. \quad (1.26)$$

A byte value can be represented in hexadecimal. For example, the byte 9b represents the polynomial  $x^7 + x^4 + x^3 + x + 1$ .



**Figure 1.6** High-level computation structure of the encryption of AES-128. RF and KSF denote the round function and KSF, respectively. RF<sub>last</sub> is the last round function, which is different from the other rounds

---

**Algorithm 1.1** AES-128 Encryption Algorithm in the Function Level

---

**Input:** Plaintext  $P$ , 128-bit key  $K$ , round function RF, the last round function RF<sub>last</sub>, key schedule function KSF

**Output:** Ciphertext  $C$

```

1:  $sk_0 \leftarrow K;$ 
2:  $state_1 \leftarrow P \oplus sk_0;$ 
3: for  $i = 1, 2, \dots, 9$  do
4:    $sk_i \leftarrow KSF(sk_{i-1});$ 
5:    $state_{i+1} \leftarrow RF(state_i, sk_i);$ 
6: end for
7:  $C \leftarrow RF_{last}(state_{10}, sk_{10});$ 
8: return  $C;$ 

```

---

### Addition

Addition of two bytes,  $v_7\|v_6\|v_5\|v_4\|v_3\|v_2\|v_1\|v_0$  and  $u_7\|u_6\|u_5\|u_4\|u_3\|u_2\|u_1\|u_0$ , returns

$$(v_7 \oplus u_7)\|(v_6 \oplus u_6)\|(v_5 \oplus u_5)\|(v_4 \oplus u_4)\|(v_3 \oplus u_3)\|(v_2 \oplus u_2)\|(v_1 \oplus u_1)\|(v_0 \oplus u_0). \quad (1.27)$$

### Multiplication

Multiplication in  $GF(2^8)$  corresponds with multiplication of polynomials modulo, an irreducible binary polynomial of degree 8. The irreducible polynomial of AES is defined as

$$P(x) = x^8 + x^4 + x^3 + x + 1. \quad (1.28)$$

Because the multiplication by  $v(x) \cdot 02$  and  $v(x) \cdot 03$  is later introduced inside the round function, more details of the operation  $v(x) \cdot 02$  are explained here.  $v(x) \cdot 02$  is written as

$$(v_7x^7 + v_6x^6 + v_5x^5 + v_4x^4 + v_3x^3 + v_2x^2 + v_1x + v_0) \cdot x \quad (1.29)$$

$$= v_7x^8 + v_6x^7 + v_5x^6 + v_4x^5 + v_3x^4 + v_2x^3 + v_1x^2 + v_0x. \quad (1.30)$$

When  $v_7 = 0$ , the result is  $v_6\|v_5\|v_4\|v_3\|v_2\|v_1\|v_0\|0$  according to the definition of byte. When  $v_7 = 1$ , the irreducible polynomial  $P(x)$  is subtracted from the result. Subtraction is the inverse of the addition. Because the addition is the XOR, the subtraction is also a simple application of the XOR operations. Hence, the result is

$$(v_6x^7 + v_5x^6 + v_4x^5 + v_3x^4 + v_2x^3 + v_1x^2 + v_0x) \oplus (x^4 + x^3 + x + 1) \quad (1.31)$$

$$= v_6x^7 + v_5x^6 + v_4x^5 + (v_3 \oplus 1)x^4 + (v_2 \oplus 1)x^3 + v_1x^2 + (v_0 \oplus 1)x + 1. \quad (1.32)$$

According to the definition of byte, the result is  $v_6\|v_5\|v_4\|\bar{v}_3\|\bar{v}_2\|v_1\|\bar{v}_0\|1$ .

#### 1.5.1.2 S-box

AES uses a substitution-box (**S-box**) to mix the data. The S-box is used in both of the round function and the KSF, and thus is defined here. The S-box used in AES is a pre-determined bijective mapping from an 8-bit value to an 8-bit value. The definition of the AES S-box is shown in Table 1.6. Hereafter, the S-box transformation is described as  $S(\cdot)$ . For example,  $S(4e)$  returns  $2f$ , and  $S(d5)$  returns  $03$ .

Note that the S-box and the inverse S-box transformations are not identical. As explained later, AES decryption algorithm requires the look-up table for the inverse of  $S(\cdot)$ , that is  $S^{-1}(\cdot)$ .

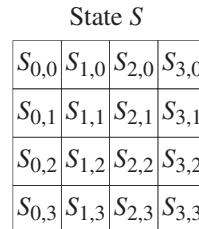
#### 1.5.1.3 State

The block size of AES is 128 bits. In AES, 128-bit data is called **state**. The 128-bit state consists of 16 bytes, and is represented as a  $4 \times 4$  two-dimensional array as depicted in Figure 1.7.

**Table 1.6** AES S-box

		Lower four digits															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
Upper four digits	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

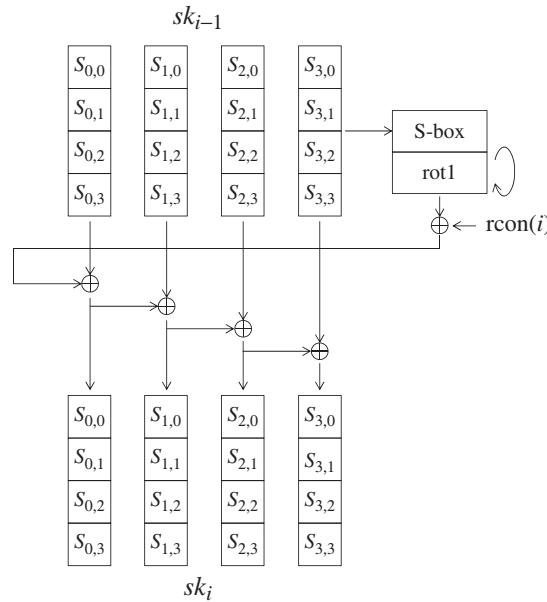
\* All the numbers in this table are in hexadecimal.

**Figure 1.7** AES state. Each cell denotes a byte

#### 1.5.1.4 Key Schedule Function (KSF)

The 128-bit key  $K$  is loaded into a 128-bit subkey  $sk_0$ . Then,  $sk_i \leftarrow \text{KSF}(sk_{i-1})$  is computed for  $i = 1, 2, \dots, 10$ . The input  $sk_{i-1}$  is represented as a state. The state is further divided into four columns:  $sk_{i-1}(\text{Col}(0))$ ,  $sk_{i-1}(\text{Col}(1))$ ,  $sk_{i-1}(\text{Col}(2))$ , and  $sk_{i-1}(\text{Col}(3))$ . The output  $sk_i$  is computed column by column. At first, a temporary 4-byte value  $\text{tmp}$  is generated by using the value of  $sk_{i-1}(\text{Col}(3))$ .

1.  $\text{tmp} \leftarrow sk_{i-1}(\text{Col}(3))$ .
2. Apply the S-box defined in Table 1.6 to each of the 4 bytes in  $\text{tmp}$ .
3. Rotate  $\text{tmp}$  by 1 byte. Precisely, let  $\text{tmp}_0 \parallel \text{tmp}_1 \parallel \text{tmp}_2 \parallel \text{tmp}_3$  be the 4 bytes of  $\text{tmp}$ . Then,  $\text{tmp}$  is updated to  $\text{tmp}_1 \parallel \text{tmp}_2 \parallel \text{tmp}_3 \parallel \text{tmp}_0$ .
4. XOR the pre-specified 1-byte constant  $rcon(i)$  to the first byte of  $\text{tmp}$ .



**Figure 1.8** Key schedule function of AES-128. The key schedule function is iterated for  $i = 1, 2, \dots, 10$

Then, by using the 4-byte value  $\text{tmp}$ , the next subkey  $sk_i$  is generated as follows.

1.  $sk_i(\text{Col}(0)) \leftarrow \text{tmp} \oplus sk_{i-1}(\text{Col}(0)).$
2.  $sk_i(\text{Col}(1)) \leftarrow sk_i(\text{Col}(0)) \oplus sk_{i-1}(\text{Col}(1)).$
3.  $sk_i(\text{Col}(2)) \leftarrow sk_i(\text{Col}(1)) \oplus sk_{i-1}(\text{Col}(2)).$
4.  $sk_i(\text{Col}(3)) \leftarrow sk_i(\text{Col}(2)) \oplus sk_{i-1}(\text{Col}(3)).$

The key schedule procedure for AES-128 is depicted in Figure 1.8.

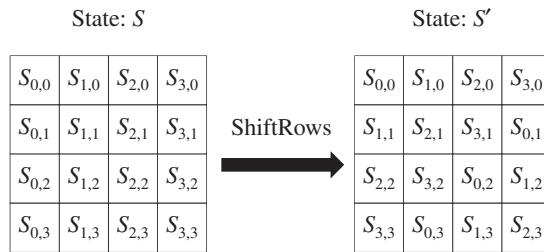
**Exercise 1.6** Write the algorithm of the key schedule function for AES-128. The similar style as Algorithm 1.1 can be used.

### 1.5.1.5 Round Function (RF)

The round function takes as input the previous 128-bit state  $state_i$  and subkey  $sk_i$ , and generates the next 128-bit state  $state_{i+1}$ . The round function consists of four transformations called **SubBytes**, **ShiftRows**, **MixColumns**, and **AddRoundKey**. It updates the state by following Algorithm 1.2.

**Algorithm 1.2 AES Round Function****Input:** Previous State  $S_i$ , subkey  $sk_i$ **Output:** New State  $S_{i+1}$ 

- 1: Set temporary state  $S_{\text{tmp}} \leftarrow S_i$ ;
- 2:  $S_{\text{tmp}} \leftarrow \text{SubBytes}(S_{\text{tmp}})$ ;
- 3:  $S_{\text{tmp}} \leftarrow \text{ShiftRows}(S_{\text{tmp}})$ ;
- 4:  $S_{\text{tmp}} \leftarrow \text{MixColumns}(S_{\text{tmp}})$ ;
- 5:  $S_{\text{tmp}} \leftarrow S_{\text{tmp}} \oplus sk_i$ ;
- 6:  $S_{i+1} \leftarrow S_{\text{tmp}}$ ;
- 7: **return**  $S_{i+1}$ ;

**Figure 1.9** ShiftRows operation**SubBytes (SB)**

SubBytes is a byte-wise operation. It updates the state by applying the S-box defined in Table 1.6 to each of the 16 bytes of the state. It is worth noting that the SubBytes operation is the only nonlinear one in the AES round function.

**ShiftRows (SR)**

ShiftRows is a row-wise byte positions swap. The state consists of four rows: row 0, row 1, row 2, and row 3. Each row of the state consists of 4 bytes. The ShiftRows operation applies a left cyclic shift by  $i$  bytes to the 4 bytes of row  $i$ . The ShiftRows operation is depicted in Figure 1.9.

**MixColumns (MC)**

MixColumns is a column-wise data mixing operation. It takes as input 4 bytes in a column and computes another 4-byte value. The same computation is applied to all of the four columns. Let  $x_0, x_1, x_2, x_3$  and  $y_0, y_1, y_2, y_3$  be the 4-byte input and 4-byte output, respectively. The  $y_0, y_1, y_2, y_3$  is computed by the following matrix operation:

$$\begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix}. \quad (1.33)$$

Each element in the matrix is written in hexadecimal.

The MixColumns operation was designed to satisfy the MDS property explained in Section 1.4.2.1. The impact of modifying 1 input byte always expands to all the 4 output bytes. More generally, the sum of the number of modified input bytes and the number of modified output bytes is always greater than or equal to 5.

### AddRoundKey (AK)

AddRoundKey updates the state by XORing the subkey  $sk_i$  to the state.

### Last Round Function (RF<sub>last</sub>)

In the last round (Round 10 for AES-128), the round function is different from the middle rounds. The MixColumns operation is not computed that is, only the SubBytes, ShiftRows, and AddRoundKey operations are performed.

**Exercise 1.7** Let us consider exchanging the order of two operations in the round function. Which of the following choices return the same result as the original AES specification even if the operations order is exchanged? Why do they return the same result?

1. SubBytes and ShiftRows
2. ShiftRows and MixColumns
3. MixColumns and AddRoundKey

## 1.5.2 AES-128 Decryption

To decrypt ciphertext  $C$  to  $P$ , the round function is applied in reverse order. The KSF is exactly the same. Eleven subkeys  $sk_0, sk_1, \dots, sk_{10}$  are generated from  $K$ . Different from the encryption algorithm,  $sk_{10}$  is firstly used, and then the decryption is processed with  $sk_9, sk_8, \dots$ , and finally with  $sk_0$ .

Inside the round function, four operations are computed in reverse order. The inverse of the AddRoundKey operation is exactly the same as the original AddRoundKey operation because the XOR operation is involution.

For the inverse of the MixColumns operation, the inversion matrix is required. Let  $b_0, b_1, b_2, b_3$  and  $a_0, a_1, a_2, a_3$  be the 4-byte input and 4-byte output to the inverse MixColumns operation, respectively. The  $a_0, a_1, a_2, a_3$  is computed by the following matrix operation:

$$\begin{bmatrix} e & b & d & 9 \\ 9 & e & b & d \\ d & 9 & e & b \\ b & d & 9 & e \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}. \quad (1.34)$$

Each element in the matrix is again written in hexadecimal.

**Table 1.7** AES inverse S-box

		Lower four digits															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
Upper four digits	0	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
	1	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
	2	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
	3	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
	4	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
	5	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
	6	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
	7	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
	8	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
	9	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
	a	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
	b	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
	c	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
	d	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
	e	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
	f	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

All the numbers in this table are in hexadecimal.

The inverse of the ShiftRows operation is relatively simple. It applies a right cyclic shift by  $i$  bytes to the 4 bytes of row  $i$ .

The inverse of the SubBytes operation requires another table to substitute each byte value. The inverse S-box, denoted by  $S(\cdot)^{-1}$ , is defined in Table 1.7.

**Exercise 1.8** Write the AES-128 decryption algorithm. The similar style as Algorithm 1.1 can be used.

### 1.5.3 Specification of AES-192 and AES-256

AES supports not only the 128-bit key but also the 192-bit and the 256-bit keys. For all the key sizes, round function is identical. The differences are the number of rounds computed and the KSF.

- AES-128 generates eleven 128-bit subkeys  $sk_0, sk_1, \dots, sk_{10}$  from 128-bit  $K$ , and the number of rounds is 10.
- AES-192 generates thirteen 128-bit subkeys  $sk_0, sk_1, \dots, sk_{12}$  from 192-bit  $K$ , and the number of rounds is 12.
- AES-256 generates fifteen 128-bit subkeys  $sk_0, sk_1, \dots, sk_{14}$  from 256-bit  $K$ , and the number of rounds is 14.

### 1.5.3.1 The Key Schedule Function for AES-192

The 192-bit key  $K$  is loaded into a  $4 \times 6$  array of bytes, which is denoted by  $Kstate_0$ . Then,  $Kstate_i \leftarrow KSF(Kstate_{i-1})$  is computed for  $i = 1, 2, \dots, 8$ . The state is further divided into six columns:  $Kstate_{i-1}(Col(0))$ ,  $Kstate_{i-1}(Col(1))$ ,  $Kstate_{i-1}(Col(2))$ ,  $Kstate_{i-1}(Col(3))$ ,  $Kstate_{i-1}(Col(4))$ , and  $Kstate_{i-1}(Col(5))$ . The output  $Kstate_i$  is computed column by column. At first, a temporary 4-byte value  $\text{tmp}$  is generated by using the value of  $Kstate_{i-1}(Col(5))$ .

1.  $\text{tmp} \leftarrow Kstate_{i-1}(Col(5))$ .
2. Apply the S-box defined in Table 1.6 to each of the 4 bytes in  $\text{tmp}$ .
3. Rotate  $\text{tmp}$  by 1 byte. Precisely, let  $\text{tmp}_0 \parallel \text{tmp}_1 \parallel \text{tmp}_2 \parallel \text{tmp}_3$  be the 4 bytes of  $\text{tmp}$ . Then,  $\text{tmp}$  is updated to  $\text{tmp}_1 \parallel \text{tmp}_2 \parallel \text{tmp}_3 \parallel \text{tmp}_0$ .
4. XOR the pre-specified 1-byte constant  $rcon(i)$  to the first byte of  $\text{tmp}$ .

Then, by using the 4-byte value  $\text{tmp}$ , the next subkey  $Kstate_i$  is generated as follows.

1.  $Kstate_i(Col(0)) \leftarrow \text{tmp} \oplus Kstate_{i-1}(Col(0))$ .
2.  $Kstate_i(Col(1)) \leftarrow Kstate_i(Col(0)) \oplus Kstate_{i-1}(Col(1))$ .
3.  $Kstate_i(Col(2)) \leftarrow Kstate_i(Col(1)) \oplus Kstate_{i-1}(Col(2))$ .
4.  $Kstate_i(Col(3)) \leftarrow Kstate_i(Col(2)) \oplus Kstate_{i-1}(Col(3))$ .
5.  $Kstate_i(Col(4)) \leftarrow Kstate_i(Col(3)) \oplus Kstate_{i-1}(Col(4))$ .
6.  $Kstate_i(Col(5)) \leftarrow Kstate_i(Col(4)) \oplus Kstate_{i-1}(Col(5))$ .

Among the 192-bit of the  $Kstate_0$ , the first four columns (128 bits) are set to  $sk_0$ , and the remaining two columns (64 bits) are set to the left half of  $sk_1$ . Among the 192-bit of the  $Kstate_1$ , the first two columns (64 bits) are set to the right half of  $sk_1$ , and the remaining four columns (128 bits) are set to  $sk_2$ . Similarly,  $sk_3, sk_4, \dots, sk_{12}$  are obtained.

Note that  $sk_{11}$  is the last four columns of  $Kstate_7$ , and then  $sk_{12}$  is the first four columns of  $Kstate_8$ . The last two columns of  $Kstate_8$  are never used. Thus, in order to omit the redundant computations, the KSF should be processed up to the first four columns of  $Kstate_8$ .

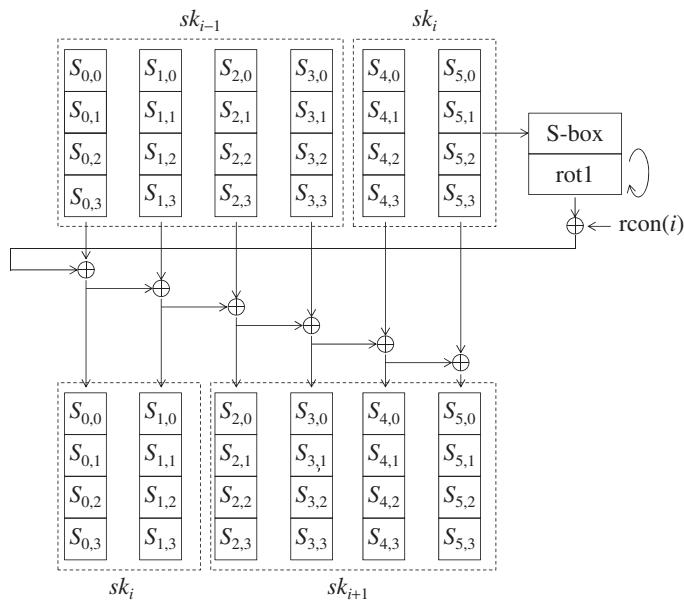
The key schedule procedure for AES-192 is depicted in Figure 1.10.

### 1.5.3.2 The Key Schedule Function for AES-256

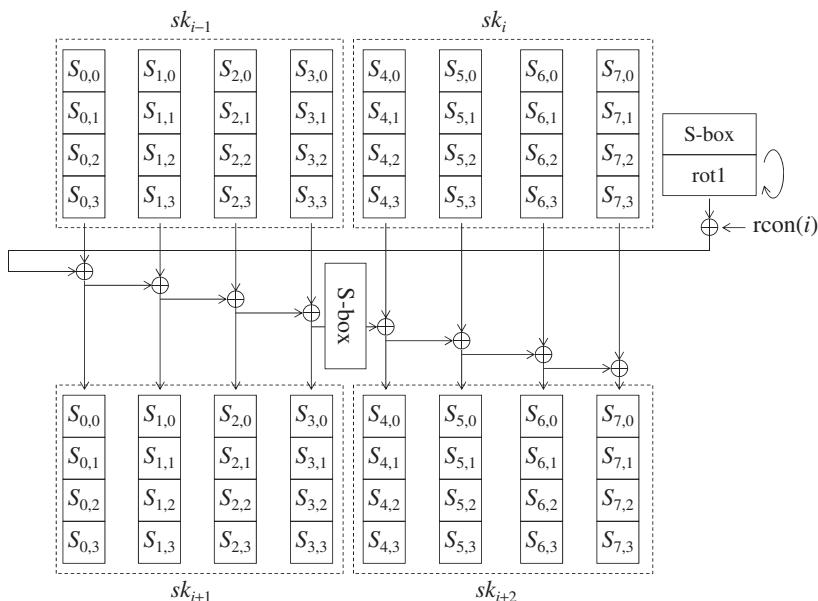
The KSF for AES-256 can be similarly defined. The size of the key state is 256 bits consisting of  $4 \times 8$ -byte array. Each key state produces two subkeys, and 15 subkeys  $sk_0, sk_1, \dots, sk_{14}$  are generated.

The update computation is very similar to the ones for AES-128 and AES-192. In order to mix the data quickly, another S-box layer is introduced between columns 3 and 4. The detailed procedure is omitted. The key schedule procedure for AES-256 is depicted in Figure 1.11.

Note that  $sk_{14}$  is the first four columns of  $Kstate_7$ . The last four columns of  $Kstate_7$  are never used. Thus, in order to omit the redundant computations, the KSF should be processed up to the first four columns of  $Kstate_7$ .



**Figure 1.10** Key schedule function of AES-192. The key schedule function is iterated until 13 subkeys are generated



**Figure 1.11** Key schedule function of AES-256. The key schedule function is iterated until 15 subkeys are generated

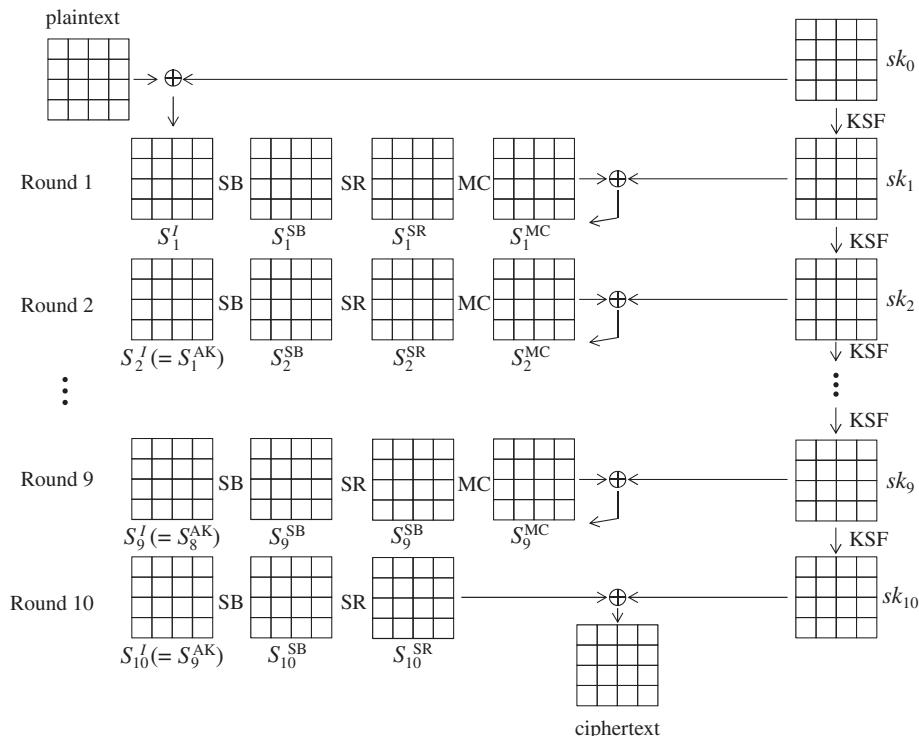
**Exercise 1.9** Compare the number of KSF calls per 128-bit subkeys for AES-128, AES-192, and AES-256 (weak mixing effect of the AES-256 KSF).

**Exercise 1.10** Compare the number of S-box calls per 128-bit subkeys for AES-128, AES-192, and AES-256 (weak nonlinearity of the AES-192 KSF).

### 1.5.4 Notations to Describe AES-128

The computation of AES-128 with all the operations is described in Figure 1.12. The state after the first XOR of the plaintext and  $sk_0$  is denoted by  $S_1^I$ . Similarly in round  $i$ , where  $i \in \{1, 2, \dots, 10\}$ ,

- the state at the beginning of the round is denoted by  $S_i^I$ ;
- the state after the SubBytes operation is denoted by  $S_i^{SB}$ ;



**Figure 1.12** Notations for each state of AES-128

	Inverse diagonal		Diagonal
0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

Figure 1.13 Notations for inside AES state

- the state after the ShiftRows operation is denoted by  $S_i^{\text{SR}}$ ;
- the state after the MixColumns operation is denoted by  $S_i^{\text{MC}}$ ;
- the state after the AddRoundKey operation is denoted by  $S_i^{\text{AK}}$ , which is equivalent to  $S_{i+1}^I$ .

As explained before, the state is represented by a  $4 \times 4$ -byte array. Using two subindices often causes misunderstanding, and thus each byte position is also denoted by a single sequence  $\{0, 1, \dots, 15\}$ . For state  $S$ , the byte  $S_{u,v}$ , where  $0 \leq u, v \leq 3$  is converted to the byte  $S[4 * u + v]$ . The byte positions in the single sequence are shown in Figure 1.13.

Byte values of state  $S$  in several different byte positions  $[a], [b], [c], \dots$  are often denoted by  $S[a, b, c, \dots]$ . For example, the 4-byte value in the column 0 of state  $S$  is denoted by  $S[0, 1, 2, 3]$ .

- The first column, or column 0, of state  $S$  is denoted by  $S[\text{Col}(0)]$ , which is equivalent to  $S[0, 1, 2, 3]$ .
- The second column, or column 1, of state  $S$  is denoted by  $S[\text{Col}(1)]$ , which is equivalent to  $S[4, 5, 6, 7]$ .
- The third column, or column 2, of state  $S$  is denoted by  $S[\text{Col}(2)]$ , which is equivalent to  $S[8, 9, 10, 11]$ .
- The fourth column, or column 3, of state  $S$  is denoted by  $S[\text{Col}(3)]$ , which is equivalent to  $S[12, 13, 14, 15]$ .
- The first row, or row 0, of state  $S$  is denoted by  $S[\text{Row}(0)]$ , which is equivalent to  $S[0, 4, 8, 12]$ .
- The second row, or row 1, of state  $S$  is denoted by  $S[\text{Row}(1)]$ , which is equivalent to  $S[1, 5, 9, 13]$ .
- The third row, or row 2, of state  $S$  is denoted by  $S[\text{Row}(2)]$ , which is equivalent to  $S[2, 6, 10, 14]$ .
- The fourth row, or row 3, of state  $S$  is denoted by  $S[\text{Row}(3)]$ , which is equivalent to  $S[3, 7, 11, 15]$ .

State  $S_i^{\text{SB}}$  becomes state  $S_i^{\text{SR}}$  after the ShiftRows operation. During this process, 4 bytes in  $S_i^{\text{SB}}[\text{Col}(j)]$  are moved to different byte positions in  $S_i^{\text{SR}}$ . The moved positions are denoted by  $\text{SR}(\text{Col}(j))$ .

- For state  $S$ , 4 bytes of  $S[\text{SR}(\text{Col}(0))]$  are equivalent to  $S[0, 7, 10, 13]$ .
- For state  $S$ , 4 bytes of  $S[\text{SR}(\text{Col}(1))]$  are equivalent to  $S[1, 4, 11, 14]$ .
- For state  $S$ , 4 bytes of  $S[\text{SR}(\text{Col}(2))]$  are equivalent to  $S[2, 5, 8, 15]$ .
- For state  $S$ , 4 bytes of  $S[\text{SR}(\text{Col}(3))]$  are equivalent to  $S[3, 6, 9, 12]$ .

Those 4-byte positions are called **diagonal**.

Similarly 4 bytes in  $S_i^{\text{SR}}[\text{Col}(j)]$  are moved to different byte positions in  $S_i^{\text{SB}}$  through the inverse of the ShiftRows operation. The moved positions are denoted by  $\text{SR}^{-1}(\text{Col}(j))$ .

- For state  $S$ , 4 bytes of  $S[\text{SR}^{-1}(\text{Col}(0))]$  are equivalent to  $S[0, 5, 10, 15]$ .
- For state  $S$ , 4 bytes of  $S[\text{SR}^{-1}(\text{Col}(1))]$  are equivalent to  $S[3, 4, 9, 14]$ .
- For state  $S$ , 4 bytes of  $S[\text{SR}^{-1}(\text{Col}(2))]$  are equivalent to  $S[2, 7, 8, 13]$ .
- For state  $S$ , 4 bytes of  $S[\text{SR}^{-1}(\text{Col}(3))]$  are equivalent to  $S[1, 6, 11, 12]$ .

Those 4-byte positions are called **inverse diagonal**.

## Further Reading

Daemen J and Rijmen V June 1998 *AES submission document on Rijndael*.

Daemen J and Rijmen V 2002 *The Design of Rijndael: AES—The Advanced Encryption Standard (AES)*. Springer-Verlag.

Deschamps JP 2009 *Hardware Implementation of Finite-Field Arithmetic* 1st edn. McGraw-Hill, Inc., New York, NY.

Paar C and Pelzl J 2010 *Understanding Cryptography: A Textbook for Students and Practitioners*. Springer-Verlag, New York.



# 2

# Introduction to Digital Circuits

## 2.1 Basics of Modern Digital Circuits

Since the invention of **complementary metal-oxide-semiconductor (CMOS)** technology, digital logic circuits have contributed to processing massive digital data. Examples of the applications based on the digital processing are laptop computers, smart phones, wearable devices, smart cards, and so on. In this chapter, we learn the basics of how to implement a digital hardware. Especially, **synchronous design**, which is commonly applied to most of the digital circuits, is introduced as a key technology.

The synchronous design uses a **clock signal** that is distributed to the whole synchronous circuits, and takes the timing of operations. To look into the details of the synchronization mechanism, the building blocks in the synchronous circuit are described. As they play an important role in the digital circuits, memory modules are also discussed.

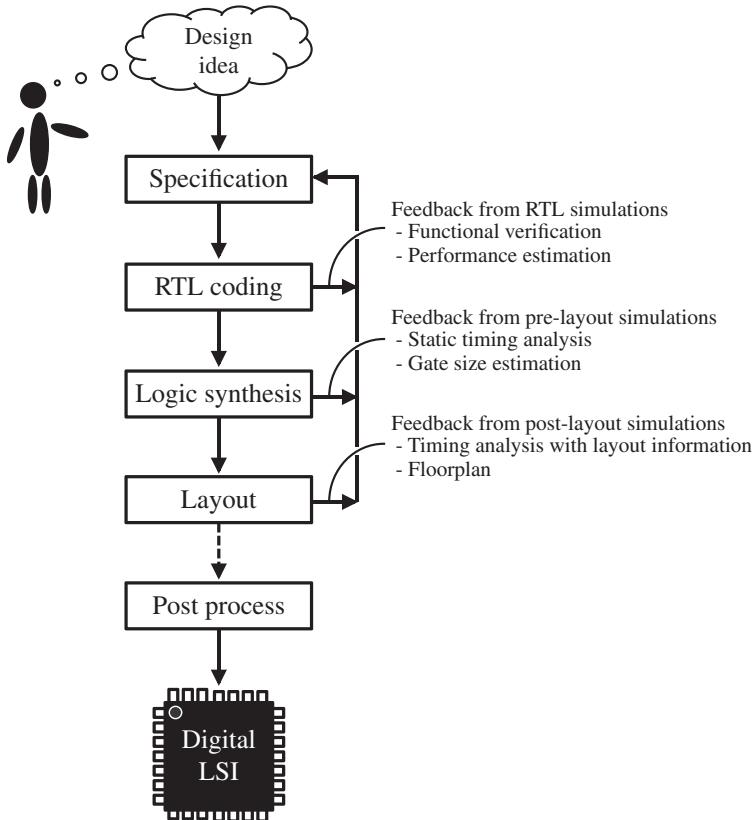
### 2.1.1 Digital Circuit Design Method

Many modern digital circuits are implemented based on the synchronous design method that can handle integrated circuits (ICs) by using commercial **design automation (DA)** tools. As illustrated in Figure 2.1, the synchronous design starts from **register transfer level (RTL)** coding after defining a design specification. At this design stage, the designers can know the speed performance and the design validity by RTL simulators. Different from a normal program code of C language, an RTL code has information of timing that determines when each variable is changed or how long it should keep the same value.

The RTL simulation speed is fast enough to perform a lot of functional tests of the code. Therefore, the debugging and performance test of the design should be done at this stage, and the simulation results are fed back to the design specification, if necessary. Note that design synthesis to a gate-level description is necessary to estimate the area cost.

### 2.1.2 Synchronous-Style Design Flow

Synchronous-style design method is one of the design methods that have led the evolution of digital circuits since around 1990s. Compared to the **asynchronous-style design flow**, where



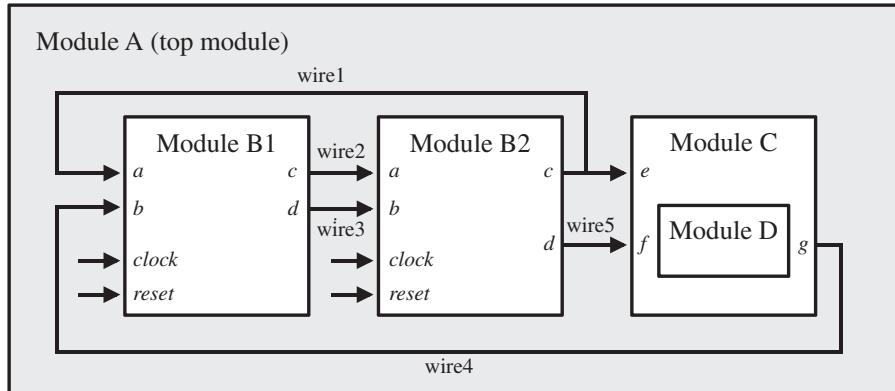
**Figure 2.1** An overview of synchronous-style design flow

the data processing timing is not triggered by the clock edge, the synchronous-style one has advantage in that there is a design flow where many design steps are automated by DA tools. An overview of the design flow is shown in Figure 2.1.

1. **Specification:** Designers define the specification of digital circuits. The specification describes the interface and its characteristics, necessary functionality, speed performance, and so on.
2. **RTL coding:** Designers can use a high-abstraction-level language as an entry of implementation of the digital circuit such as **Verilog HDL** (hardware description language) or **Verilog** for short, which directly contributes to a high productivity in the design flow.<sup>1</sup>
3. **Logic synthesis:** An RTL description programmed with Verilog is automatically synthesized to gate-level description that is called **netlist**.
4. **Layout:** **Logical gates** and **wires** in the synthesized design are allocated physically by using a layout tool automatically.

Cryptographic hardware, which is the target device in this book, can be seen as one of the digital circuits, and hence the synchronous-style design flow or similar one is often used.

<sup>1</sup> Even higher abstraction-level languages, for example, SystemC and System Verilog, have been proposed.



**Figure 2.2** Hierarchical structure in digital circuit design

### 2.1.3 Hierarchy in Digital Circuit Design

Digital circuits have many functional blocks such as **controller**, **datapath** including **arithmetic logic unit**, **memory**, and so on. Therefore, hierarchical structure is introduced in designing the digital circuit in order to divide the functionality and to ease the design verification.

Figure 2.2 illustrates the hierarchical structure in a modern digital circuit. Each node in the hierarchical directory tree is called a **module** that is a unit of functional block. The top module usually contains several submodules, and a multiple nest structure is constructed. An example of the structure of modules is shown in Figure 2.2.

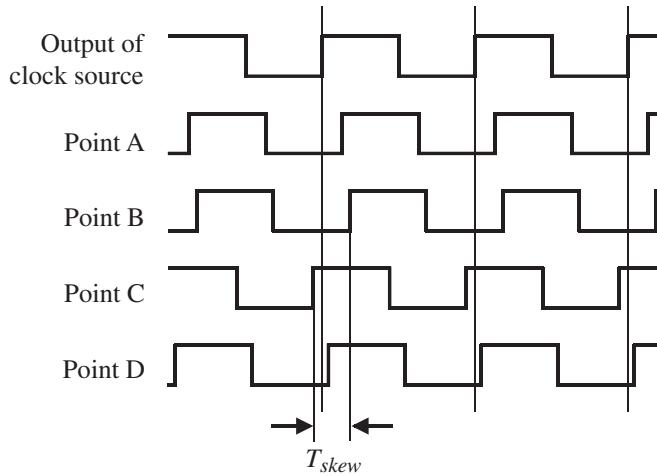
In this case, the top module has three submodules, one of which, module C, has a submodule, module D. Modules B1 and B2 have the same functionality when they are instantiated from the same module. Each module has input, output, and inout ports that are connected to ports of other modules. Modules B1 and B2 contain sequential logics that will be explained in Section 2.3.2, and hence the clock and reset signals are provided as input.

## 2.2 Classification of Signals in Digital Circuits

Before discussing the details of designing digital hardware, classification of signals, propagating through wires and modules, is explained here. There are three main types of signals; **clock**, **reset**, and **data signals**. Note that clock and reset signals are not described in an algorithm-level description, where only data flow is used. On the other hand, clock and reset signals play an important role in a synchronous-style hardware design.

### 2.2.1 Clock Signal

The clock signal requires a special consideration concerning the routing of its signal line so that all circuits can start operation at the same timing. More precisely, it is necessary that all the sequential logic gates, which will be explained later, using the clock signal have to detect the **positive edge** (or **negative edge**) of the clock at the same or quite similar timings.



**Figure 2.3** Image of clock skew

However, even if the wiring delay of the clock signal is routed carefully, there exists a slight difference in the arrival time of the sequential logics. Such timing difference is called **clock skew**, and its maximum difference is denoted as  $T_{skew}$  as illustrated in Figure 2.3. Moreover, there exists another clock timing difference called **clock jitter**, which also effects the timing difference. It is a distortion of the signal caused when it is generated in a clock generator and is distributed to the circuit. In this chapter, it is assumed that the clock jitter is appropriately controlled for simplicity.

The simplest way to realize the synchronous-style circuit is to use a single clock with a clock frequency,  $f_{clk}$ . The clock period,  $T_{clk}$ , is the inversion of the clock frequency. That is,

$$T_{clk} = 1/f_{clk}. \quad (2.1)$$

Although the timing management becomes complex, it is also possible to employ multiple clocks with different clock frequencies.

### 2.2.2 Reset Signal

**Reset signal** is also a special signal used for initializing the value of logics. When powering on a synchronous hardware, the initial state of the sequential logics (details in Section 2.3.2) is not always the expected initial values. In order to make the same initial state after powering on, a power-on reset is normally performed. In addition, the reset can interrupt an operation while it is running, and can initialize the state at any time.

In a synchronous design, synchronous or asynchronous reset can be used. The synchronous reset initializes the sequential logics when the active reset signal is detected when the positive edge of the clock signal arrives. On the other hand, the asynchronous reset can initialize the sequential logics regardless of the clock timing. Therefore, it can reset the sequential logics even when the clock signal is not provided, for example, immediately after powering on the circuit.

Both the resets have merits and demerits depending on the available library of sequential logics and design rule for the reset signal. The asynchronous reset is used for the sequential

logics in this chapter, considering that further discussion can be continued without loss of generality. However, the deactivate timing of the reset should be handled with care in relation to the clock signal timing as the release timing of the reset needs to be the same for all the sequential logics. Namely, the arrival timings of the asynchronous reset signals vary depending on the sequential circuit position, and moreover there exist the clock skew and clock jitter. One possible way for an appropriate reset is adjusting the deactivation timing of the reset with the clock signals (i.e., synchronously releasing the reset), whereas the activation timing keeps asynchronous with the clock signals (i.e., asynchronously starting the reset).

**Exercise 2.1** As for the circuit shown in Figure 2.2, suppose that Modules B1 and B2 use a 100 MHz clock signal and an asynchronous reset signal as their inputs in common (i.e., Modules B1 and B2 use the same clock and the same reset signal), and there is no clock skew or clock jitter. If the arrival time of the reset signal to the module B1 is 1 ns earlier than that to Module B2, the modules may have a problem that they cannot perform a correct operation after the reset signal is deactivated. Explain the reason with considering the timing difference of the positive edge of the clock signal and the deactivation timing of the reset signals in Modules B1 and B2.

### 2.2.3 Data Signal

In the context of the synchronous-style design, data signals are regarded as signals that change at the positive edge of the clock. In simple terms, all the signals other than the clock and reset signals are considered as data signal. Except several special cases,<sup>2</sup> input and output of the sequential logics are connected to the data signals, and combinatorial logics, which will be explained later, are composed of data signals.

## 2.3 Basics of Digital Logics and Functional Modules

### 2.3.1 Combinatorial Logics

**Combinatorial logics** have input and output data signals, and the output signals are determined after evaluating the input signals. For instance, one-bit **full adder** (FA) operates the bit-wise operations as

$$s = a \oplus b \oplus c_{in}, \quad (2.2)$$

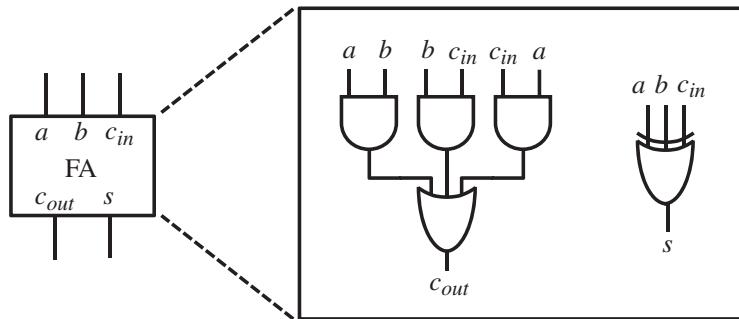
$$c_{out} = (a \wedge b) \vee (b \wedge c_{in}) \vee (c_{in} \wedge a), \quad (2.3)$$

where  $s$  is the bit-wise sum,  $a$  and  $b$  the operands, and  $c_{in}$  and  $c_{out}$  the carry-in and carry-out signals, respectively. The FA module and its corresponding combinatorial logics are illustrated in Figure 2.4.

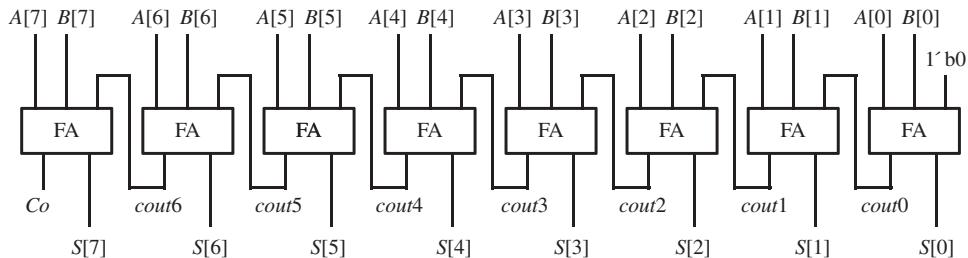
As shown in Figure 2.5, connecting eight FAs, 8-bit **ripple-carry adder** is implemented.<sup>3</sup> The pseudo Verilog code is described in Figure 2.6.

<sup>2</sup> They are clock divider, reset control circuits, and so on.

<sup>3</sup> The right-most FA can be replaced with half adder, where the carry-in signal,  $c_{in}$ , is omitted.



**Figure 2.4** One-bit full adder module and its corresponding combinatorial logics



**Figure 2.5** 8-bit ripple-carry adder based on FAs

Suppose that the input signals are changed from all zeros to all ones at the same time, that is, both  $A$  and  $B$  are changed from 0 to F. The **least significant bit** (LSB) of the addition result,  $S[0]$ , is calculated as 1 after the three-input XOR operation by Equation (2.2), and the carry-out signal,  $cout_0$  changes from 0 to 1 by Equation (2.3). Then,  $S[1]$  and  $cout_1$  are determined based on the signal change of  $cout_0$ . In this way, the addition results are determined bit by bit from LSB to **most significant bit** (MSB). That is, all the output signals are determined after the input signals are changed.

Note that the calculation time of the combinatorial logic, which can be measured as the difference between the input and the output determination times, varies from the input differences in value. It is called **path delay**. Intuitively, a smaller difference in the input signals tends to result in a less path delay.<sup>4</sup> However, this is not always true.

### 2.3.2 Sequential Logics

In contrast with combinatorial logics, the output of **sequential logics** is determined with its current value and the input signals. Figure 2.7 shows the schematics and of positive-edge-triggered **delay flip flop** or **DFF** for short with asynchronous reset. There are three input ports,  $CLK$ ,  $D$ ,

<sup>4</sup> If there is no difference in the input signals between positive edges of the clock signals, the path delay becomes zero.

```

module 8bit_adder (A, B, S, Co);

    input [7:0] A, B;
    output [7:0] S;
    output Co;

    wire cout0, cout1, ..., cout7;
    wire s0, s1, ..., s7;

    full_adder FA0(A[0], B[0], 1'b0, s0, cout0);
    full_adder FA1(A[1], B[1], cout0, s1, cout1);

    :
    full_adder FA7(A[7], B[7], cout6, s7, cout7);

    assign S = {s7, s6, ..., s0};
    assign Co = cout7;

endmodule

module full_adder (a, b, cin, s, cout);

    input a, b, cin;
    output s, cout;

    assign s = a | b | cin;
    assign cout = (a & b) | (b & cin) | (cin & a)

endmodule

```

**Figure 2.6** Pseudo Verilog code for 8-bit ripple-carry adder

and  $RST$ , and one output port,  $Q$ . The output of DFF is initialized when a reset signal becomes low, that is,  $RST = 1$ . Note that the reset signal is active-low here, and therefore the reset is performed when  $RST = 1$ . It is assumed that the initialization values for DFFs are zeros.

When the reset signal is released, that is,  $RST = 0$ , DFF changes its state by taking the input data via  $D$  at the positive edge of  $CLK$ . Immediately after the state changes in DFF, the output signal that can be obtained via  $Q$  is changed. Except the timing at the positive edge of the clock, DFF keeps the previous states, that is, the value of the output does not change. The truth table for DFF is shown in Table 2.1.

One of the possible DFF implementations is also shown in Figure 2.7. The box labeled with TG is called **transfer gate** that determines the connection of input and output signals of TG depending on two control signals.

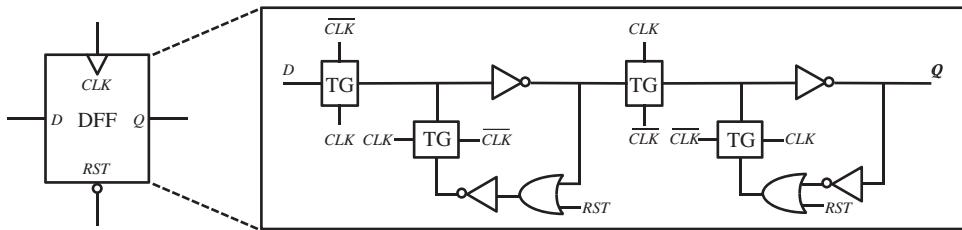


Figure 2.7 DFF with asynchronous reset

Table 2.1 Truth table of DFF with asynchronous reset

Reset (RST)	Clock (CLK)	Input (D)	Output (Q)	Comment
1	X	X	0	Initialize to zero
0	↑	0	0	Retrieve input data at the positive edge of CLK
0	↑	1	1	
0	0	X	Q	Keep previous state
0	1	X	Q	
0	↓	X	Q	

↑: positive edges of CLK signal, ↓: negative edges of CLK, "X": do not care.

When  $RST = 1$ ,  $Q$  becomes 0 regardless of the value of  $CLK$ , as shown in Figure 2.8. This indicates that the reset signal is effective asynchronously with the clock signal.

After the reset is released, that is,  $RST = 0$ , the state of DFF is determined by  $CLK$  and  $D$ . Figure 2.9 shows an example of the state transition when the clock signal changes from  $CLK = 0$  to  $CLK = 1$ . Suppose that  $Q = d1$  and  $D = d2$  when  $CLK = 0$ . The right-hand side of the DFF circuit has a ring circuit that is composed of only combinatorial logics. Therefore, the output of DFF is fixed as  $Q = d1$ . Note that after the reset,  $d1 = 0$ .

As the timing of the  $CLK$  becomes 1, that is, positive edge of the clock, TGs change the states, and the ring circuit is shifted to the left-hand side of the DFF to store the value of  $d2$ . Moreover, the value of  $Q$  changes from  $d1$  to  $d2$ . All the signals in the DFF circuits are  $d2$  or  $\bar{d}2$  after  $CLK$  becomes 1. In other words, the value of  $D$  is retrieved and stored in the DFF, and  $Q$  is replaced with  $D$ .

For example, sequential logics enable us to construct a counter that increments number synchronized with the clock signal. The incrementation timing is at the positive edge of the clock signal if positive-edge-triggered DFFs are used. An 8-bit up counter can be realized with the 8-bit adder, as introduced in Figure 2.5, and the 8-bit DFF.

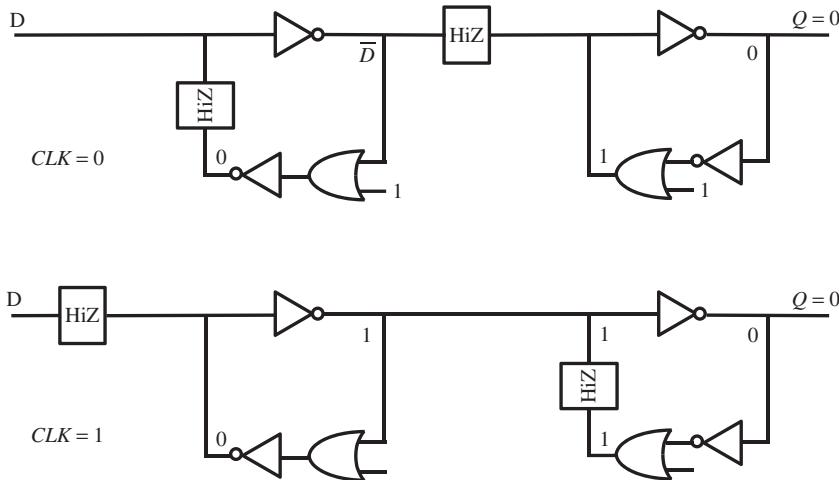
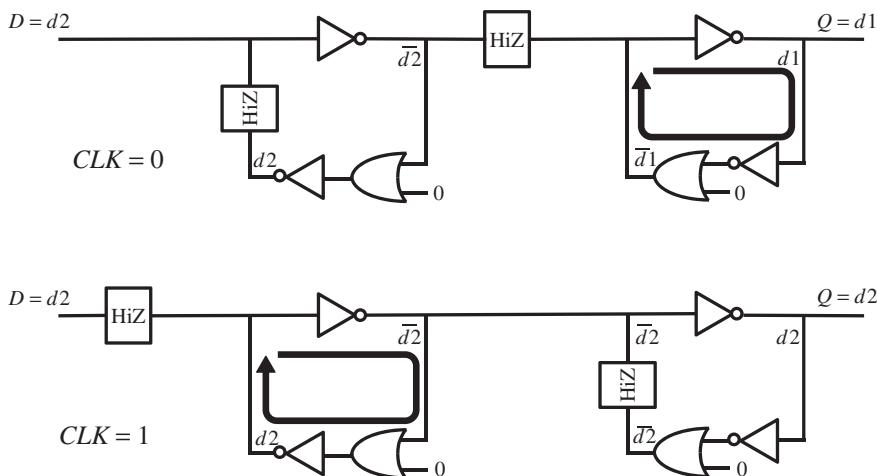
**Figure 2.8** State of DFF when  $RST = 1$  (reset is provided)**Figure 2.9** State change in DFF for a normal operation,  $RST = 0$ 

Figure 2.10 describes the pseudo Verilog code for the 8-bit up counter, and the corresponding timing waveform is shown in Figure 2.11. While the reset is activated, 8-bit DFF is initialized, and hence the output of the 8-bit adder,  $cnt\_next[7:0]$ , becomes 01 as the adder module increments the input value,  $cnt[7:0]$ , by one.

At the timing of the first positive edge of the clock signal,  $clk$ , after the reset signal,  $rst\_n$ , is deactivated, 8-bit DFF retrieves the value of  $cnt\_next[7:0]$ , and the output of DFF becomes 01. Likewise,  $cnt[7:0]$  counts up to ff using 255 clock cycles after the reset is released, and it becomes  $cnt[7:0] = 00$  in the next cycle.

```

module up_counter (clk, rst_n, cnt)

    input  clk, rst_n;
    output [7:0] cnt;

    reg [7:0] cnt;
    reg [7:0] cnt_next; // Output of combinatorial logics

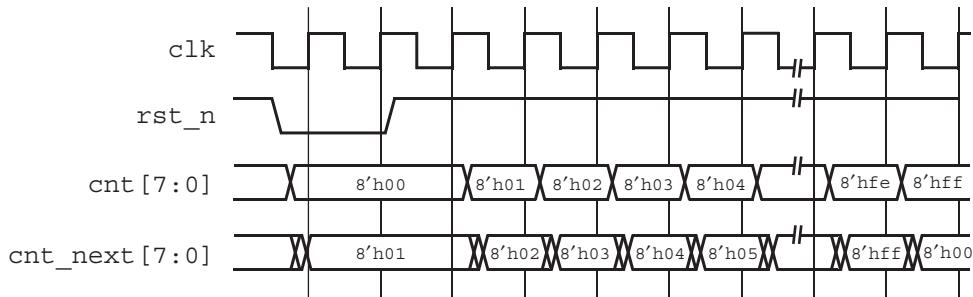
    8bit_adder 8add(cnt, 1'b1, cnt_next, Cout);

    always @(posedge clk or negedge rst_n) begin
        if (rst_n == 0)
            cnt <= 0;
        else
            cnt <= cnt_next;
    end

endmodule

```

**Figure 2.10** Pseudo Verilog code for 8-bit up counter



**Figure 2.11** Timing waveform for 8-bit up counter

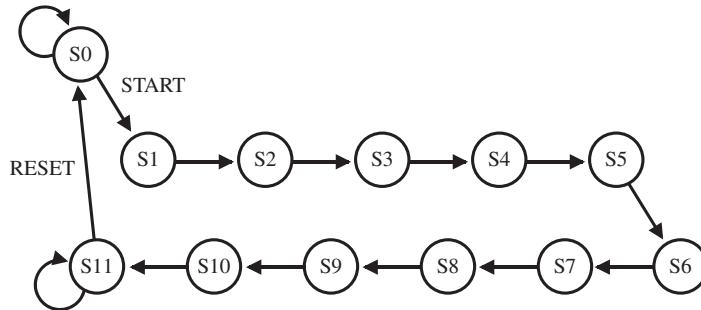
### 2.3.3 Controller and Datapath Modules

#### 2.3.3.1 Finite State Machine as Controller

A controller logic is regarded as **finite state machine (FSM)** whose next state is determined by the current state and several external conditions. FSM is often described with bubbles and arrows as illustrated in Figure 2.12. The bubble represents the state, and the arrow indicates the state transition. Figure 2.12 is a possible FSM representation for a controller design that controls an encryption of a 10-round block cipher.

Suppose that there are two external conditions, one condition, START, is for starting a cipher operation, that is, encryption, and another condition is RESET that is used for initializing the block cipher hardware. For simplicity, 1-round operation is iteratively performed in a single clock cycle under fixed input variables<sup>5</sup> after START. The operation is finished at the 10th round, and the ciphertext is available in the calculation result.

<sup>5</sup> More precisely, secret key and plaintext are fixed.



**Figure 2.12** State machine for an encryption hardware of a 10-round block cipher

**Table 2.2** State transitions of FSM shown in Figure 2.8

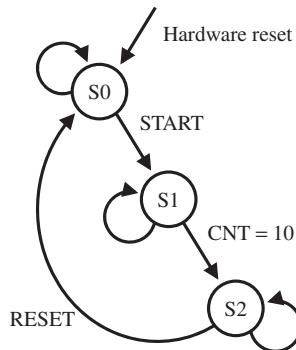
External condition	State	Control
Hardware reset	: S0	Initial state
START	S0 S1 S2 : S10	Initial state Perform 1st-round operation Perform 2nd-round operation : Perform 10th-round operation
	S11	Keep result
RESET	S11 S0	Keep result Initial state

As the cipher operation has to be stopped in 10 cycles, the state, S11, for keeping the result is prepared. If RESET is detected at the state, S11, that is, the corresponding reset signal becomes active, the state goes to the initial state, S0. Note that RESET is different from a hardware reset that is normally provided to reset the entire circuit including FSM and even other modules. Table 2.2 explains the details of FSM. In this case, there exist 12 states in total. The corresponding pseudo Verilog code is shown in Figure 2.13. As the states are realized with sequential logics, at least three DFFs are necessary. Moreover, the state transitions are straightforward from S1 to S11 in this case. That is, there is no legal way to stop the state transitions after START except providing the hardware reset.

As is often the case with block ciphers, the same operation is performed for each round except a small treatment of the first and last rounds. Suppose that all the rounds use an identical **round operation**, that is, round operation unit is called iteratively from FSM. In this case, the states, S1–S11, are redundant as they are the states of calling the same round operation. Having a counter unit that counts the number of round operations performed, FSM in Figure 2.14 can be realized with less bubbles.

```
module 10r_enc_fsm (clk, rst_n, START, RESET, state);  
  
    input  clk; rst_n;  
    input  START; // Start Encryption  
    input  RESET; // Reset FSM  
    output [3:0] state;  
  
    reg [3:0] state;  
    reg [3:0] next_state;  
  
    always @(posedge clk or negedge rst_n) begin  
        if (rst_n == 0)  
            state <= 0;  
        else  
            state <= state_next;  
    end  
  
    always @(state or START or RESET) begin  
        case (state)  
            4'b0000: begin // Initial State  
                if (START == 1'b1) next_state = 4'b0001;  
                else next_state = 4'b0000;  
            end  
            4'b0001: // 1st round operation  
                next_state = 4'b0010;  
            4'b0010: // 2nd round operation  
                next_state = 4'b0011;  
                :  
            4'b1010: // 10th round operation  
                next_state = 4'b1011;  
            4'b1011: begin // Keep Result  
                if (RESET == 1'b1) next_state = 4'b0000;  
                else next_state = 4'b1011;  
            end  
            default:  
                next_state = 4'b0000;  
        endcase  
    end  
  
endmodule
```

**Figure 2.13** Pseudo Verilog code for FSM for an encryption hardware of a 10-round block cipher



**Figure 2.14** Simplified state machine for an encryption hardware of a 10-round block cipher

**Table 2.3** State transitions for Figure 2.13

External condition	State	Control
Hardware reset	: S0	Initial state, CNT = 0
START	S0 S1 : S1 S2	Initial state Perform 1st-round operation, and increment CNT : 10th-round operation, and increment CNT Keep result
RESET	S2 S0	Keep result Initial state

The advantage of the less bubbles lies not only in the simplicity but in the possibility of choosing any number of rounds. The sequence listed in Table 2.3 is the state transitions for the simplified FSM.

The pseudo Verilog code for the combinatorial logics of `next_state` can be described as shown in Figure 2.15. The code is configurable with the number of rounds simply by changing the parameter `N_ROUND` that is defined in the first line of the code.<sup>6</sup>

### 2.3.3.2 Datapath

The **datapath** contains functional units that process data operations, for example, arithmetic addition, modular multiplication, permutation transformation, and so on. On the basis of the orders from FSM, the datapath starts operations with single or multiple inputs depending on the operation. The operation result appears on the output of the datapath after all the signals are evaluated and determined. Note that it takes a fixed amount of time until the operation result is available, which is called **path delay**.

<sup>6</sup> In the case that `N_ROUND` is 16 or more, the bit width of `next_state` needs to be increased.

```

`define N_ROUND 10
  always @(state or START or RESET or CNT) begin
    case (state)
      2'b00: begin // Initial State
        if (START == 1'b1) next_state = 2'b01;
        else next_state = 4'b0000;
      end
      2'b01: // 1st to 10th round operations
        if (CNT == N_ROUND) next_state = 2'b10;
        else next_state = 4'b0000;
      end
      2'b10: begin // Keep Result
        if (RESET == 1'b1) next_state = 2'b00;
        else next_state = 2'b10;
      end
      default:
        next_state = 2'b00;
    endcase
  end

```

**Figure 2.15** Pseudo Verilog code for encryption of a 10-round block cipher

The operation time should be short enough to guarantee correct operations, that is, the operation result needs to be determined for any input values within the **clock period**,  $T_{clk}$ , with an appropriate margin, which will be explained later in this chapter. However, some operations such as 64-bit modular multiplication spend more time compared to the case of 8-bit addition, and hence multiple clock cycles may be used until the result is available.

The path delay of the datapath usually depends on the value change of the inputs. If all the input values to be evaluated in the datapath are completely the same as the previously evaluated ones, all the signals in the datapath never change. In other words, the operation result is available with the operation time zero. Moreover, depending on the functionality of the datapath, there is a case that the input value difference does not effect the output value.<sup>7</sup> On the other hand, some change in the input value causes signal changes in the datapath, the so-called **signal toggles**, and it normally effects the output of the datapath.

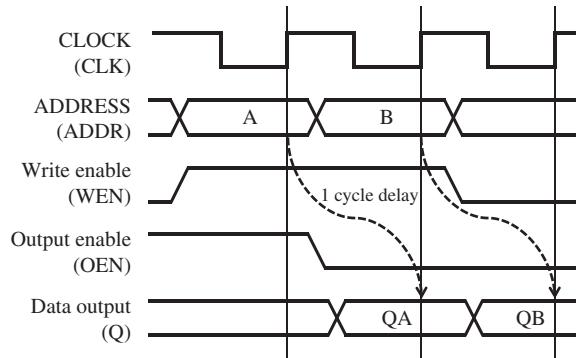
## 2.4 Memory Modules

### 2.4.1 Single-Port SRAM

A **static random access memory (RAM)** or **SRAM** is often used in the modern circuit design. It serves as a local memory that is dedicated to a specific module or as a global memory that can be accessed from multiple modules. SRAM is a significantly important module that allows to store temporary data with the high density as single bit can be stored with six transistors.<sup>8</sup>

<sup>7</sup> For different input values that generate the same output, signal toggles might occur in the datapath or even on the output of the datapath until the output result is determined.

<sup>8</sup> Higher density SRAM is proposed such as one-transistor SRAM.



**Figure 2.16** Example for read operation of single-port SRAM

A single-port SRAM has a common address port and it supports read or write operations together with a **write enable signal**. Owing to its specification, the single-port SRAM is not able to perform read and write operations at the same time. Figure 2.16 shows the timing waveform for a read operation of a typical single-port SRAM that is synchronized with the clock signal. The address data is fetched at the timing of positive edge of the clock signal and checks if the write enabled is high (read) or low (write). SRAM decodes the address data and sets the corresponding data on the data output signals that are controlled by the output enable signal. If the output signal is low, the SRAM data can be observed from the data output after a while, which causes a one-cycle delay as shown in the figure. If the address data is changed in the next cycle, for example, from A to B in the figure, the corresponding data, QB, is output subsequently to QA.<sup>9</sup>

For a write operation, written data and address are fetched at the same timing when the write-enabled signal is detected low at the positive edge of the clock. As shown in Figure 2.17, the write operations can be performed every single cycle.

As the read and write operations of single-port SRAM are performed sequentially, bandwidth becomes limited by the bus width of the data signal and the operating clock frequency. In order to overcome the limitation, one can consider the use of multiple ports.

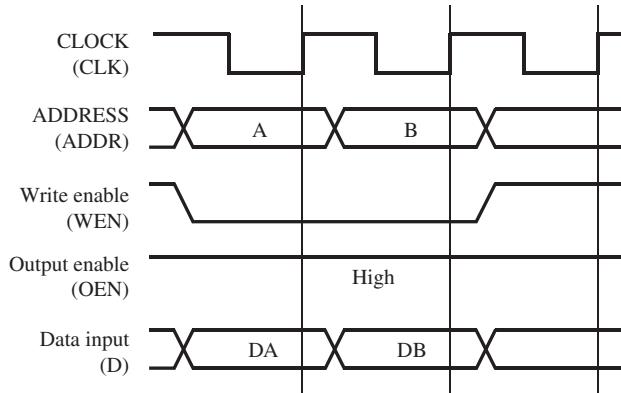
#### 2.4.2 Register File

A **register file** consists of several registers that are realized based on DFFs or SRAM, for instance. The register file is usually used in central processing unit (CPU) in executing instructions, and hence multiple read ports are required to accelerate a multioperand operation. For instance, the two-operand addition,  $c = a + b$ , can be performed in a single cycle when using a register file with two read ports and one write port. Such register file is called 2R-1W register file. Hardware modules can choose different types of memory modules depending on different purposes.

Compared to SRAM and DFF, the feature of register file is summarized in Table 2.4.

The bandwidth is decided by the bus width and the number of read/write ports in the case of register file, whereas it is proportional to the bus width only in the case of single-port SRAM.

<sup>9</sup> Some SRAMs support **burst access mode**, where data for a contiguous address space can be read out without setting the corresponding address one by one.



**Figure 2.17** Example for write operation of single-port SRAM

**Table 2.4** Features of SRAM, Register File, and DFF

	Single-port SRAM	Register file	DFF
Bandwidth	$\propto$ Bus width ( $n$ )	$\propto n \times$ Read/Write ports	Arbitrary
Shared level	Flexible	Module	Combinatorial logic
Memory cost*	Efficient	Medium	Inefficient
Memory size	Large	Medium ~ Small	Small

\*Depending on the size of memory and available library, register file or even DFF might become more cost efficient than SRAM.

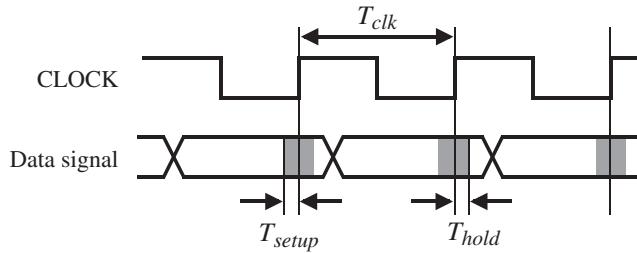
The shared level of the single-port SRAM is quite flexible, that is, it can be used in a single or multiple modules as it is cost efficient and is able to support a large size capacity. On the contrary, the register file is normally dedicated to a specific module owing to the limitation as for the cost and size.

## 2.5 Signal Delay and Timing Analysis

### 2.5.1 Signal Delay

#### 2.5.1.1 Setup Time and Hold Time

In a basic synchronous design, all signals are retrieved by DFFs at the positive edge of the clock. This is realized by the TGs, which determine the circuit connectivity as explained in Section 2.3.2. Therefore, the input signal  $D$  should not be changed at the positive edge of the clock for a stable operation of DFF.



**Figure 2.18** Setup time and hold time

As shown in Figure 2.18, **setup time** or  $T_{\text{setup}}$  is defined by the minimum time period that the data signal has to be determined before the positive edge of the clock signal. On the contrary, **hold time** or  $T_{\text{hold}}$  is the minimum time period that the data signal has to be held after the positive edge of the clock signal. The gray-colored parts in the data signal represent the forbidden time of the data change.

### 2.5.1.2 Critical Path Delay

A path delay,  $T_{pd}$ , is defined as the time spent for a signal propagation through a path between the input and the output of a combinatorial logic. The **critical path delay**, denoted as  $T_{cpd}$ , is a path delay that has the maximum signal transition time among all of the paths in the combinatorial logic. Suppose that a module is implemented with the synchronous-style design method. The following condition must be satisfied in order to satisfy the setup-time requirement.

$$T_{cpd} \leq T_{\text{clk}} - T_{\text{skew}} - T_{\text{setup}}. \quad (2.4)$$

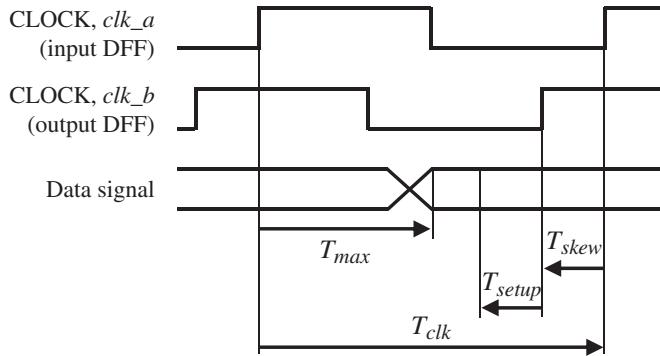
Suppose that data is provided to a combinatorial logic at the positive edge of `clk_a`, and the output of the combinatorial logic is retrieved by DFF whose clock signal is `clk_b`. The positive edge of `clk_b` arrives to DFF earlier than that of `clk_a` by  $T_{\text{skew}}$ , whereas the period is the same as `clk_a`. This is the worst condition for the setup-time requirement as for clock skew. In this case, the timing budget for the combinatorial logic is reduced from  $T_{\text{clk}}$  to  $T_{\text{clk}} - T_{\text{skew}}$ . Moreover, taking the setup margin into consideration, the critical path delay should be  $T_{\text{clk}} - T_{\text{skew}} - T_{\text{setup}}$  at most.

That is, the maximum clock frequency,  $f_{\max}$ , for the circuits illustrated on the right-hand side of Figure 2.19 is derived as

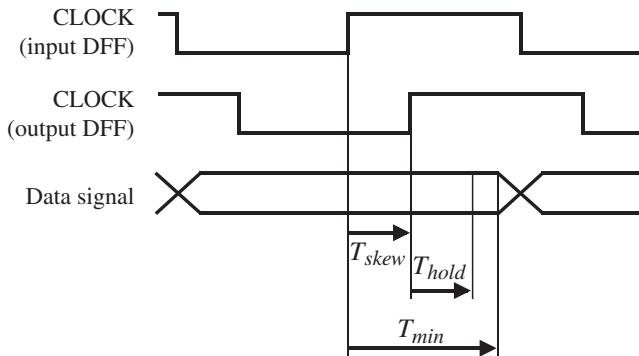
$$f_{\max} = \frac{1}{T_{\max} + T_{\text{skew}} + T_{\text{setup}}}. \quad (2.5)$$

As for the hold-time requirement as illustrated in Figure 2.20, the following condition must hold.

$$T_{\min} \geq T_{\text{hold}} + T_{\text{skew}}, \quad (2.6)$$



**Figure 2.19** Condition for satisfying setup time

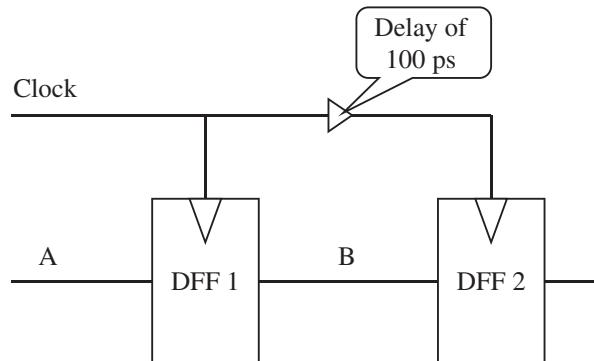


**Figure 2.20** Condition for satisfying hold time

where  $T_{\min}$  is the minimum path delay of the combinational logic. In this case, it is assumed that the phase  $\text{clk\_b}$  is delayed by  $T_{\text{skew}}$  to set the worst case for clock skew. Note that the condition does not include  $T_{\text{clk}}$ . This fact indicates that the hold-time requirement should be adjusted by the addition of delay elements, the so-called **hold buffer**, in the combinational circuit if  $T_{\min}$  is too small.

Both the setup-time and hold-time conditions introduced here assume the worst case, and hence it may not happen in a real design. In order to increase the maximum clock frequency and to reduce the cost of hold buffers, timing analysis should be performed with a precise timing model.

**Exercise 2.2** Discuss the impact on  $f_{\max}$  for different  $T_{\text{setup}}$  when  $T_{\text{cpd}}$  is 1 ns and  $T_{\text{skew}}$  is 100 ps.



**Figure 2.21** Example of hold buffer

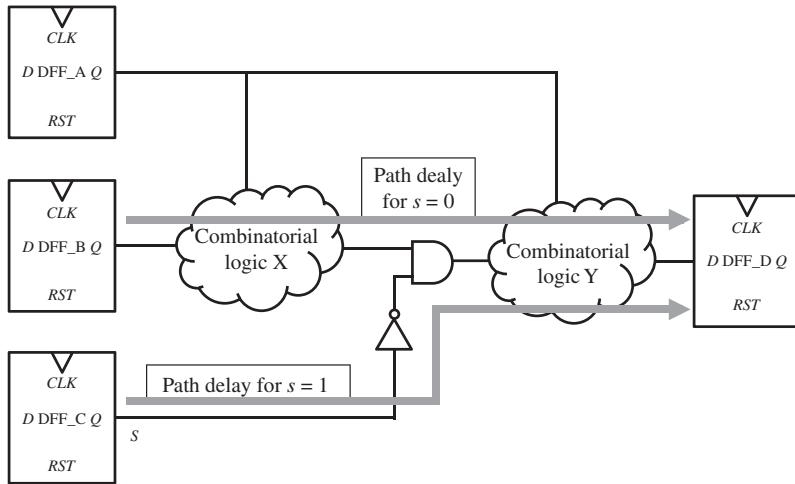
**Exercise 2.3** As shown in Figure 2.21, there are two DFFs connected in sequence. Assume that the clock signal for DFF 2 is delayed for more 100 ps than the clock signal for DFF 1, at which point a hold buffer should be added, point A or point B. How much delay should the hold buffer provide?

## 2.5.2 Static Timing Analysis and Dynamic Timing Analysis

### 2.5.2.1 Timing Analysis without Test Vectors

Once RTL coding is completed, it is synthesized with a library and converted into gate-level netlist so that the behavior of the functional module is represented with bit-wise operations and memory modules including DFF. Therefore, compared to the RTL description, the netlist enables us to evaluate the timing delay of the combinatorial logics much closer. At this moment, designers can perform a rough estimation for the critical timing delay,  $T_{\max}$ , based on the delay model of each gate and wire connecting gates. This analysis method is called **static timing analysis (STA)** named after the way of estimating the delay. That is, STA does not care for the value in the netlist. The advantage of STA is its fast analysis because any functional simulation with a test vector is not required. However, it sometimes causes an accurate result for the delay estimation. More precisely, the result of STA becomes significantly marginal when evaluating the paths that never happen in reality.

In order to see this, consider the example case shown in Figure 2.22. Suppose that DFFs provide inputs to a combinatorial logic that is composed of combinatorial logics X and Y, one INV gate and one AND gate. The outputs of DFF\_A and DFF\_B are connected to the combinatorial logic X. Therefore, at the positive edge of the clock, the gates in the combinatorial logic X start being evaluated. It is assumed that all the signals in the combinatorial logic X are determined in  $T_X$ .



**Figure 2.22** Example circuit for timing analysis

DFF\_C provides the signal  $s$ , and it determines the input for the combinational logic Y. When  $s = 1$ , the combinational logic Y takes 0 and the output of DFF\_A as inputs. In this case, the path delay is estimated from the path between DFF\_C and DFF\_D as  $T_{CD}$ . Here, it is assumed that the path starting from DFF\_A is not a critical one.

On the other hand, in the case of  $s = 0$ , the path delay should be evaluated between DFF\_B and DFF\_D as  $T_{BD}$ . If the delay of the combinational logic X is  $T_{BD} > T_{CD}$ , the critical path delay is derived from the path delay as  $T_{BD}$ . However, it sometimes happens that the signal  $s$  never becomes 0 depending on the usage of the design. In this case, the path between DFF\_B and DFF\_D is regarded as **false path**. Obvious false paths, such as the case shown in Figure 2.22, can be eliminated easily in STA, but it is quite difficult to find all the false paths in a complicated combinational logic.

### 2.5.2.2 Timing Analysis with Test Vectors

Timing analysis with test vectors, which is called **dynamic timing analysis (DTA)**, is also used for timing verification. DTA overcomes the major problem of STA as it excludes all of the false paths and focuses only on the **true paths** of the design. In addition, DTA checks the functionality of the design at the same time. DTA is suitable when the designer can prepare the test vectors such that the critical path is activated. However, DTA has the disadvantage that it requires more analysis time compared to STA. Furthermore, if the prepared test vectors do not activate the critical path, DTA results report a faster clock frequency than what the fabricated chip can work correctly without setup-time errors.

## 2.6 Cost and Performance of Digital Circuits

### 2.6.1 Area Cost

Digital circuits are normally implemented on silicon wafer, and occupy some space. Larger space usage results in a higher cost in fabrication as far as the silicon die size is restricted with the circuit area.<sup>10</sup> Moreover, the larger circuits tend to consume more power if the gates in the circuits are activated in a constant rate. In this regard, cost of a design circuit can be regarded as the size of the circuit or the gate equivalent by one definition.

The size of the circuit can be measured with the actual size, for example, in units of nano square meters ( $\text{nm}^2$ ). The **gate equivalent** is typically calculated by dividing the circuit area by the area of a 2-input NAND gate whose drivability is 1.

### 2.6.2 Latency and Throughput

In the field of digital circuits, performance often means computational speed. In order to improve computational speed, we have several factors to optimize, which are clock frequency used for a digital circuit, the number of clock cycles needed for computation, the amount of data processed in a clock cycle, and so on. Especially, **latency** and **throughput** are the two key metrics for performance.

**Latency** is defined as the response time of computation. Suppose that given computation requires  $N$  cycles, the latency,  $L$ , can be expressed as

$$L = \frac{N}{f}, \quad (2.7)$$

where  $f$  is the clock frequency. Note that the number of clock cycles is also used as another indicator of latency.<sup>11</sup>

**Throughput**,  $R$ , can be derived as

$$R = \frac{D}{L} = \frac{D}{N} \cdot f, \quad (2.8)$$

where  $f$  is the clock frequency and  $D$  the amount of input data processed in a computation. If the amount of data per clock cycle,  $\frac{D}{N}$ , is constant, it is found that the throughput,  $R$ , is simply proportional to the clock frequency,  $f$ . The alternative view is that the amount of data per clock cycle should be increased in order to improve the throughput under the situation, where the clock frequency cannot be increased.

---

<sup>10</sup> In some cases, the number of pins determines the circuit area.

<sup>11</sup> In addition, latency may be measured simply with a response time especially in the case that a circuit does not get triggered with the clock signal, that is, asynchronous operations.

**Exercise 2.4** Assume that there is a given calculation that requires 160 clock cycles to process 1024-bit data, and the critical path delay of its implementation is 40 ns, that is,  $T_{cpd} = 40$  ns. Calculate the minimum latency and the maximum throughput for this calculation.

## Further Reading

- Ercegovac MD and Lang T 2004 *Digital Arithmetic*. Morgan Kaufmann Oxford, San Francisco, CA.  
Hennessy JL and Patterson DA 2002 *Computer Architecture: A Quantitative Approach* 3rd edn. Morgan Kaufmann Publishers Inc., San Francisco, CA.  
(ed. Oklobdzija VG) 2007 *Digital Systems and Applications*, The Computer Engineering Handbook 2nd edn. CRC Press, Boca Raton, FL.

# 3

# Hardware Implementations for Block Ciphers

This chapter introduces several hardware implementations that can be often used for block ciphers, focusing on AES. AES is widespread and plays a significant role to guarantee the security of many secure systems. The AES hardware can be implemented on a wide variety of architectures with synchronous-style design. There exist several options depending on the requirement for speed performance, area cost, and so on. General discussions on hardware architecture are firstly mentioned, and secondly the AES hardware implementations are introduced. Finally, we explore the details of the merits and demerits of each architecture throughout this chapter.

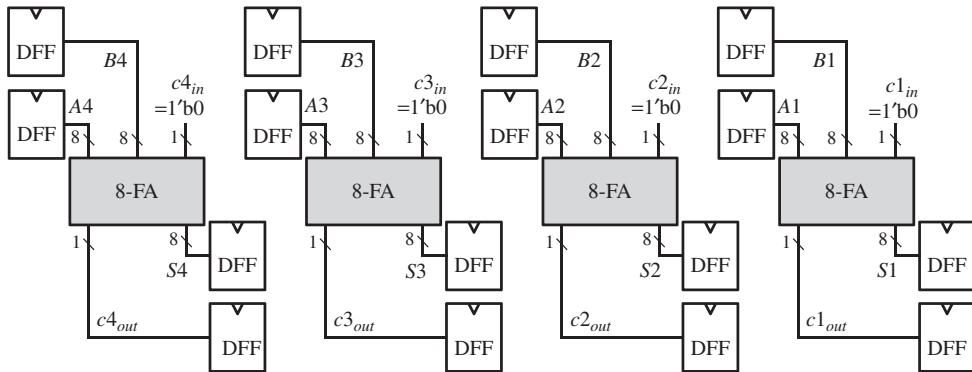
## 3.1 Parallel Architecture

In order to achieve a high-throughput performance, a hardware implementation is designed to process several operations or modules simultaneously. It is called **parallel architecture** and often used in timing-critical and real-time systems. It can also be understood that designers aiming at such parallelism have to increase the value of  $\frac{D}{N}$  discussed in Section 2.6.2.

### 3.1.1 Comparison between Serial and Parallel Architectures

Power consumption of the parallel architecture is relatively large as signal toggles are increased in proportion to the number of parallelized modules. For instance, Figure 3.1 shows an example, where four independent 8-bit additions are implemented in parallel. Compared to the implementation, where only one 8-bit adder is used, the data throughput with the parallel architecture becomes four times higher.

As all the modules running in parallel have to perform addition independently, it has to be assumed that any data dependencies do not exist between modules. When there is a data dependency, they have to be operated sequentially in general. For instance, let us consider the case that a 32-bit addition is implemented with four 8-bit adder modules. The carry bits



**Figure 3.1** Parallel architecture of four 8-bit additions

are propagated among modules as  $c2_{in} = c1_{out}$ ,  $c3_{in} = c2_{out}$ , and  $c4_{in} = c3_{out}$ ; the parallel architecture illustrated in Figure 3.1 cannot be employed as the carry-in signals of the left three 8-bit FAs are fixed to zeros.

Figure 3.2 illustrates a serial architecture for 32-bit adder. It takes four cycles to complete 32-bit addition since it is based on 8-bit additions. As can be seen from the figure, the architecture requires several selectors, which causes an extra area cost and path delay.<sup>1</sup> Nevertheless, the critical-path delay is expected much lower than the hardware with combinatorial logics of a 32-bit carry-ripple adder.

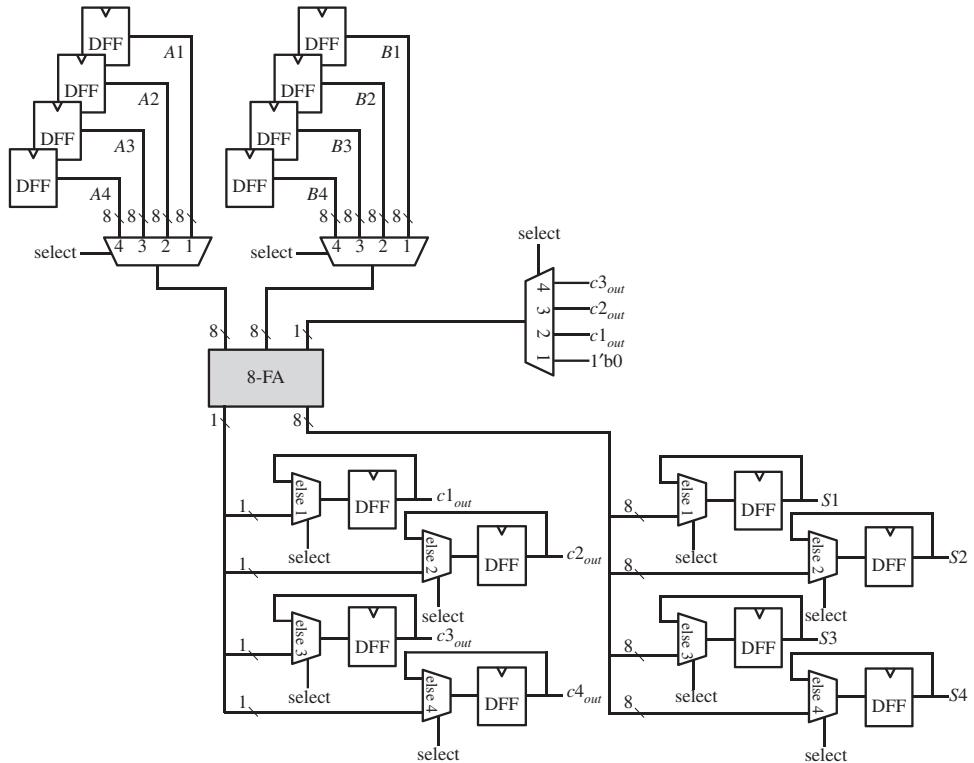
### 3.1.2 Algorithm Optimization for Parallel Architectures

An algorithmic optimization sometimes solves the data-dependency problem or facilitates the data parallelism. In a straightforward implementation, only right-most adder module in Figure 3.1 can start a correct operation as  $c_{in}$  signals of other three modules are not determined. However, considering the fact that the value of  $c_{in}$  has at most two possibilities, 0 or 1, the three adder modules can start operation assuming the two cases,  $c_{in} = 0$  or 1. The corresponding architecture is shown in Figure 3.3.

In this case, the number of total adder modules increases up to seven in total. This adder is famous as the **carry-select adder**. Although there exist several selector logics to select the addition results, it can be seen that this optimization makes the best use of the parallel processing of 8-bit adders.

**Exercise 3.1** Make a comparison table for the speed performance (throughput and latency) and area cost for the hardware implementations shown in Figures 3.1–3.3.

<sup>1</sup> The cost can be reduced if a shared hardware resource is available for memory such as an SRAM module.



**Figure 3.2** Serial architecture for 32-bit adder (multi-cycle carry-ripple adder)

### 3.2 Loop Architecture

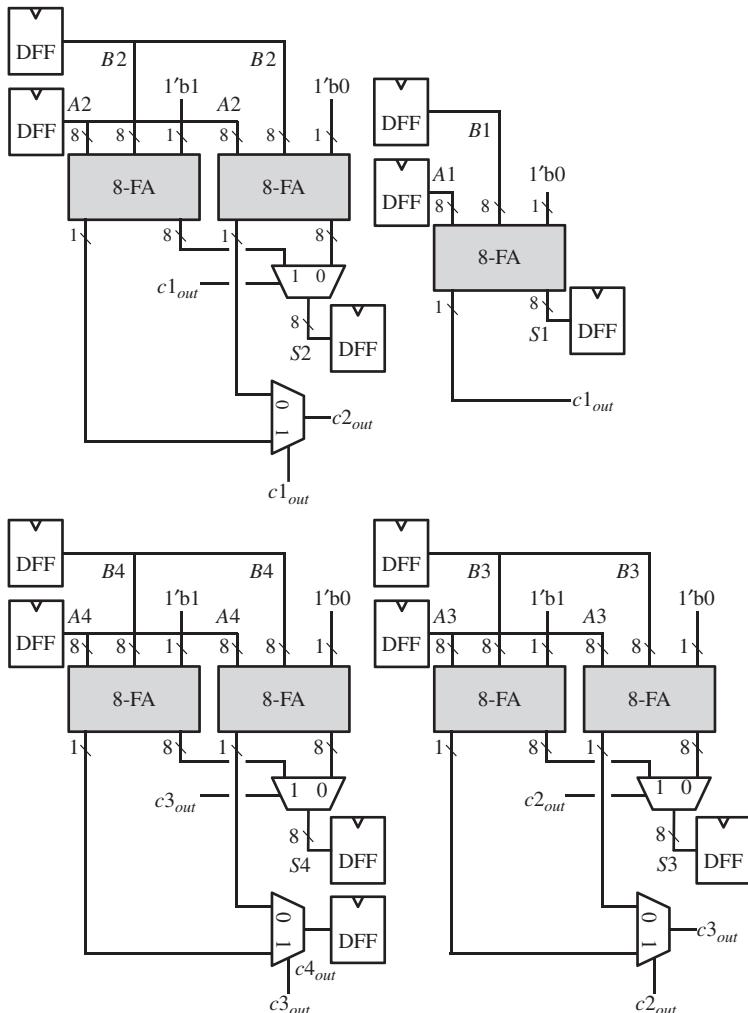
In order to utilize a hardware resource efficiently, iterative use of combinatorial logics is often seen in hardware implementations. Combined with DFFs (delay flip flops), such hardware is called **loop architecture**. A simplified block diagram for the loop architecture is illustrated in Figure 3.4.

#### 3.2.1 Straightforward (Loop-Unrolled) Architecture

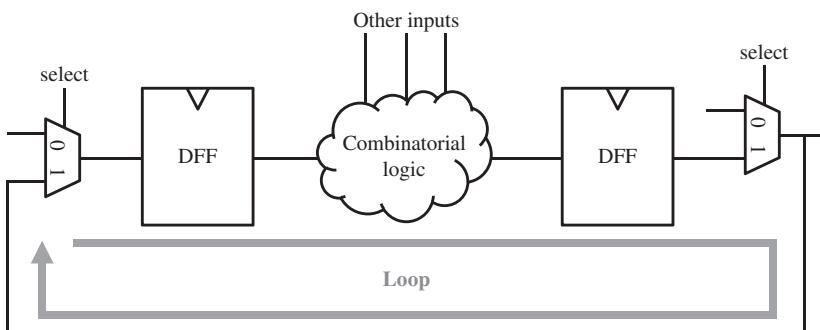
The loop architecture is basically for reducing the hardware cost efficiently, while maintaining the speed performance. As one of the simple examples, let us see a hardware implementation that performs a modular computation of

$$S = (A_1 + A_2 + \dots + A_8) \bmod 2^8, \quad (3.1)$$

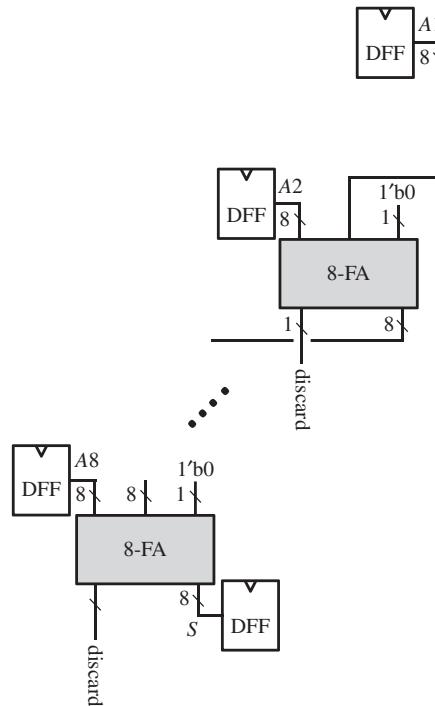
where  $A_1, A_2, \dots, A_8$ , are all 8-bit positive integers. The straightforward (**loop-unrolled**) implementation for this example is to prepare seven 8-bit adder modules as combinatorial logics as shown in Figure 3.5.



**Figure 3.3** Parallelized architecture for 32-bit adder (carry-select adder)



**Figure 3.4** Loop architecture



**Figure 3.5** Straightforward (loop-unrolled) implementation of 8-operand modular addition

Eight operands for the addition,  $A_1, A_2, \dots, A_8$ , are all assumed to be stored in 8-bit DFFs as they should be provided to the combinatorial logic at the same timing. The operation result,  $S$ , is performed in a single clock cycle and stored in 8-bit DFFs after the addition. In this case, the critical-path delay becomes  $7 \cdot T_{8add}$ , where  $T_{8add}$  is the critical-path delay of the 8-bit adder module.

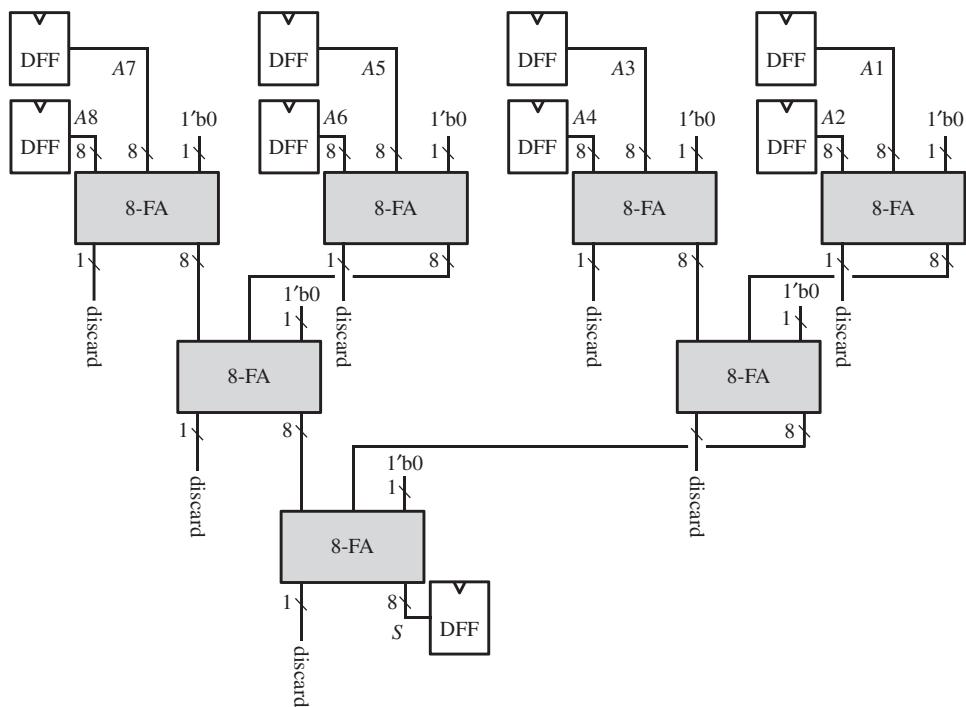
By changing the connection of 8-bit FAs as shown in Figure 3.6, the critical-path delay is shortened to  $3 \cdot T_{8add}$ , whereas the area cost is the same.

### 3.2.2 Basic Loop Architecture

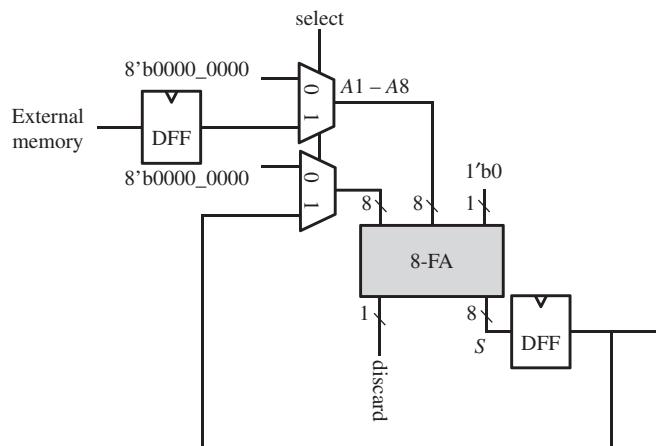
Alternatively, the same functionality is realized only with one 8-bit adder module as shown in Figure 3.7. This architecture is easy to understand by dividing the computation into sequential ones as

$$\left\{ \begin{array}{l} S = (S + A_1) \bmod 2^8, \\ S = (S + A_2) \bmod 2^8, \\ \vdots \\ S = (S + A_8) \bmod 2^8, \end{array} \right.$$

where  $S = 0$  is assumed before the computation.



**Figure 3.6** Optimized implementation of 8-operand modular addition



**Figure 3.7** Loop architecture for 8-operand modular addition

One of the two inputs for 8-bit adder can be provided from external memory such as SRAM, and another input is connected to the output of DFF storing intermediate addition result and the final result. The critical-path delay becomes  $T_{8add}$ , however, it takes eight cycles to finish the computation. If the additional timing penalty caused by the selector, the clock skew, and the setup timing margin is small enough, the speed performance of the above mentioned two architectures shown in Figures 3.5 and 3.7 can be regarded similar under the maximum clock frequency.

The advantage of the latter loop architecture is obviously in its effectiveness of area cost. Moreover, the loop architecture offers **scalability**, that is, more than eight operands can be used for modular addition with a slight modification on the select signals. The loop architecture enables us to perform more iterations such as a 100-operand modular addition.

**Exercise 3.2** *Make a comparison table for the speed performance (throughput and latency) and area cost for the hardware implementations shown in Figures 3.5–3.7.*

### 3.3 Pipeline Architecture

**Pipeline architecture** is commonly used in high-performance RISC (reduced instruction set computer) CPUs (central processing units) to support a flexible and efficient software implementation. The grain size of the computation of instructions in an RISC CPU is small enough as they just support basic operations. Accordingly, the operation of each pipeline stage becomes simple, and the clock frequency can be increased so that the throughput of the software could be accelerated. Note that the pipeline will stall if there is data-dependent operations, which is called **pipeline stall**. It is known that such stalls diminish the performance merit of the pipeline architecture.

#### 3.3.1 Pipeline Architecture for Block Ciphers

The pipeline architecture can also be applied to hardware implementations for block ciphers in order to accelerate multiple independent operations with a low area cost. The number of the pipeline stages should be defined according to the degree of parallelism necessary for a target application.

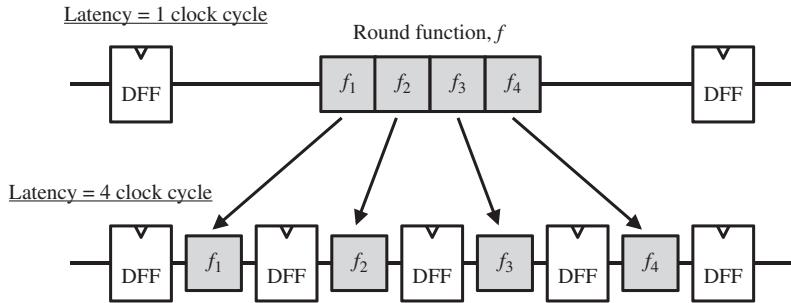
Here, let us consider a cryptographic application that requires an encryption hardware of a 10-round block cipher. Suppose that the round function  $f$  is divided into four functions as

$$f = f_4 \circ f_3 \circ f_2 \circ f_1, \quad (3.2)$$

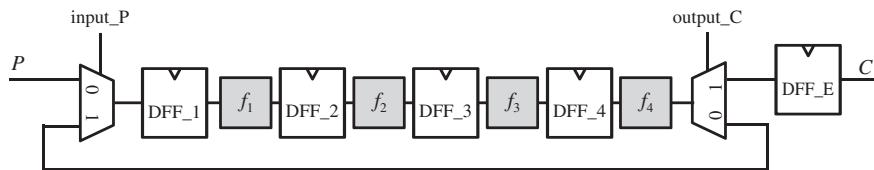
in order to perform at a high clock frequency. Additionally, assuming that each round operation is identical for simplicity, the 10-round encryption function is described as

$$C = f^{10}(P), \quad (3.3)$$

where  $P$  and  $C$  are plaintexts and ciphertexts, respectively.



**Figure 3.8** Four-stage architecture for the round function,  $f$



**Figure 3.9** Four-stage pipeline architecture for 10-round encryption

The combinatorial logics for the round function,  $f$ , is divided, and three more DFFs are inserted between the small functions,  $f_i$  ( $1 \leq i \leq 4$ ) as shown in Figure 3.8. As a result, the latency becomes four clock cycles.

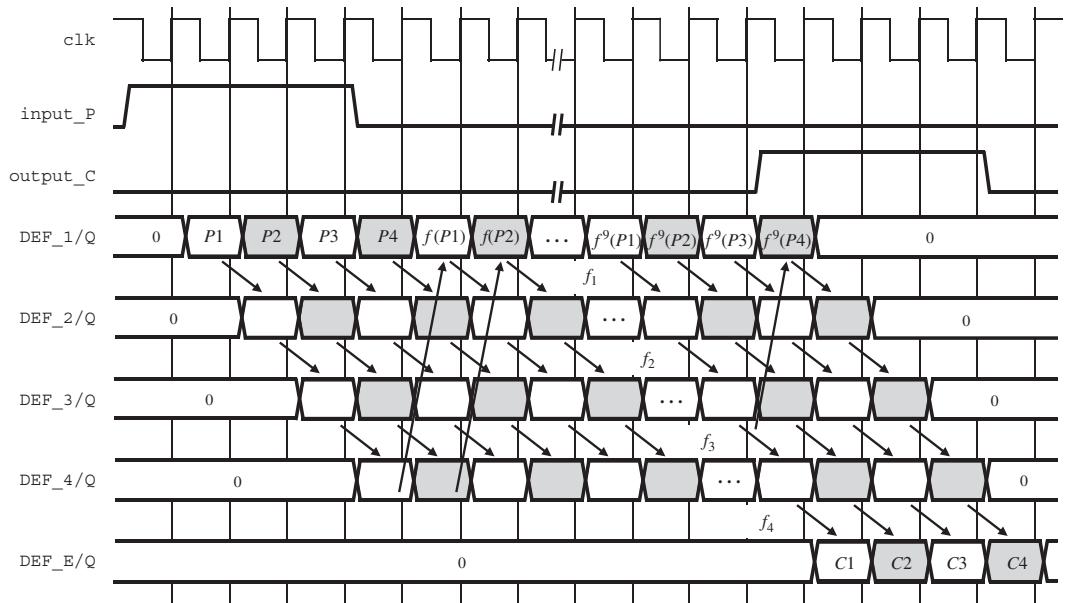
### 3.3.2 Advanced Pipeline Architecture for Block Ciphers

On the basis of the four-stage architecture, 10-round loop architecture is implemented as illustrated in Figure 3.9. The throughput performance can be improved up to by four times by executing multiple encryption operations in parallel. Note that, when executing the encryption for a single plaintext, the architecture does not have any merits as for the throughput and latency.

Let us see the details of the effect of pipeline architecture for the iterative operations in the timing waveform shown in Figure 3.10. The first plaintext,  $P1$ , is processed after it is stored, in DFF\_1, and  $f_1(P1)$  is stored in DFF\_2 in the next cycle. Suppose that the size of plaintext is  $w$ . Likewise, one round operation for  $P1$  is complete in four cycles, and the result of  $f(P1)$  is stored again in DFF\_1. This means that operations,  $f_1, f_2, f_3$ , and  $f_4$ , are used only once in four cycles as for the round operation for  $P1$ . Therefore, the throughput becomes

$$R = \frac{w \cdot f_{clk}}{40},$$

when operating a single plaintext with the architecture illustrated in Figure 3.9. This is because a pipeline stall happens.



**Figure 3.10** Timing waveform for four-stage pipelined 10-round encryption

In order to make the best use of the architecture's capability, four different plaintexts,  $P_1$ ,  $P_2$ ,  $P_3$ , and  $P_4$ , are sent to DFF\_1 in four consecutive cycles. In this case, the throughput is improved four times higher than the previous case as

$$R = \frac{4w \cdot f_{clk}}{40} = \frac{w \cdot f_{clk}}{10}.$$

**Exercise 3.3** Discuss the merits and demerits of increasing the number of pipeline stages.

### 3.4 AES Hardware Implementations

In this section, we will consider the several different hardware implementations for AES-128, and compare the cost and speed performance.

#### 3.4.1 Straightforward Implementation for AES-128

One of the most straightforward implementations for AES-128 encryption hardware is shown in Figure 3.11. It operates 10-round AES-128 encryption in one cycle. It starts from the AddRoundKey operation that XORs the plaintext and the subkey,  $sk_0$ .

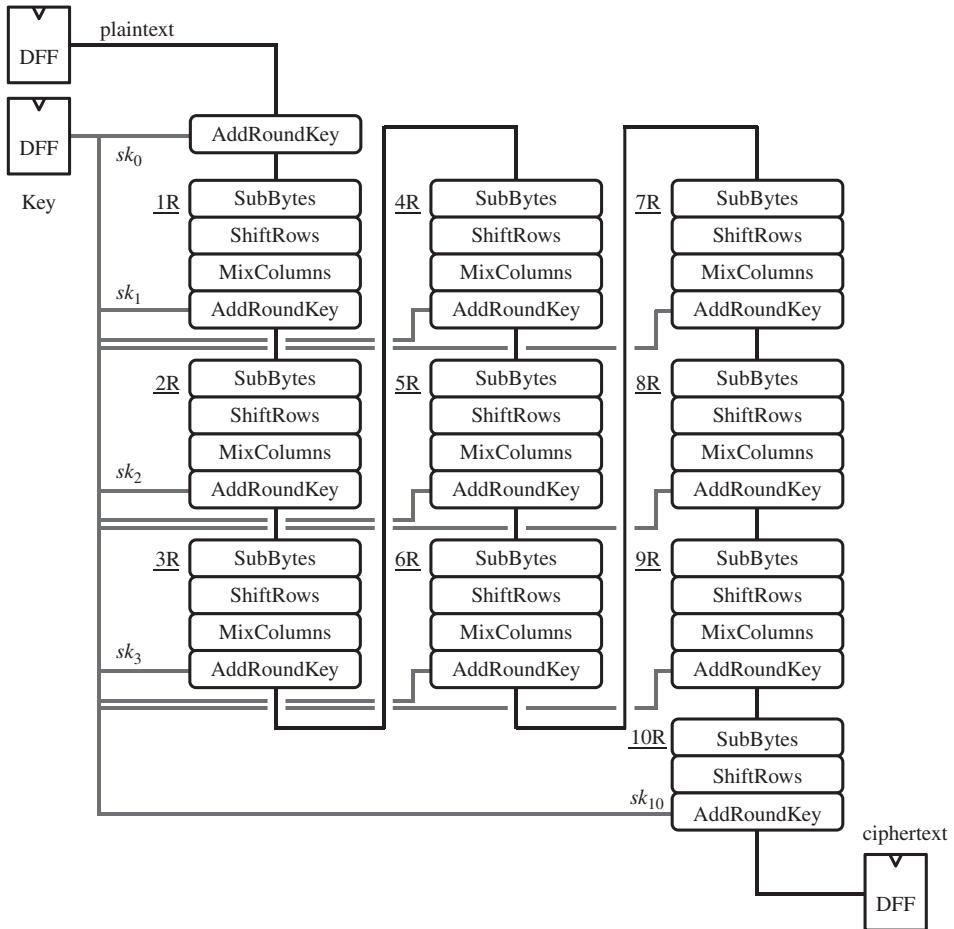
The round operation consists of four transformations, SubBytes, ShiftRows, MixColumns, and AddRoundKey. The 128-bit AES state is aligned in a byte-wise manner, and each byte of them is independently operated in the SubBytes transformation. The ShiftRows transformation changes the alignment of the bytes, which can be realized just by wiring in hardware implementations. Then, the MixColumns transformation performs a finite field operation in  $GF(2^8)$ , where the four-byte inputs transform four-byte outputs (see the details in Section 1.5). Finally, the round operation generates 128-bit data after the AddRoundKey transformation, where a subkey,  $sk$ , is added to the output of the MixColumns transformation. The round operation is performed, in total, 10 times, and the ciphertext is output. Note that, the MixColumns transformation is not performed in the 10th round operation.

The AES-128 decryption hardware can be implemented by reversing the order of the encryption transformations as shown in Figure 3.12. Notice that the inverse of the AddRoundKey transformation is the same as the AddRoundKey transformation as they are both XOR operations. It is worth mentioning that the transformation order of (Inv)SubBytes and (Inv)ShiftRows in the round operation can be easily switched as the (Inv)ShiftRows transformations is implemented with byte-wise wiring.<sup>2</sup> Therefore, it is possible to switch the order of (Inv)SubBytes and (Inv)ShiftRows transformations.

Even if switching the order of InvShiftRows and InvSubBytes in Figure 3.12, the order of transformations in the round operation is still slightly different between Figures 3.11 and 3.12. That is, in order to have the same sequence of transformations for both AES encryption and decryption, ignoring the inverse properties, the order of the AddRoundKey

---

<sup>2</sup> However, the wiring overhead may cause the penalty in the area cost and the critical-path delay.



**Figure 3.11** Straightforward implementation for AES-128 encryption

and InvMixColumns transformations needs to be changed. In changing the order, the subkeys,  $sk_9$  to  $sk_1$ , are required to go through the InvMixColumns transformation, that is,

$$sk'_i = \text{InvMixColumns}(sk_i), \quad (3.4)$$

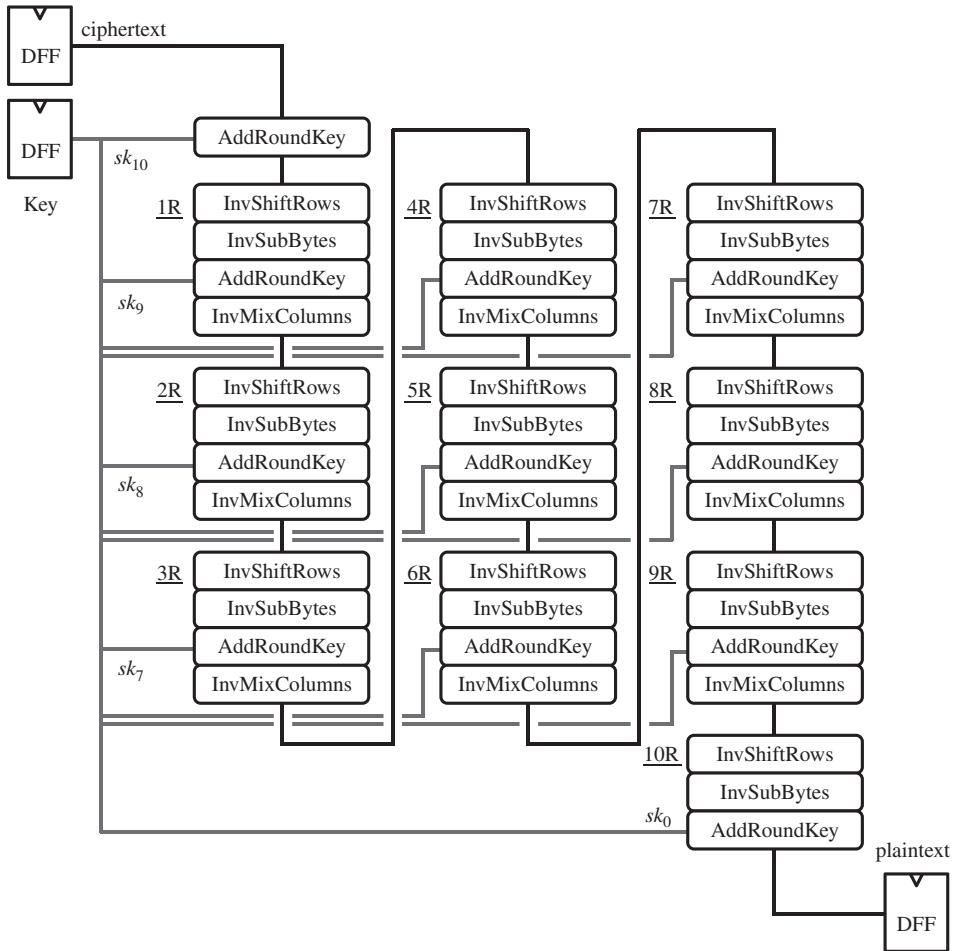
where  $i = \{1, 2, \dots, 9\}$  because

$$\text{InvMixColumns}(I \oplus sk_i) = \text{InvMixColumns}(I) \oplus \text{InvMixColumns}(sk_i) \quad (3.5)$$

$$= \text{InvMixColumns}(I) \oplus sk'_i, \quad (3.6)$$

where  $I$  is an intermediate value that is output from InvShiftRows and InvSubBytes transformations.

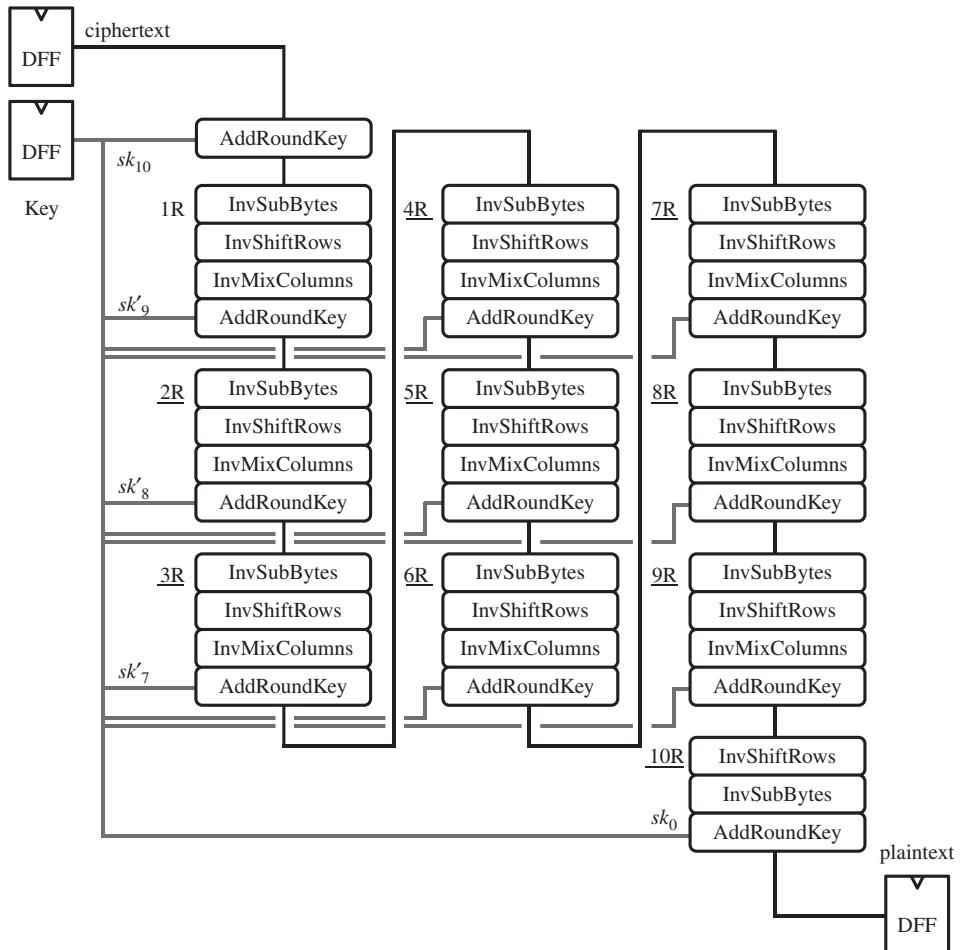
Figure 3.13 shows another AES-128 decryption hardware that has the same operation sequence of the AES-128 encryption shown in Figure 3.11. The modification in Figure 3.13



**Figure 3.12** Straightforward implementation for AES-128 decryption

motivates us to implement a round operation that can be used for both encryption and decryption in order to optimize the hardware cost. That is, one of the optimizations in hardware is to implement each transformation so that its inverse transformation could also be supported efficiently with sharing a hardware resource.

Here, we excluded the detailed descriptions for the Key Scheduling Function or KSF module that generates 128-bit subkeys provided to the AddRoundKey transformation. It can be performed off-line, that is, it can be precomputed before the encryption and decryption, as far as the secret key is not frequently changed and there is enough memory space to store the subkeys. Alternatively, in order to reduce the hardware cost, the subkey generation can be performed round by round. In the following sections, we will explore several different types of implementations for AES-128.

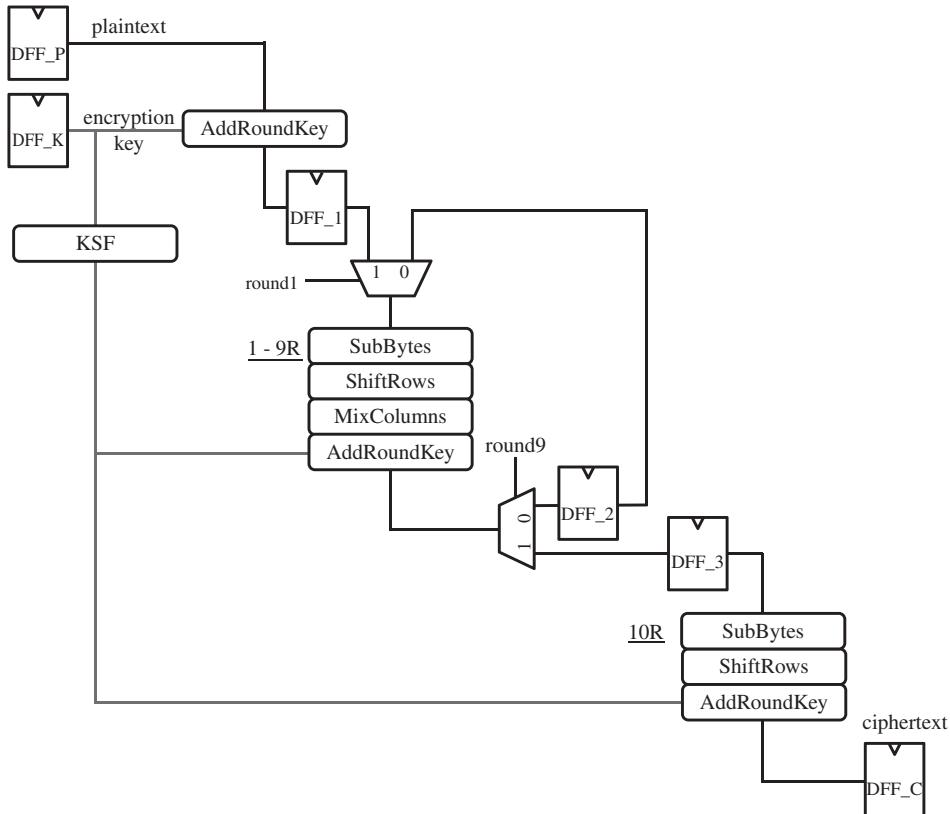


**Figure 3.13** Straightforward implementation for AES-128 decryption with modified key scheduling

**Exercise 3.4** Consider the straightforward implementations of AES-192 and AES-256, and compare their performance with the straightforward implementation of AES-128.

### 3.4.2 Loop Architecture for AES-128

Figure 3.14 shows a loop architecture for AES. The first AddRoundKey operates with the plaintext and the encryption key, and the result is stored in DFF\_1. Then, the round operations



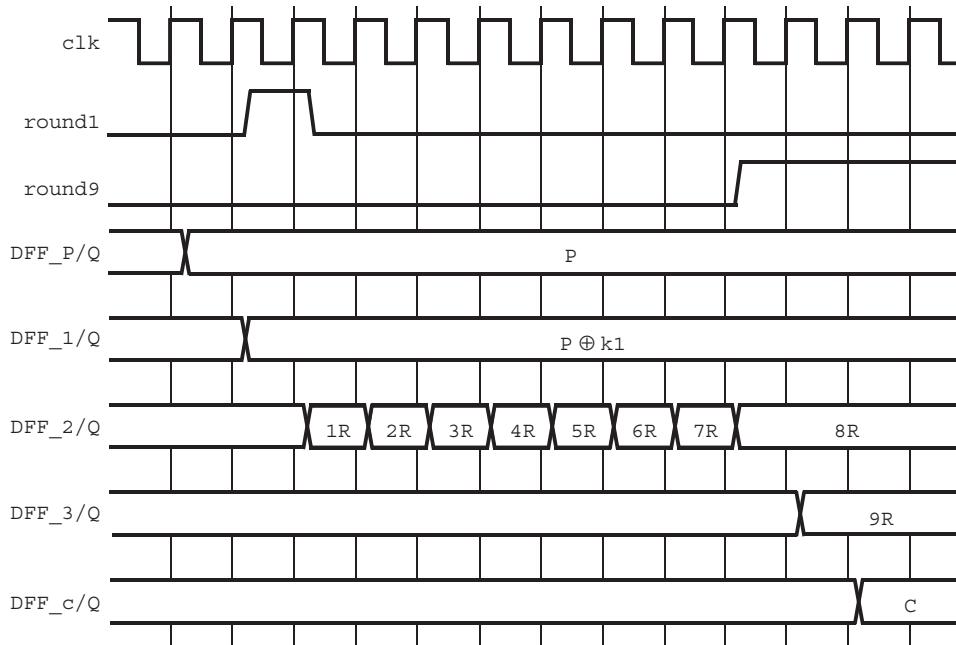
**Figure 3.14** Loop architecture (I) for AES-128 encryption

are iteratively performed nine times with the loop architecture including DFF\_2 in the datapath. The result of the ninth round (9R) is stored in DFF\_3, whereas DFF\_2 keeps the result of the eighth round (8R). The 10th round operation is performed on the output of 9R, and the result is stored in DFF\_C as ciphertext. The corresponding timing waveform can be found in Figure 3.15.

This loop architecture requires three AddRoundKey blocks, two MixColumns and ShiftRows blocks, and one MixColumns block. The total number of DFFs is  $5 \cdot 128$  or 640 except the DFFs for encryption key. Here, we ignore the hardware cost for KSF as the implementation style varies for the use case, for example, encryption key is fixed for a long term.

The critical-path delay in the architecture can be estimated as

$$T_{cpd} = T_{SB} + T_{SR} + T_{MC} + T_{AK} + T_{MUX} + T_{DEMUX} + T_{margin}, \quad (3.7)$$



**Figure 3.15** Timing waveform for loop architecture in Figure 3.14

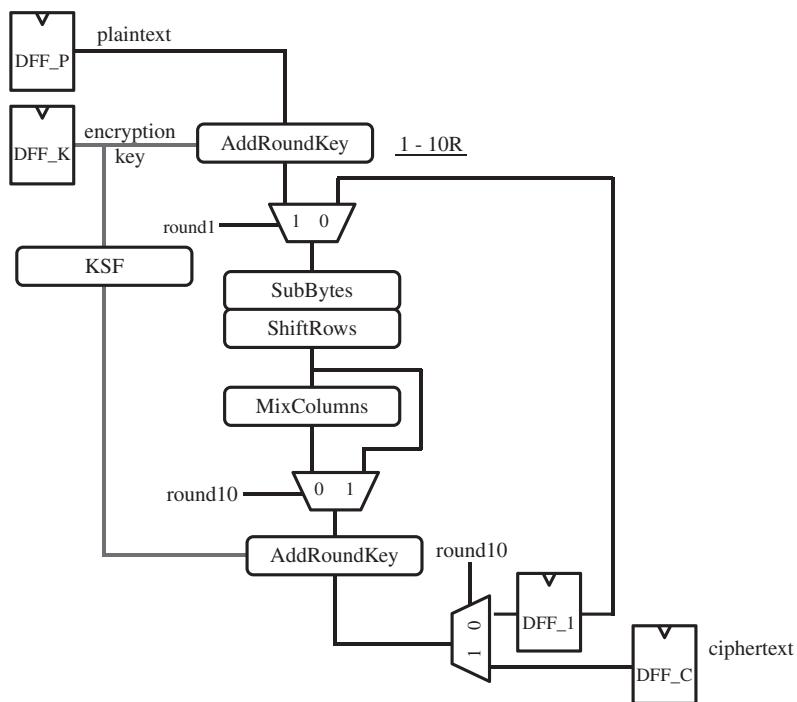
where  $T_{SB}$ ,  $T_{SR}$ ,  $T_{MC}$ , and  $T_{AK}$  are the delay times in MixColumns, ShiftRows, MixColumns, and AddRoundKey blocks.  $T_{MUX}$  and  $T_{DEMUX}$  are the delay times for multiplexer and demultiplexer, and  $T_{margin}$  takes the setup time and clock skew into consideration.

There are several ways to reduce the hardware cost. One alternative is shown in Figure 3.16, where two AddRoundKey, one MixColumns, one ShiftRows, and one MixColumns blocks are used. The timing waveform is depicted in Figure 3.17. It is found that the MixColumns block is skipped only at the 10th round. Only three DFFs are needed to support this loop architecture except the DFFs for encryption key.

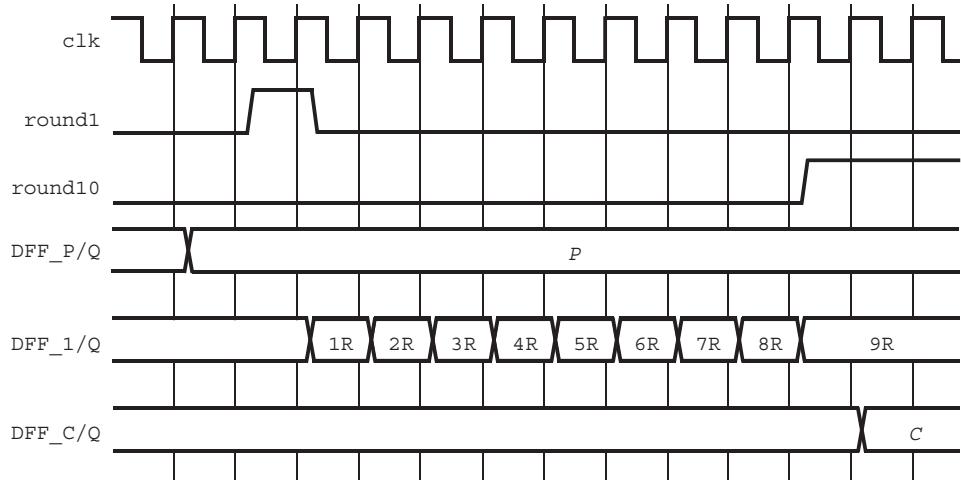
However, the critical-path delay is increased as

$$T_{cpd} = T_{SB} + T_{SR} + T_{MC} + 2 \cdot T_{AK} + 2 \cdot T_{MUX} + T_{DEMUX} + T_{margin}. \quad (3.8)$$

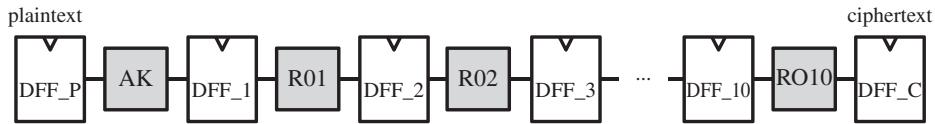
**Exercise 3.5** Consider the implementations of AES-192 and AES-256 with loop architecture, and compare their performance with the implementation of AES-128 with loop architecture.



**Figure 3.16** Loop architecture (II) for AES-128 encryption



**Figure 3.17** Timing waveform for loop architecture in Figure 3.16



**Figure 3.18** High-throughput pipeline architecture for AES-128

### 3.4.3 Pipeline Architecture for AES-128

On the basis of the discussion in Section 3.3, we consider two-level pipeline architecture for AES-128 encryption. The first-level pipeline is for the 10-round operation, and the second-level one exploits the operations in each round operation.

#### 3.4.3.1 High-Throughput Architecture

When AES encryption is used in practice, a mechanism called **mode of operation** is employed for security reasons.<sup>3</sup> Among various mode of operations, the so-called **counter mode** or **CTR mode** uses a counter value as a plaintext for AES encryption, and the message is XORed with the encryption result. Therefore, even for a long message, all the AES encryptions can be parallelized using the counter values. Accordingly, the throughput performance is the same as that for one AES encryption.

In this regard, the pipeline architecture shown in Figure 3.18 is appropriate as the amount of data processed per cycle is fast enough (128 bits/cycle), and a high clock frequency is expected. If the architecture operates at 100 MHz, the throughput becomes 12.8 Gbps.

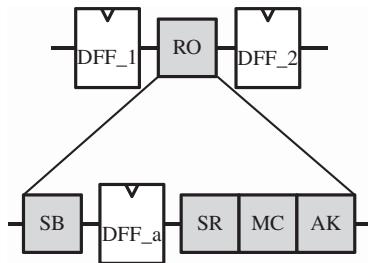
What can be done for further improvements in throughput? One promising way is to reduce the critical-path delay by introducing additional pipeline stages in the critical block. In the case of AES encryption, MixColumns tends to be the critical block as far as its nonlinear transformation is implemented with combinatorial logics.<sup>4</sup> Therefore, it is natural to divide the MixColumns transformation into several small operations, or prepare an additional DFF immediately after MixColumns as shown in Figure 3.19. Note that throughput improvements often come with the price in hardware cost.

#### 3.4.3.2 Combination with Loop Architecture

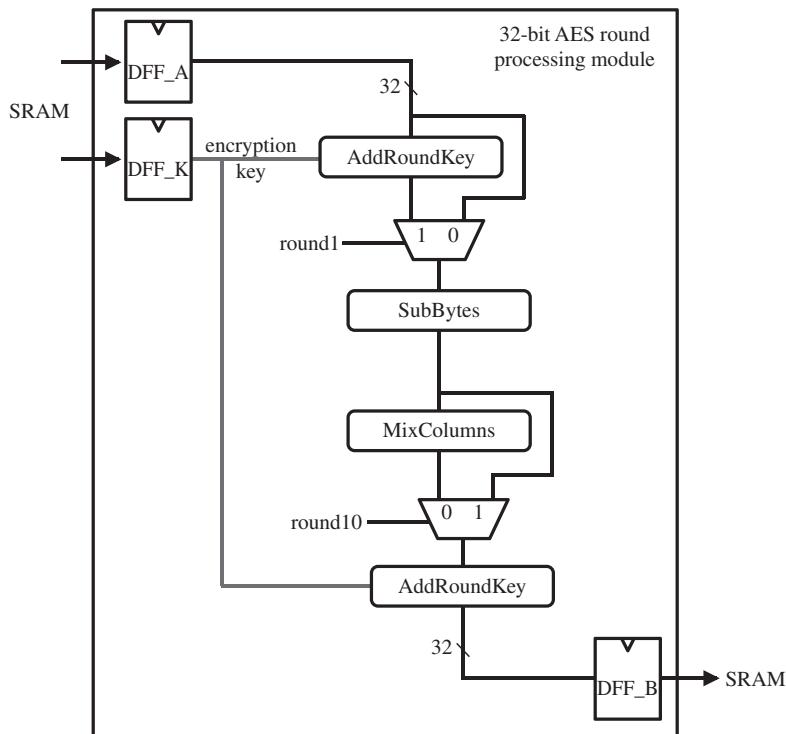
It is possible to apply the pipeline to the loop architecture as shown in Figure 3.9. In this case, the cost area will be much lower than the case for the architecture in Figure 3.18.

<sup>3</sup> For a straightforward block cipher usage if the message contains the same 128-bit plaintext blocks, one can know that they are identical from the ciphertext, which may leak information.

<sup>4</sup> Look-up table implementation is normally faster than the logic implementation, but it requires  $2^8$  or 256 bytes of memory for AES S-box. See the AES S-box table in Table 1.6.



**Figure 3.19** Pipeline in the round operation



**Figure 3.20** Round operation with 32-bit or 4-byte datapath

### 3.4.4 Compact Architecture for AES-128

In the previous discussions, the width of the datapath is all fixed to 128 bits. As we can easily imagine from the fact that 8-bit or 32-bit CPU can operate AES-128, the datapath can be narrow by performing byte-wise operations. For instance, the architecture shown in Figure 3.20 employs 32-bit or 4-byte datapath that performs 4-byte operations in one cycle. That is, it requires four cycles to complete a round operation.

All the intermediate values are stored in a memory module such as SRAM. In the case of the architecture of Figure 3.20, 4-byte data should be aligned in accordance with the ShiftRows transformation when writing the data to SRAM or reading out from SRAM. The merit of using 4-byte datapath is that one MixColumns operation, which operates 4 bytes, can be performed in one cycle.

## Further Reading

(ed. Verbauwhede IM) 2007 *Secure Integrated Circuits and Systems*. Springer-Verlag.



# 4

# Cryptanalysis on Block Ciphers

In this chapter, several cryptanalyses against block ciphers are introduced. The discussion is mainly focused on AES-128, which is the AES with a 128-bit key, although many of the discussions can also be applied to other block ciphers in general.

The ideal security of block ciphers and the goal of the cryptanalysis are firstly defined. Then, many techniques in several cryptanalytic approaches are introduced. The first topic is the differential cryptanalysis, which provides the basic concepts of cryptanalysis. The second topic is the impossible differential cryptanalysis. Finally, the last topic is the integral cryptanalysis. Key recovery attacks against reduced-round versions of AES-128 are demonstrated for each approach.

## 4.1 Basics of Cryptanalysis

### 4.1.1 Block Ciphers

Block cipher  $E$  takes as input a key  $K$  of a fixed bit-length, and produces a one-to-one map (permutation) from  $b$ -bit plaintext to  $b$ -bit ciphertext, where  $b$  is a bit-length of the fixed block size. Let  $k$  be a bit-length of the fixed key size. Then, block ciphers are described as follows:

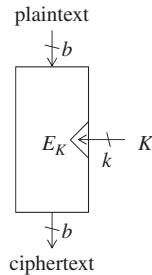
$$K \in \{0, 1\}^k \quad (4.1)$$

$$E_K : \{0, 1\}^b \rightarrow \{0, 1\}^b. \quad (4.2)$$

The permutation  $E_K$  is required to behave completely independent for different choices of  $K$ . The model of block cipher is depicted in Figure 4.1.

On the one hand, from mathematical perspective, any key size and any block size are acceptable. On the other hand, from engineering perspective, suitable choices of those sizes are limited.

The key size is often determined by a cost of managing keys and required security for the block cipher. Long keys have higher security than short keys, while the managing cost for long keys is more expensive than the one for short keys.



**Figure 4.1** Model of block cipher

**Table 4.1** Key size and block size of widely used block ciphers

Algorithm	Key size, $k$	Block size, $b$
AES-128	128	128
AES-192	192	128
AES-256	256	128
Camellia-128	128	128
Camellia-192	192	128
Camellia-256	256	128
Triple-DES	168	64
PRESENT	80	64
PRESENT	128	64
HIGHT	128	64

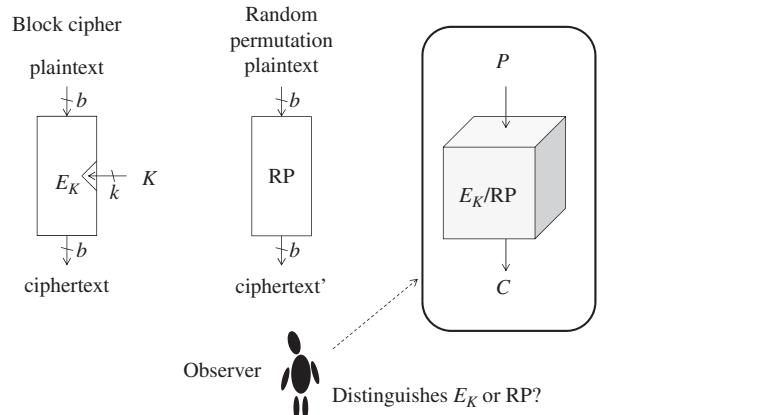
The block size is often determined by the processor size of the target platform. For example, if block ciphers are designed to be used in a high-end software with a 64-bit processor, typical choices of the block size are a multiple of 64 bits such as 128 bits or 256 bits. If they are designed to be suitable for resource-restricted environment such as an RFID tag, a smaller block size is chosen. Several examples of the key size and block size of widely used block ciphers are summarized in Table 4.1.

#### 4.1.2 Security of Block Ciphers

Block ciphers are required to provide some robustness against cryptanalysis. To measure the security of block ciphers, security notions must be defined. There are several classes of security notions. Three major notions are **key recovery resistance**, **plaintext recovery resistance**, and **indistinguishability** from a random permutation.

**Key recovery resistance:** For any choice of the key  $K$ , the block cipher must resist the attack that efficiently recovers the value of  $K$ .

**Plaintext recovery resistance:** For any choice of the key  $K$ , and for any choice of the ciphertext  $C$ , the block cipher must resist the attack that efficiently recovers the corresponding plaintext value  $P$  such that  $E_K(P) = C$ .



**Figure 4.2** Indistinguishability

**Indistinguishability:** For any choice of the key  $K$ , the permutation  $E_K$  must be indistinguishable from a random permutation. This notion is often referred as **pseudo-random permutation**. In detail, under a fixed key, a block cipher becomes a  $b$ -bit permutation. Without the knowledge of the key  $K$ , nobody can compute a valid pair of plaintext and ciphertext. Then, another  $b$ -bit permutation is chosen randomly from all the possible  $b$ -bit permutations, which is called a **random permutation**. Suppose that there is a  $b$ -bit permutation in which either of a block cipher with an unknown fixed key or a random permutation is implemented. An observer aims to distinguish which of the block cipher or the random permutation is implemented without the knowledge of the key. The indistinguishability is depicted in Figure 4.2.

Here, the term “efficiently” in the security notions leaves ambiguity. More details are explained in a later section explaining generic attacks.

If the key recovery resistance is broken on a block cipher, the other two notions are broken automatically. Therefore, the key recovery resistance is the weakest security notion among the above three. That is to say, from a designer’s point of view, the key recovery resistance is the easiest security notion to satisfy, while from an attacker’s point of view, it is the hardest security notion to break. The definition of “success of the attack” depends on which class of security notions is expected on that cipher. In general, block ciphers are expected to have ideal security. Thus, efficiently breaking indistinguishability is considered to be a significant vulnerability for block ciphers.

#### 4.1.3 Attack Models

To measure the security of block ciphers, not only the security notion but also the attacker’s ability must be defined. The assumed attacker’s ability is often referred as the **attack model**, which is mainly classified by the information that attackers can obtain.

**Ciphertext only attack:** The attacker can only observe ciphertexts produced by the encryption system. As long as plaintexts are chosen accordingly to the uniform distribution, the

corresponding ciphertexts also follow the uniform distribution. Therefore, in the ciphertext only attack, the attacker must suppose the bias of the plaintexts such as the bias in natural languages or ASCII codes. It is rare to see that modern symmetric-key ciphers are broken by the ciphertext only attack.

**Known plaintext attack:** The attacker can observe pairs of plaintext and the corresponding ciphertext. Owing to the knowledge of plaintexts, the known plaintext attack is stronger than the ciphertext only attack. The known plaintext attack, in practice, simulates the attacker who eavesdrops the communication between two players. The attacker does not have any control of the plaintext to be encrypted.

**Chosen plaintext attack:** The attacker has an ability to make the system encrypt any plaintext of the attacker's choice. The attacker can observe the corresponding ciphertext. Owing to the ability of choosing plaintexts, the chosen plaintext attack is stronger than the known plaintext attack. The chosen plaintext attack, in practice, simulates the attacker who impersonates a valid communication player. In the chosen plaintext attack, the plaintexts to be encrypted must be chosen before the attack. Once one of the chosen plaintexts is encrypted, the attacker loses the ability to choose new plaintexts.

**Adaptively chosen plaintext attack:** The attacker has an ability to choose plaintexts to be encrypted even after chosen messages are encrypted by the system. The adaptively chosen plaintext attack, in practice, simulates the online active attackers who change the attack strategy depending on the behavior of the system.

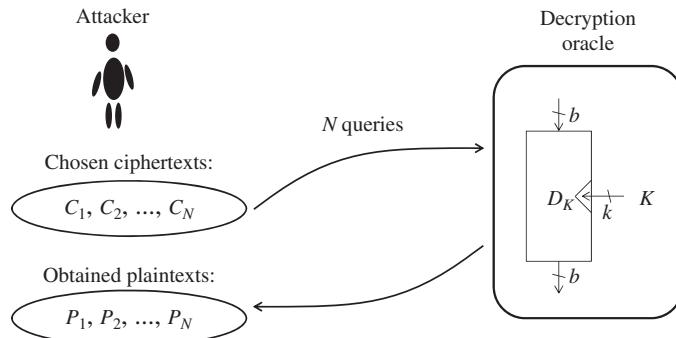
**Chosen ciphertext attack:** Chosen ciphertext attack is a similar model as the chosen plaintext attack. The attacker has an ability to make the system decrypt any ciphertext of the attacker's choice. From mathematical perspective, the ability of choosing plaintexts is as strong as the ability of choosing ciphertexts. However, considering the usage in practice, the latter model is often regarded to be stronger than the former model.

**Adaptively chosen ciphertext attack:** Adaptively chosen ciphertext attack is a similar model as the adaptively chosen plaintext attack. The attacker has an ability to choose ciphertexts to be decrypted even after chosen ciphertexts are decrypted by the system.

It is possible to combine models for the plaintext and the ciphertext. Thus, the strongest attack model that can be directly obtained from the above classification is the adaptively chosen plaintext and ciphertext attack.

Block ciphers are usually required to have ideal security. Thus, the indistinguishability must be ensured against the adaptively chosen plaintext and ciphertext attack.

Note that the attackers cannot compute the correspondence between plaintexts and ciphertexts by themselves because the key value is unknown. In any of the above-mentioned attack models, the corresponding ciphertext (resp. plaintext) of known or chosen plaintext (resp. ciphertext) can be obtained only by forcing the system to output the result. Those requests to the system are called **queries**. A system that returns results from queries is called an **oracle**. In particular, an oracle that accepts plaintexts as queries and returns the corresponding ciphertexts is called an **encryption oracle**. Similarly, an oracle that accepts ciphertexts as queries and returns the corresponding plaintexts is called a **decryption oracle**. As an example, the chosen ciphertext attack is described in Figure 4.3.



**Figure 4.3** Chosen ciphertext attack accessing decryption oracle

Other than the above-mentioned models, cryptographers developed various types of attack models. Some of them were introduced only for an academic interest. Two examples of those complicated models are introduced below, however, those are not discussed more in this book.

**Multikey model:** There are several users with their own keys in the system. The attacker, under some plaintext or ciphertext model, can access any user. For example, if the attack model is the known plaintext attack in the multikey model, the attacker can eavesdrop pairs of plaintext and ciphertext from multiple users.

**Related-key model:** The related-key model is a variant of the multikey model, in which the attacker has more knowledge about the users' keys. Suppose that there are  $i$  users,  $\text{user}_1, \text{user}_2, \dots, \text{user}_i$  in the system, and they use the key labeled as  $K_1, K_2, \dots, K_i$ , respectively. The assumption in the related-key model is that the attacker does not know the value of  $K_1, K_2, \dots, K_i$ , however, the relation of each key pair is known. The bit-wise XOR is a typical example of the relation in the related-key model. Namely, for two indices  $i_1, i_2 \in \{1, 2, \dots, i\}, i_1 \neq i_2$ , the values of  $K_{i_1}$  and  $K_{i_2}$  are unknown, but their relation  $K_{i_1} \oplus K_{i_2}$  is known.

#### 4.1.4 Complexity of Cryptanalysis

The efficiency of the cryptanalysis is usually measured by three types of quantity: **data**, **time**, and **memory**.

**Data:** The data quantity is measured by the number of queries.

**Time:** The time quantity is measured by computational cost executed by an attacker offline.

The unit of the computational cost is usually the cost of one execution of the encryption algorithm  $E_K$  or the decryption algorithm  $D_k$ . The computational cost for the oracle is not counted as the computational cost of the attack.

**Memory:** An attacker often needs to store queried plaintexts (resp. ciphertexts) and returned ciphertexts (resp. plaintexts) from the oracle. The attacker may need to store intermediate values generated during the attack. To store such data, a certain amount of memory is required. The memory quantity is measured by memory requirement. In many cases, the unit of the memory requirement is one value of the block size ( $b$  bits), or a byte (8 bits).

A triplet of those three types of quantity is called **attack complexity**. Let  $(D, T, M)$  be the data, time, and memory quantity of an attack against some security notion under some attack model. This indicates that if

1. the attacker can ask  $D$  queries to the oracle under the assumed attack model,
2. the attacker can spend the cost of executing the encryption or decryption algorithm  $T$  times, and
3. the attacker has enough memory to store  $M$  data of the size  $b$  bits,

the attacker succeeds in attacking the target cipher with respect to the target security notion in the target attack model.

#### 4.1.5 Generic Attacks

Because block ciphers use a fixed size key, it is information-theoretically impossible to keep the key secret permanently. There always exists an attack that recovers the key in a finite time. This principle is applied to any block cipher even if it is ideally designed. Such attacks are called **generic attacks**. In the following, two generic attacks against block ciphers are explained.

##### 4.1.5.1 Brute Force Attack (Exhaustive Search)

In the brute force attack, the attacker tries to recover the  $k$ -bit correct key value by simply examining all the  $2^k$  possibilities. The attack is often called an **exhaustive search**. The brute force attack starts by obtaining pairs of plaintext and ciphertext. The number of necessary pairs depends on the key size ( $k$  bits) and the block size ( $b$  bits). For simplicity, suppose that the block size is bigger than the key size, that is,  $k < b$ . In this case, the attack requires only one pair of a plaintext and the corresponding ciphertext. The attack consists of the online phase for collecting a pair of plaintext and ciphertext in the known plaintext attack and the offline phase for recovering the correct key value.

In the online phase, the attacker makes one query of a known plaintext  $P_1$  to the encryption oracle, and receives the corresponding ciphertext  $C_1$ . Then, with the knowledge of a valid pair  $(P_1, C_1)$ , the attacker aims to recover the correct key in the offline phase. Let  $G$  be a  $k$ -bit variable representing the key guess. The brute force attack computes the value of  $E_G(P_1)$  for all the  $2^k$  possibilities of  $G$ , and checks the match of the computed results and  $C_1$ . The detailed process of the brute force attack is shown in Algorithm 4.1.

If  $G$  is the correct guess, the equation  $\text{tmp} = C_1$  is satisfied with probability 1. If  $G$  is not the correct guess,  $\text{tmp} = C_1$  occurs with probability  $2^{-b}$ . When  $b$  is bigger than  $k$ , the event of the probability  $2^{-b}$  is unlikely to occur by  $2^k$  trials. Thus, Algorithm 4.1 can successfully return the correct key  $K$ . The brute force attack is illustrated in Figure 4.4. Because it works in the known plaintext attack, the attack can also work in the stronger attack models.

**Algorithm 4.1** Brute Force Attack for  $k < b$ 

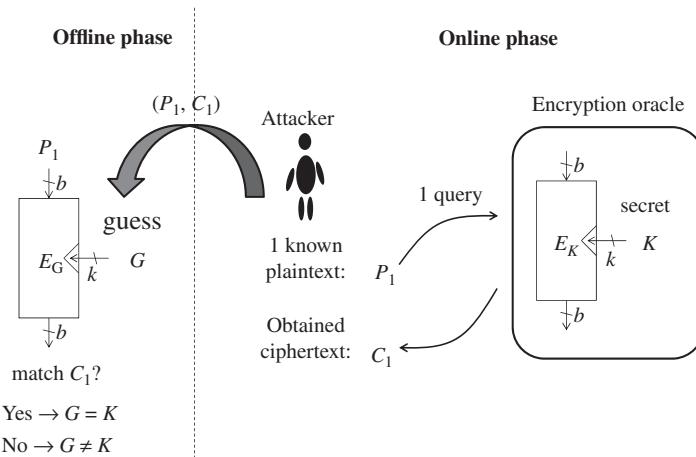
**Input:** A plaintext  $P_1$  and the corresponding ciphertext  $C_1$

**Output:** A secret key  $K$

```

1: for  $G \leftarrow 0, 1, \dots, 2^k - 1$  do
2:    $\text{tmp} \leftarrow E_G(P_1)$ ;
3:   if  $\text{tmp} = C_1$  then
4:     return  $G$ ;
5:   end if
6: end for

```



**Figure 4.4** Brute force attack for  $k < b$

When  $k < b$ , the data complexity of the brute force attack is one known plaintext. The time complexity of the brute force attack is  $2^k$  encryption operations. The brute force attack does not need to store any intermediate value during the attack but for a pair of a plaintext and a ciphertext  $(P_1, C_1)$ . Thus, the memory requirement of the brute force attack is negligible.

Let us consider the case when the key size is equal to the block size, that is,  $k = b$ . The attack algorithm basically follows Algorithm 4.1. However, the probability that a wrong guess satisfies  $\text{tmp} = C_1$  is not negligible. Owing to the assumption  $k = b$ ,  $2^b - 1$  wrong keys are examined. Then, an event with probability  $2^{-b}$  is satisfied after  $2^b - 1$  trials with the following number:

$$2^{-b} \cdot (2^b - 1) = 1 - 2^{-b}. \quad (4.3)$$

By assuming that the block size,  $b$ , is big enough,

$$1 - 2^{-b} \approx 1. \quad (4.4)$$

Therefore, besides the correct key  $K$ , one false positive (a value wrongly judged to be the correct key) is expected. If more than one key candidates remain, it is impossible to determine the correct key value only with one pair of  $(P_1, C_1)$ . Then, the attack needs to obtain

**Algorithm 4.2** Brute Force Attack for  $k > b$ 

**Input:**  $(P_1, C_1), (P_2, C_2), \dots, (P_r, C_r)$ , where  $r = \lceil \frac{k}{b} \rceil$

**Output:**  $K$

```

1: for  $G \leftarrow 0, 1, \dots, 2^k - 1$  do
2:    $\text{tmp} \leftarrow E_G(P_1);$ 
3:   if  $\text{tmp} = C_1$  then
4:      $\text{tmp}_i \leftarrow E_G(P_i)$  for all  $i \in \{2, 3, \dots, r\}$ ;
5:     if  $\text{tmp}_i = C_i$  for all  $i \in \{2, 3, \dots, r\}$  then
6:       return  $G;$ 
7:     end if
8:   end if
9: end for

```

another pair  $(P_2, C_2)$ , and also examines the match of  $E_G(P_2)$  and  $C_2$  for the remaining key candidates. The correct key guess can also satisfy the match of those two values, while the wrong key guess succeeds the test only with probability  $2^{-b}$ . In the end, the brute force attack successfully recovers the correct key  $K$  with at most two pairs of plaintexts and ciphertexts.

Finally, let us consider the case when the key size is much bigger than the block size, that is,  $k > b$ . The attack procedure is basically the same as the other cases but for the number of necessary pairs to detect the correct key  $K$ . Each pair of plaintext and ciphertext  $(P_i, C_i)$  yields a  $b$ -bit relation  $C_i = E_K(P_i)$ . This can be regarded as a  $b$ -bit filter for the key  $K$ , that is, the  $k$ -bit key space can be reduced to  $k - b$  bits by checking the match of a  $b$ -bit relation. Therefore, to reduce the key space to 1, the number of necessary pairs is described by

$$\left\lceil \frac{k}{b} \right\rceil. \quad (4.5)$$

The attack procedure is shown in Algorithm 4.2.

The procedure in Algorithm 4.2 first checks the match of  $E_G(P_1)$  and  $C_1$ , and then checks the match of the other pairs. By processing the attack in this order, the computational cost can be optimized to  $2^k$  encryption operations compared to the case that the match of all pairs is checked simultaneously.

**Exercise 4.1** Suppose that there is a block cipher whose key size is 256 bits and block size is 64 bits (in practice, the GOST block cipher adopts those parameters). Calculate the number of pairs of plaintexts and ciphertexts to recover a correct key  $K$  with the brute force attack against this block cipher.

**Exercise 4.2** Modify Algorithm 4.2 as shown in Algorithm 4.3. Compare the computational cost of the brute force attacks with Algorithms 4.2 and 4.3 against the block cipher with the parameters in the previous exercise.

**Algorithm 4.3** Modified Brute Force Attack for  $k > b$ 

**Input:**  $(P_1, C_1), (P_2, C_2), \dots, (P_r, C_r)$ , where  $r = \lceil \frac{k}{b} \rceil$

**Output:**  $K$

```

1: for  $G \leftarrow 0, 1, \dots, 2^k - 1$  do
2:    $\text{tmp}_i \leftarrow E_G(P_i)$  for all  $i \in \{1, 2, 3, \dots, r\}$ ;
3:   if  $\text{tmp}_i = C_i$  for all  $i \in \{1, 2, 3, \dots, r\}$  then
4:     return  $G$ ;
5:   end if
6: end for

```

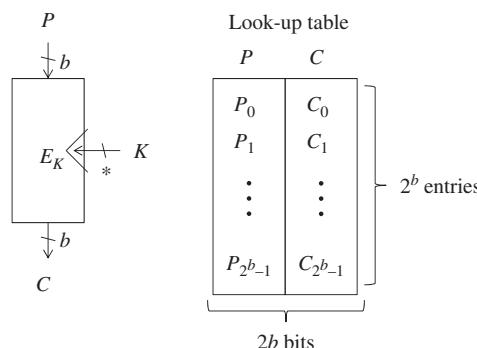
**4.1.5.2 Codebook Attack (Dictionary Attack)**

**Codebook** means the set of all plaintexts and the corresponding ciphertexts. For any newly given ciphertext, the attacker can obtain the corresponding plaintext by looking up the codebook. Thus, the plaintext recovery resistance is broken. The attack is also called a **dictionary attack**. The codebook attack is illustrated in Figure 4.5. The codebook attack breaks the plaintext recovery resistance irrespective of the key size. Note that even with the codebook, the attacker cannot obtain any information of the key  $K$  only by observing the correspondence of plaintexts and ciphertexts even if the key size  $k$  is smaller than the block size  $b$ .

The codebook attack requires to obtain all the  $2^b$  plaintext–ciphertext pairs. The attack can work under the known plaintext model. The data complexity of the codebook attack is  $2^b$  pairs of plaintexts and ciphertexts. The time complexity of the codebook attack is one access of look-up table, which is negligible. The attack requires to store all  $2^b$  plaintext–ciphertext pairs in a local memory as the look-up table. Thus, the memory requirement is  $2^b 2b$ -bit values.

**4.1.6 Goal of Shortcut Attacks (Cryptanalysis)**

For any block cipher, being attacked by generic attacks is not regarded to be a flaw of the design because it is theoretically impossible to prevent. The purpose of discussing generic attacks is that they can be used to know the goal of the design. That is to say, for secure block ciphers, any attack with a less cost than the one in the generic attack must be prevented.



**Figure 4.5** Codebook attack

Such attacks are called **shortcut attacks**. For a block cipher, allowing a shortcut attack is usually considered to be a significant vulnerability. In other words, the goal of the cryptanalysis is finding a shortcut attack, while the goal of the cipher's designer is preventing any shortcut attack.

Recall the two generic attacks explained earlier. The brute force attack indicates that with the attack complexity (Data, Time, Memory) =  $(\text{negl.}, 2^k, \text{negl.})$ , the key is recovered in any block cipher, where  $\text{negl.}$  represents “negligibly small.” The codebook attack indicates that with the attack complexity (Data, Time, Memory) =  $(2^b, \text{negl.}, 2^b)$ , the plaintext is recovered in any block cipher. Considering those facts, the number of queries for any shortcut attack must be smaller than  $2^b$ , and the computational cost for any shortcut attack must be smaller than  $2^k$ . The memory requirement of a valid shortcut attack is often bounded by  $2^k$ . In summary, the following lemma is obtained for a shortcut attack.

**Lemma 4.1.1** *The complexity of a shortcut attack must satisfy all of the following three conditions.*

$$\text{Data} < 2^b, \text{Time} < 2^k, \text{Memory} < 2^k. \quad (4.6)$$

In some cases, the attack with a complexity of  $\text{Data} = 2^b$ ,  $\text{Time} < 2^k$ , and  $\text{Memory} < 2^k$  is discussed. This is because there is no generic method to recover the key faster than the brute force attack even with the full codebook. This motivation is theoretically true, but in practice, the cipher's security has already been broken. The key recovery attack with the full codebook is less interesting than the key recovery attack without the full codebook.

From the next section, several popular approaches of the cryptanalysis against block ciphers are introduced.

## 4.2 Differential Cryptanalysis

### 4.2.1 Basic Concept and Definition

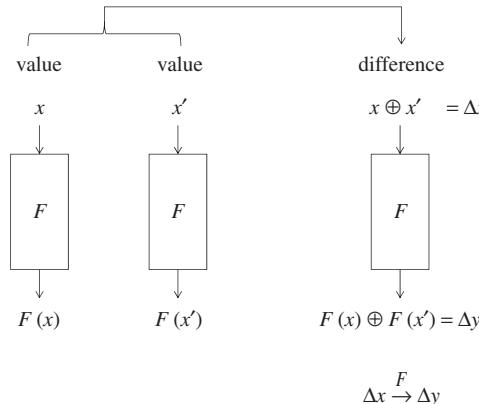
Differential cryptanalysis was established by Biham and Shamir. Suppose that two input messages  $M$  and  $M'$  are processed by the same function  $F$ . Namely,  $F(M)$  and  $F(M')$  are computed. The differential cryptanalysis focuses on the **difference** of two computations.

**Definition 4.2.1** *A difference of two values  $x$  and  $x'$  is represented as  $\Delta x$ , which is defined as an XOR of two values, that is,*

$$\Delta x \triangleq x \oplus x'. \quad (4.7)$$

Suppose that two values  $x$  and  $x'$  with a difference  $\Delta x$  are processed by the function  $F$ , and let  $\Delta y$  be the output difference. Then,  $\Delta y$  is computed as follows:

$$\Delta y = F(x) \oplus F(x'). \quad (4.8)$$

**Figure 4.6** Illustration of difference

**Definition 4.2.2** *The event that a pair of values with difference  $\Delta x$  changes to a pair of values with difference  $\Delta y$  through the function  $F$  is represented as*

$$\Delta x \xrightarrow{F} \Delta y. \quad (4.9)$$

*When the change of difference is focused without specifying the function, the following notation is used:*

$$\Delta x \rightarrow \Delta y. \quad (4.10)$$

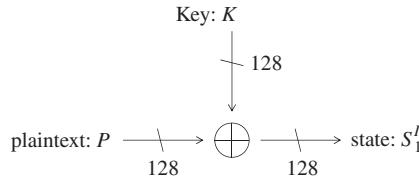
The concept of the difference is illustrated in Figure 4.6. The fact that the difference  $\Delta x$  becomes  $\Delta y$  after some operation is called “ $\Delta x$  is propagated to  $\Delta y$  (with some operation).”

### 4.2.2 Motivation of Differential Cryptanalysis

What for the differential cryptanalysis focuses on the difference of two computations rather than the result for each value? The reason is that, in many computations widely adopted in current block cipher algorithms, deterministic (probability 1) differential propagation can be constructed even without knowing the key value. This fact makes the cryptanalysis easy.

In many block ciphers, the key or values derived by the key is mixed to the plaintext or values derived by the plaintext with the XOR operation. Recall the specification of AES in Section 1.5. As the first operation, XOR of the plaintext and the key is taken, and the updated state  $S_1^I$  is computed. The operation is depicted in Figure 4.7.

Let us check how the differential cryptanalysis impact the block cipher analysis, by comparing it with the analysis focusing on a single value.



**Figure 4.7 Mixing key and plaintext in AES**

#### 4.2.2.1 Analysis with a Single Value

Suppose that the attacker obtains a single plaintext  $P$  in the known plaintext attack model. The attacker does not know the key value  $K$ . In AES, the plaintext is XORed with the key  $K$ , and the value of the state  $S_1^I$  is represented as  $P \oplus K$ . Because  $K$  is unknown to the attacker, the value of  $P \oplus K$  is unknown to the attacker. Thus, the attacker loses all information about the computation, and cannot continue the attack for the subsequent computations.

#### 4.2.2.2 Analysis with Difference of Two Values

Suppose that the attacker obtains two plaintexts  $P$  and  $P'$  in the known plaintext attack model. Then, the difference of the two plaintexts is represented as  $P \oplus P'$ . The attacker does not know the key value  $K$ . After taking XOR of the plaintext and the key, the state value  $S_1^I$  for the two plaintexts is represented by  $S_1^I = P \oplus K$  and  $S_1^{I'} = P' \oplus K$ . Then, the difference of the values of  $S_1^I$  for the two plaintexts is represented as

$$\Delta S_1^I = S_1^I \oplus S_1^{I'} \quad (4.11)$$

$$= (P \oplus K) \oplus (P' \oplus K) \quad (4.12)$$

$$= P \oplus P' \oplus K \oplus K \quad (4.13)$$

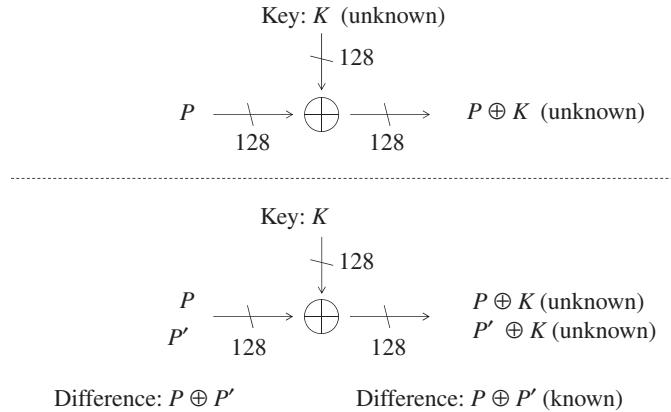
$$= P \oplus P'. \quad (4.14)$$

Although each of the value of  $P \oplus K$  and  $P' \oplus K$  is unknown, the attacker still knows their difference irrespective of the key value  $K$ . The above-mentioned two cases are depicted in Figure 4.8.

#### 4.2.3 Probability of Differential Propagation

For a given pair of values  $x$  and  $x'$  and a function  $F$  without any secret value, the corresponding difference after processing  $F$  can be computed deterministically, or the difference of  $F(x)$  and  $F(x')$  can be computed without any error probability.

The goal of the differential cryptanalysis is recovering the secret key  $K$  of a target block cipher  $E_K$ . The attacker usually cannot compute the difference of  $E_K(x)$  and  $E_K(x')$  without knowing  $K$ .



**Figure 4.8** Comparison of analysis with value and with difference

**Table 4.2** A 4-bit to 4-bit function  $S_{[4]}$

Input	$x$	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Output	$S_{[4]}(x)$	C	0	F	A	2	B	9	5	8	3	D	7	1	E	6	4

All values are described in the hexadecimal format.

To solve the issue, the differential cryptanalysis often specifies the difference of input values to a function  $E_K$ , **input difference**, and the difference of output values from  $E_K$ , **output difference**, without specifying the exact input value of  $x$ , and then evaluates the probability that the event  $\Delta x \xrightarrow{E_K} \Delta y$  is satisfied when  $x$  is randomly chosen according to the uniform distribution. This probability is represented by

$$\Pr[\Delta x \xrightarrow{E_K} \Delta y]. \quad (4.15)$$

As a case study, consider the differential propagation of a 4-bit to 4-bit function  $S_{[4]}$ , which is defined in Table 4.2. Note that  $S_{[4]}$  is the S-box computation adopted in a lightweight block cipher TWINE.

Consider calculating the probability that the input difference 5 changes into the output difference A, after the computation of  $S_{[4]}$ , that is,  $\Pr[5 \xrightarrow{S_{[4]}} A]$ . To calculate the probability, for all possible input values  $x \in \{0, 1, \dots, F\}$ , another input value is obtained by XORing the input difference,  $x \oplus 5$ , and the difference of  $S_{[4]}(x)$  and  $S_{[4]}(x \oplus 5)$  is computed. Then, the probability that the output difference is A is calculated. For  $x = 0$ , two input values are 0 and  $0 \oplus 5 = 5$ . The output difference is  $S_{[4]}(0) \oplus S_{[4]}(5) = C \oplus B = 7$ . The same evaluation is done for all  $x \in \{0, 1, \dots, F\}$ . The result is summarized in Table 4.3.

From Table 4.3, 2 out of 16 possibilities will result in the output difference A. Therefore,

$$\Pr[5 \xrightarrow{S_{[4]}} A] = 2/16 = 2^{-3}. \quad (4.16)$$

**Table 4.3** Output difference of  $S_{[4]}$  when input difference is 5

$x$	$x \oplus 5$	$S_{[4]}(x)$	$S_{[4]}(x \oplus 5)$	$S_{[4]}(x) \oplus S_{[4]}(x \oplus 5)$
0	5	C	B	7
1	4	0	2	2
2	7	F	5	A
3	6	A	9	3
4	1	2	0	2
5	0	B	C	7
6	3	9	A	3
7	2	5	F	A
8	D	8	E	6
9	C	3	1	2
A	F	D	4	9
B	E	7	6	1
C	9	1	3	2
D	8	E	8	6
E	B	6	7	1
F	A	4	D	9

Similarly, the probability of the differential propagation from 5 to other output differences can be obtained from Table 4.3.

$$\Pr[5 \xrightarrow{S_{[4]}} 0] = 0/16 = 0 \quad (4.17)$$

$$\Pr[5 \xrightarrow{S_{[4]}} 1] = 2/16 = 2^{-3} \quad (4.18)$$

$$\Pr[5 \xrightarrow{S_{[4]}} 2] = 4/16 = 2^{-2} \quad (4.19)$$

$$\Pr[5 \xrightarrow{S_{[4]}} 3] = 2/16 = 2^{-3} \quad (4.20)$$

$$\Pr[5 \xrightarrow{S_{[4]}} 4] = 0/16 = 0 \quad (4.21)$$

$$\Pr[5 \xrightarrow{S_{[4]}} 5] = 0/16 = 0 \quad (4.22)$$

$$\Pr[5 \xrightarrow{S_{[4]}} 6] = 2/16 = 2^{-3} \quad (4.23)$$

$$\Pr[5 \xrightarrow{S_{[4]}} 7] = 2/16 = 2^{-3} \quad (4.24)$$

$$\Pr[5 \xrightarrow{S_{[4]}} 8] = 0/16 = 0 \quad (4.25)$$

$$\Pr[5 \xrightarrow{S_{[4]}} 9] = 2/16 = 2^{-3} \quad (4.26)$$

$$\Pr[5 \xrightarrow{S_{[4]}} A] = 2/16 = 2^{-3} \quad (4.27)$$

$$\Pr[5 \xrightarrow{S_{[4]}} B] = 0/16 = 0 \quad (4.28)$$

$$\Pr[5 \xrightarrow{S_{[4]}} C] = 0/16 = 0 \quad (4.29)$$

$$\Pr[5 \xrightarrow{S_{[4]}} D] = 0/16 = 0 \quad (4.30)$$

$$\Pr[5 \xrightarrow{S_{[4]}} E] = 0/16 = 0 \quad (4.31)$$

$$\Pr[5 \xrightarrow{S_{[4]}} F] = 0/16 = 0. \quad (4.32)$$

**Exercise 4.3** As Equations (4.17)–(4.32), compute the probability of the differential propagation from  $C$  to other output differences.

The probability of the differential propagation is formally defined as follows.

**Definition 4.2.3 (Probability of Differential Propagation from  $\Delta x$  to  $\Delta y$  through  $F$ )**  
*Let  $F$  be a function that is either unknown or known to the attacker. Let  $z$  be the bit-length of input value to the function  $F$ . The probability that a fixed input difference  $\Delta x$  changes to a fixed output difference  $\Delta y$  through the function  $F$  is represented as*

$$\Pr[\Delta x \xrightarrow{F} \Delta y] \triangleq \frac{\#\{x | F(x) \oplus F(x \oplus \Delta x) = \Delta y\}}{2^z}. \quad (4.33)$$

#### 4.2.4 Deterministic Differential Propagation in Linear Computations

The case study in Figure 4.8 can be generalized more. Let  $F_{\text{xor}} : \{0, 1\}^n \times \{0, 1\}^n \rightarrow \{0, 1\}^n$  be a function that returns XOR of two  $n$ -bit input values. Then, XOR of a plaintext and the fixed key  $K$  can be represented as  $F_{\text{xor}}(\cdot, K)$ . As shown in Equation (4.14), for any value of  $P$  and  $P'$ , the input difference to  $F_{\text{xor}}(\cdot, K)$  and the output difference from  $F_{\text{xor}}(\cdot, K)$  become identical. That is to say, the following lemma holds for  $F_{\text{xor}}(\cdot, K)$ .

**Lemma 4.2.4** For any difference  $\Delta x$  and constant  $t$ ,

$$\Pr[\Delta x \xrightarrow{F_{\text{xor}}(\cdot, t)} \Delta x] = 1. \quad (4.34)$$

Moreover, the probability that a difference other than  $\Delta x$  is output is 0. Namely, the following lemma holds.

**Lemma 4.2.5** For any difference  $\Delta x, \Delta y$ , and constant  $t$  such that  $\Delta x \neq \Delta y$ ,

$$\Pr[\Delta x \xrightarrow{F_{\text{xor}}(\cdot, t)} \Delta y] = 0. \quad (4.35)$$

More generally, suppose that two values for the first input variable to  $F_{\text{xor}}$  have a difference  $\Delta x$  and two values for the second input variable have a difference  $\Delta x'$ . Then, the following lemma holds for  $F_{\text{xor}}$ .

**Lemma 4.2.6** *For any difference  $\Delta x$  and  $\Delta x'$ ,*

$$\Pr[(\Delta x, \Delta x') \xrightarrow{F_{\text{xor}}} \Delta x \oplus \Delta x'] = 1. \quad (4.36)$$

Note that Lemma 4.2.4 is the special case of Lemma 4.2.6, in which the difference of the second input variable is 0.

An interesting observation in Equation (4.36) is that

$$\Delta x \oplus \Delta x' = F_{\text{xor}}(\Delta x, \Delta x'). \quad (4.37)$$

That is to say, the output difference can be computed as if the difference were the input value to the function  $F_{\text{xor}}$ . In the differential cryptanalysis, an attacker aims to obtain a high probability differential propagation only with determining differences without determining the paired values. The approach is useful for block cipher analysis because the plaintext values become unknown quickly owing to the key.

Not only for the XOR operation but also for any type of linear computations, the differential propagation can be computed deterministically without determining the exact paired values. A linear function  $L$  satisfies the following property:

$$L(a) \oplus L(b) = L(a \oplus b). \quad (4.38)$$

Therefore, the difference of two values  $a$  and  $b$  after a linear computation  $L$  can be computed by taking their difference as input to  $L$ . Namely, the following lemma holds, which is illustrated in Figure 4.9.

**Lemma 4.2.7** *For any input difference  $\Delta x$  and a linear computation  $L$ ,*

$$\Pr[\Delta x \xrightarrow{L} L(\Delta x)] = 1. \quad (4.39)$$

*For any difference  $\Delta y$  such that  $\Delta y \neq L(\Delta x)$ ,*

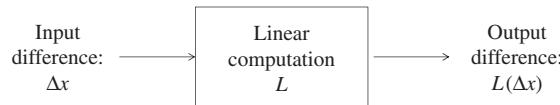
$$\Pr[\Delta x \xrightarrow{L} \Delta y] = 0. \quad (4.40)$$

Other two examples of the computation that are widely adopted in current block cipher algorithms are the rotation operation and the multiplication over a finite field.

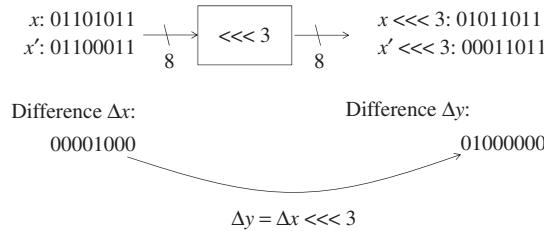
**Rotation:** Let “ $\lll r$ ” be the left rotation by  $r$ -bits. Then, for any input difference  $\Delta x$ ,

$$\Pr[\Delta x \xrightarrow{\lll r} (\Delta x \lll r)] = 1. \quad (4.41)$$

The differential propagation of the 3-bit rotation for 8-bit values is depicted in Figure 4.10.  $\Delta y = \Delta x \lll 3$  holds for any choices of  $x$  and  $x'$ .



**Figure 4.9** Computing output difference in a linear computation  $L$



**Figure 4.10** Differential propagation over a rotation operation

**Multiplication over a finite field:** Let “ $\cdot t$ ” be the multiplication with a constant  $t$  over a finite field. Then, for any input difference  $\Delta x$ ,

$$\Pr[\Delta x \xrightarrow{\cdot t} (\Delta x \cdot t)] = 1. \quad (4.42)$$

A composition of more than one linear computations are also linear. Hence, for any composition of linear computations, the output difference can be computed deterministically from the input difference. An example is the MixColumns operation in AES. The MixColumns operation linearly maps a 32-bit value to another 32-bit value, whose computation consists of the multiplications over a finite field and XOR of several values. Thus, if the input difference to the MixColumns operation is determined, the corresponding output difference can be deterministically computed without determining the exact values.

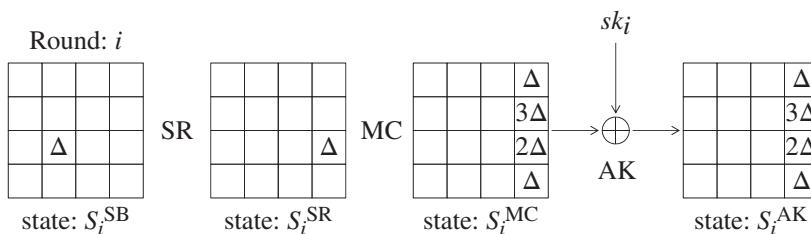
As an application to AES, the deterministic differential propagation for the linear computation part of the AES round function is explained below, which is described in Figure 4.11.

The linear operation part of the AES round function in round  $i$  is ShiftRows, MixColumns, and AddRoundKey. Suppose that difference of the input state to the ShiftRows operation  $\Delta S_i^{\text{SB}}$  has a nonzero byte difference  $\Delta$  in the third row of the second column. Hereafter, a sequence of 16-byte values is used to represent a 128-bit value of the state. More precisely, the state whose  $j$ th byte value is  $v_j$  for  $j = 0, 1, \dots, 15$  is denoted by

$$(v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}, v_{11}, v_{12}, v_{13}, v_{14}, v_{15}). \quad (4.43)$$

With this notation, the above  $\Delta S_i^{\text{SB}}$  is represented as

$$\Delta S_i^{\text{SB}} = (0, 0, 0, 0, 0, 0, \Delta, 0, 0, 0, 0, 0, 0, 0, 0, 0). \quad (4.44)$$



**Figure 4.11** Differential propagation for linear operations of AES round function

After the ShiftRows operation, each byte in the third row is rotated by 2 bytes to left. Thus, in the difference of the state  $S_i^{\text{SR}}$ , a nonzero byte difference  $\Delta$  moves to the third row in the fourth column, that is,

$$\Delta S_i^{\text{SR}} = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \Delta, 0). \quad (4.45)$$

Then, the MixColumns operation is applied. Regarding the first, second, and third columns, the output difference is computed as

$$\begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}. \quad (4.46)$$

Therefore, the differences for those columns are 0. Regarding the fourth column, the output difference is computed as

$$\begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ \Delta \\ 0 \end{bmatrix} = \begin{bmatrix} \Delta \\ 3\Delta \\ 2\Delta \\ \Delta \end{bmatrix}. \quad (4.47)$$

In the end, after the MixColumns operation, the difference of the state value  $S_i^{\text{MC}}$  becomes as follows:

$$\Delta S_i^{\text{MC}} = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \Delta, 3\Delta, 2\Delta, \Delta). \quad (4.48)$$

**Exercise 4.4** Suppose that the input state to the MixColumns operation has the difference  $(0, 0, x, 0, 0, y, 0, 0, z, 0, 0, 0, 0, 0, 0, w)$ , where each of  $x, y, z, w$  is a nonzero byte value. Compute the corresponding output difference after the MixColumns operation.

Finally, the AddRoundKey operation is applied, which computes XOR of the state and the subkey. From Lemma 4.2.4, the output difference of the XOR operation is the same as the input difference. Thus, the difference after the AddRoundKey operation is represented as follows:

$$\Delta S_i^{\text{AK}} = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \Delta, 3\Delta, 2\Delta, \Delta), \quad (4.49)$$

which concludes the differential propagation for the linear computation part of the AES round function.

#### 4.2.5 Probabilistic Differential Propagation in Nonlinear Computations

When a function  $F$  does not satisfy the definition of linear operation, that is,

$$\exists(a, b) \text{ such that } F(a) \oplus F(b) \neq F(a \oplus b), \quad (4.50)$$

**Table 4.4** Differential distribution table of  $S_{[4]}$ 

		Output difference															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	0	16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
	1	0	0	2	0	0	2	0	0	0	2	2	2	4	0	0	
	2	0	0	0	2	2	2	0	2	0	0	4	2	0	0	2	
	3	0	0	2	0	0	2	2	2	2	0	0	0	0	0	2	
	4	0	0	0	2	0	0	2	0	0	2	0	4	0	2	2	
	5	0	2	4	2	0	0	2	2	0	2	2	0	0	0	0	
	6	0	2	0	0	0	4	0	2	0	2	0	0	2	2	2	
Input difference	7	0	0	0	2	2	2	2	0	2	4	0	0	2	0	0	
	8	0	2	2	4	2	2	0	0	0	0	0	0	0	2	0	
	9	0	0	0	2	0	0	0	2	4	0	2	0	2	2	0	
	A	0	2	0	0	2	0	0	4	2	2	0	2	0	0	2	
	B	0	0	2	0	2	0	2	2	0	0	0	2	2	4	0	
	C	0	0	2	0	2	0	0	0	2	2	2	0	0	2	4	
	D	0	4	2	2	0	0	0	0	2	0	0	2	2	0	2	
	E	0	2	0	0	4	0	2	0	0	0	2	0	2	0	2	
	F	0	2	0	0	0	2	4	0	2	0	2	2	0	2	0	

no efficient method is known to derive a high probability differential propagation. Hence, all the possibilities must be taken into consideration according to Equation (4.33) in the definition of the probability of differential propagation.

AES adopts a single nonlinear operation in a round function, which is the SubBytes operation. SubBytes applies an S-box to each byte of the state in order to substitute a byte (8-bit) value into another one according to a prespecified conversion table. Analyzing 8-bit to 8-bit S-box is complicated. Hence, in this section, the 4-bit to 4-bit S-box  $S_{[4]}$  defined in Table 4.2 is mainly analyzed as a small example, and the extension to the 8-bit to 8-bit S-box is explained later.

For a nonlinear function, the attacker aims to obtain the probability of the differential propagation for all possible patterns. Most of the block ciphers adopt a small size S-box, for example, a 4-bit S-box or an 8-bit S-box, in order to avoid inefficiency when it is implemented. Hence, computing all the possibilities is usually feasible. In detail, the attacker computes Equation (4.33) for all the choices of the input difference  $\Delta IN$  and the output difference  $\Delta OUT$ . Equations (4.17)–(4.32) show the probabilities of the differential propagation when  $\Delta IN$  is fixed to 5 and  $\Delta OUT$  is unfixed. To complete it, the analysis is iterated by changing  $\Delta IN$  for all the possibilities.

Owing to the redundancy, the detailed analysis is omitted. The result is shown in Table 4.4. The row with  $\Delta IN = 5$  corresponds to Equations (4.17)–(4.32). Table 4.4 is called a **differential distribution table**. Hereafter, the differential distribution table is denoted by **DDT**. Each entry of the DDT represents the number of values that satisfies

$$S_{[4]}(x) \oplus S_{[4]}(x \oplus \Delta IN) = \Delta OUT \quad (4.51)$$

of the corresponding  $\Delta IN$  and  $\Delta OUT$ . The probability of the differential propagation is obtained by dividing the number in each entry with the number of possible values of  $x$ , that

is,  $2^4$ . This indicates that the differential propagation whose entry is 4 occurs with a higher probability than the others.

The DDT of  $S_{[4]}$  has several properties.

**Each entry is even:** When  $x = a$ , where  $a \in \{0, 1, \dots, 15\}$ , satisfies the differential propagation in Equation (4.51),  $x = (a \oplus \Delta IN)$  always satisfies Equation (4.51). Hence, each entry of DDT is always an even number.

**Deterministic propagation for zero difference:**  $\Delta IN = 0$  means that the pair has an identical value. Thus, the output value is also identical. When  $\Delta IN = 0$ ,  $\Delta OUT$  is 0 with probability 1.

**Minimizing differential probability:** To avoid a high probability differential propagation, each entry of DDT should be as small as possible. Combining the above-mentioned two properties, the best design strategy of  $S_{[4]}$  against the differential cryptanalysis is having one entry with 4 and six entries with 2 in all the rows and columns.

The S-box used in AES has the similar property to  $S_{[4]}$ . Because the input and the output size of the AES S-box is 8 bits, or 256, the DDT of the AES S-box has 256 rows and 256 columns. Owing to its large size, the exact DDT is omitted in this book. Two important properties are summarized as follows.

**Lemma 4.2.8** *For all  $\Delta IN$  but for  $\Delta IN = 0$ :*

- *there is 1  $\Delta OUT$  whose entry is 4;*
- *there are 126  $\Delta OUT$  whose entry is 2;*
- *there are 129  $\Delta OUT$  whose entry is 0.*

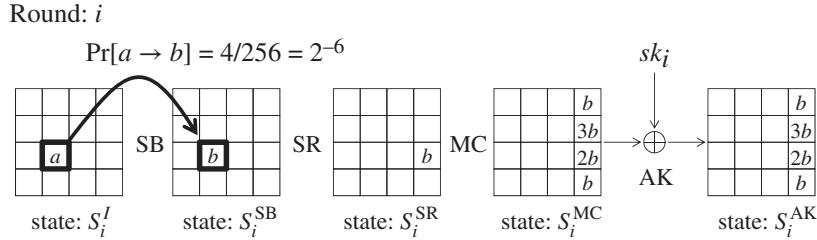
**Lemma 4.2.9** *For all  $\Delta OUT$  but for  $\Delta OUT = 0$ :*

- *there is 1  $\Delta IN$  whose entry is 4;*
- *there are 126  $\Delta IN$  whose entry is 2;*
- *there are 129  $\Delta IN$  whose entry is 0.*

**Exercise 4.5** *Make a DDT of the AES S-box by writing a program. Then, confirm Lemmas 4.2.8 and 4.2.9.*

**Exercise 4.6** *Explain the reason why the 8-bit S-box whose DDT has any nonzero element is 2 cannot be constructed.*

As an application to AES, the probability of the differential propagation for one round is explained below, which is described in Figure 4.12.



**Figure 4.12** Differential propagation for AES one round

The analysis starts from two input values to the SubBytes operation in round  $i$  denoted by state  $S_i^I$ . The input difference  $\Delta S_i^I$  is set to have a nonzero byte difference  $a$  in the third row of the second column, that is,

$$\Delta S_i^I = (0, 0, 0, 0, 0, 0, a, 0, 0, 0, 0, 0, 0, 0, 0). \quad (4.52)$$

Then, the S-box is applied to all bytes in the state by the SubBytes operation. From Lemma 4.2.8, for any nonzero input difference, there exists one output difference that will be produced with probability  $4/256 = 2^{-6}$ . Denote the output difference by  $b$ . Thus, the difference of the output state  $S_i^{SB}$  is

$$\Delta S_i^{SB} = (0, 0, 0, 0, 0, 0, b, 0, 0, 0, 0, 0, 0, 0, 0), \quad (4.53)$$

and the probability of the differential propagation is as follows:

$$\Pr[\Delta S_i^I \xrightarrow{\text{SB}} \Delta S_i^{SB}] = 2^{-6}. \quad (4.54)$$

The propagation from  $\Delta S_i^{SB}$  to  $\Delta S_i^{AK}$  is the same as the one in Figure 4.11. Then, with probability 1, the difference of the state  $S_i^{AK}$  becomes as follows:

$$\Delta S_i^{AK} = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, b, 3b, 2b, b). \quad (4.55)$$

In the end, the following differential propagation with probability  $2^{-6}$  is obtained for AES one round.

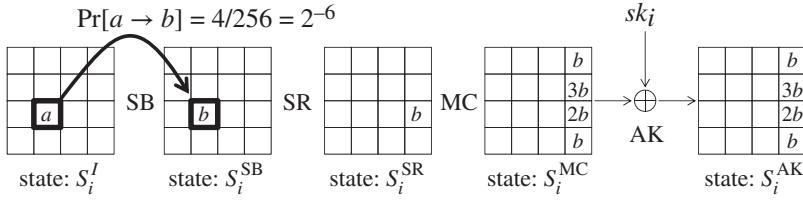
$$\Pr[\Delta S_i^I \xrightarrow{\text{AES 1R}} \Delta S_i^{AK}] = 2^{-6}. \quad (4.56)$$

#### 4.2.6 Probability of Differential Propagation for Multiple Rounds

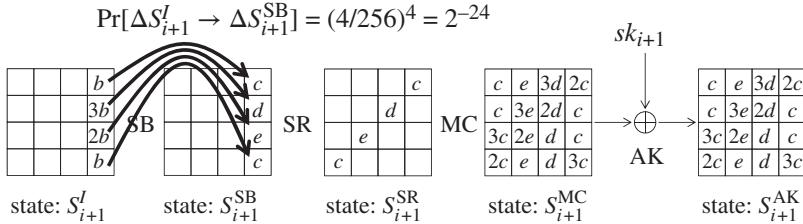
Equation (4.33) in the definition of the probability of the differential propagation can be computed only if the computation size is small, that is, the value of  $z$  in Equation (4.33) is small. Namely, it can be evaluated for the 4-bit to 4-bit mapping  $S_{[4]}$  or the 8-bit to 8-bit mapping of the AES S-box  $S$ , while it cannot be evaluated for a 128-bit to 128-bit mapping of the entire AES operation. The goal of the attacker is attacking as many rounds as possible, thus the analysis for a larger map is necessary.

When a differential propagation for multiple rounds is evaluated, the differential cryptanalysis iteratively applies the analysis for a single round by assuming that the probability of the

Round:  $i$



Round:  $i + 1$



**Figure 4.13** Differential propagation for AES two rounds

differential propagation can be evaluated independently for each round. The validity of this assumption comes from the fact that the internal state value is XORed by an unknown subkey in every round. Suppose that all subkeys are chosen accordingly to the uniform distribution. Then, between two SubBytes operations, the internal state value becomes uniformly random owing to the subkey XOR, which makes the probability evaluation of the two SubBytes operations independent. Note that the key schedule function of AES is not so strong that  $sk_i$  and  $sk_{i+1}$  can be completely random. In this sense, the assumption of the independent probability evaluation in different rounds is not true. However, to make the discussion simple, it is usually assumed that each subkey is chosen uniformly randomly. Hereafter, the same assumption is applied in this book.

An example analysis for AES reduced to two rounds (for round  $i$  and round  $i + 1$ ) is illustrated in Figure 4.13. The analysis for round  $i$  is the same as Figure 4.12, and another round is appended after the last state in round  $i$ .

By following the assumption of the independence in different rounds, the probability of the differential propagation in round  $i + 1$  is evaluated independently from the one in round  $i$ . For each nonzero input difference to the S-box,  $b$ ,  $3b$ , and  $2b$ , Lemma 4.2.8 ensures that there exists one output difference that will be produced with probability  $4/256 = 2^{-6}$ . Let  $c$ ,  $d$ , and  $e$  be those output differences from the S-box. There are 4 bytes with a nonzero difference, and the probability that the propagation is achieved in all bytes simultaneously is obtained by multiplying the probability of each event. Thus,

$$\Pr[\Delta S_{i+1}^I \xrightarrow{\text{SB}} \Delta S_{i+1}^{\text{SB}}] = (4/256)^4 = 2^{-24}. \quad (4.57)$$

After the SubBytes operation, the remaining is the linear computation, and thus the differential propagation is determined with probability 1. The probability of the differential

propagation in round  $i + 1$  shown in Figure 4.13 is concluded as:

$$\Pr[\Delta S_{i+1}^I \xrightarrow{\text{AES IR}} \Delta S_{i+1}^{\text{AK}}] = 2^{-24}. \quad (4.58)$$

Finally, the probability of the differential propagation for two rounds is obtained by multiplying the probability for round  $i$  and for round  $i + 1$ . Thus,

$$\Pr[\Delta S_i^I \xrightarrow{\text{AES 2R}} \Delta S_{i+1}^{\text{AK}}] = 2^{-6} \times 2^{-24} = 2^{-30}. \quad (4.59)$$

If the assumption of independence between different rounds is not used, the probability for round  $i + 1$  must be a conditional probability that the input value is the output of round  $i$ , that is,

$$\Pr[\Delta S_i^I \xrightarrow{\text{AES IR}} \Delta S_i^{\text{AK}} | \Delta S_{i+1}^I \xrightarrow{\text{AES IR}} \Delta S_{i+1}^{\text{AK}}]. \quad (4.60)$$

Evaluating the conditional probability especially for many rounds is hard (computationally infeasible). To make the analysis simple and effective, the differential cryptanalysis usually assumes the independence of differential propagations in different rounds.

#### 4.2.7 Differential Characteristic for AES Reduced to Three Rounds

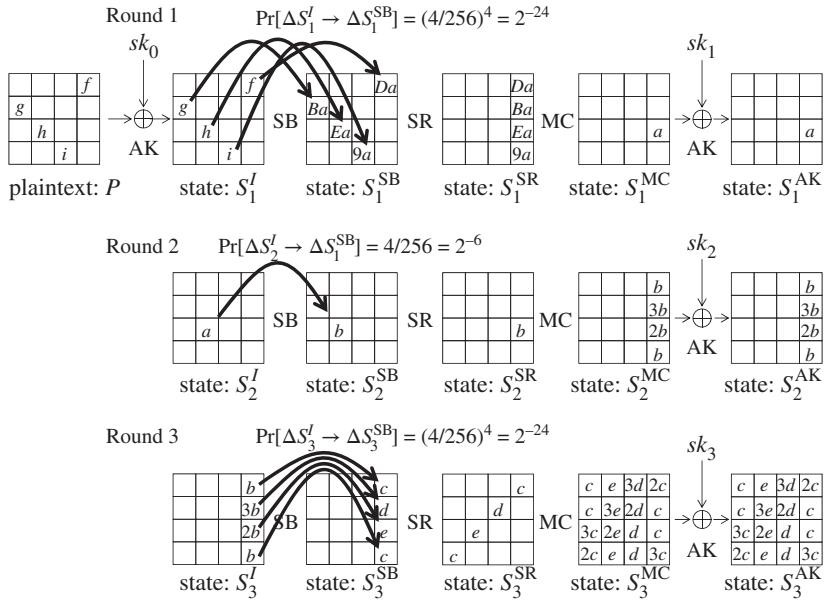
In the differential cryptanalysis, the attacker first chooses a pair of an input difference and an output difference that will be propagated with a high probability. How the difference will propagate in the attack between the chosen differences is called **differential characteristic**.

As explained so far, the probability of the differential characteristic decreases only by the SubBytes operation. A byte with nonzero difference through the SubBytes operation is called an **active byte with respect to the difference**, or simply an **active byte**. The attack strategy is choosing active-byte positions so that the number of active bytes is minimized.

Regarding AES, a differential characteristic with a relatively high probability,  $2^{-54}$ , can be constructed up to three rounds, which is described in Figure 4.14. Its details are explained below. AES adopts different round functions for the middle rounds and the last round. In the differential cryptanalysis, the attacker later appends several rounds after the differential characteristic in order to recover the key. When the differential characteristic is constructed, only the round function for the middle rounds is considered.

The last two rounds of the differential characteristic in Figure 4.14 are the same as the ones in Figure 4.13. The analysis starts from the single active byte at state  $S_2^I$  whose difference is denoted by  $a$ . This is propagated to  $b$  after the SubBytes operation with a probability  $2^{-6}$ . After the MixColumns operation in round 2, a single active byte expands to 4 active bytes, and then those derive four differential propagation with probability  $2^{-6}$  for the SubBytes operation in round 3. The remaining computations in round 3 are linear computations, and thus the attacker obtains the corresponding difference at state  $S_3^{\text{AK}}$  with probability 1.

Then, another round is appended to those two rounds. If one more round is appended in the end of the two rounds, that is, as round 4, the number of active bytes will increase by 16 bytes, which is inefficient. Therefore, another round is appended to the beginning of those two rounds, that is, as round 1. The attacker computes the propagation of the difference  $\Delta S_2^I$  in the backward direction. The AddRoundKey operation simply computes XOR of the subkey



**Figure 4.14** Differential characteristic for AES three rounds with probability  $2^{-24}$

$sk_1$  and the state value  $S_2^I$ . As shown in Lemma 4.2.4, the difference never changes with the constant XOR, which derives the following difference in state  $S_1^{MC}$ .

$$\Delta S_1^{MC} = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, a, 0). \quad (4.61)$$

The inverse MixColumns operation is then applied. No difference exists for the first, second, and the third columns. Thus, the difference after the inverse MixColumns operation is also 0. For the fourth column, the difference is computed as follows.

$$\begin{bmatrix} E & B & D & 9 \\ 9 & E & B & D \\ D & 9 & E & B \\ B & D & 9 & E \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ a \\ 0 \end{bmatrix} = \begin{bmatrix} Da \\ Ba \\ Ea \\ 9a \end{bmatrix}. \quad (4.62)$$

As a result, the difference of the state  $S_1^{SR}$  becomes as

$$\Delta S_1^{SR} = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, Da, Ba, Ea, 9a). \quad (4.63)$$

In the inverse ShiftRows operation, each byte in row  $j$ , where  $j \in \{0, 1, 2, 3\}$ , is rotated by  $j$  bytes to the right. This makes the difference of the state  $S_1^{SB}$  as

$$\Delta S_1^{SB} = (0, Ba, 0, 0, 0, 0, Ea, 0, 0, 0, 0, 9a, Da, 0, 0, 0). \quad (4.64)$$

From Lemma 4.2.9, for any nonzero output difference of the S-box, there exists one input difference that is achieved with probability  $4/256 = 2^{-6}$ . Let  $f, g, h$ , and  $i$  be the input

differences to the S-box such that

$$\Pr[f \xrightarrow{S} \text{Da}] = 2^{-6}, \quad (4.65)$$

$$\Pr[g \xrightarrow{S} \text{Ba}] = 2^{-6}, \quad (4.66)$$

$$\Pr[h \xrightarrow{S} \text{Ea}] = 2^{-6}, \quad (4.67)$$

$$\Pr[i \xrightarrow{S} 9a] = 2^{-6}. \quad (4.68)$$

This makes the difference of the state  $S_1^I$  as

$$\Delta S_1^I = (0, g, 0, 0, 0, 0, h, 0, 0, 0, 0, i, f, 0, 0, 0). \quad (4.69)$$

Finally, another AddRoundKey operation is inverted, which does not affect the differential propagation. The plaintext difference  $\Delta P$  is the same as  $\Delta S_1^I$ , that is,

$$\Delta P = (0, g, 0, 0, 0, 0, h, 0, 0, 0, 0, i, f, 0, 0, 0). \quad (4.70)$$

As explained in the previous section, the total probability of the three-round differential characteristic in Figure 4.14 is the multiplication of the probability of each round evaluated independently, which is

$$(2^{-6})^4 \times 2^{-6} \times (2^{-6})^4 = 2^{-54}. \quad (4.71)$$

**Exercise 4.7** How many differential characteristics with probability  $2^{-54}$  following the same active-byte positions as Figure 4.14 exist for AES three rounds?

**Exercise 4.8** How many differential characteristics with probability  $2^{-55}$  exist for AES three rounds?

#### 4.2.8 Distinguishing Attack with Differential Characteristic

The three-round differential characteristic in Figure 4.14 can directly be used to attack the security notion of indistinguishability against AES reduced to three rounds. The attack is called a **distinguishing attack**. For simplicity, suppose that the attack model is the chosen plaintext attack, in which the attacker can query the plaintext of his choice to observe the corresponding ciphertext.

Recall Figure 4.2 that shows the indistinguishability framework. The attacker (observer) interacts with a 128-bit permutation that is either AES reduced to three rounds with an unknown key or a 128-bit random permutation. By observing plaintexts and ciphertexts,

---

**Algorithm 4.4** Distinguishing Attack against AES Reduced to 3 Rounds

---

**Input:** A differential characteristic propagating from  $\Delta P$  to  $\Delta S_3^{\text{AK}}$  with probability  $2^{-54}$

**Output:** A determining bit  $B \in \{0, 1\}$

```

1: Choose  $2^{54}$  distinct plaintexts  $P_i$  for  $i = 1, 2, \dots, 2^{54}$ ;
2: for  $i \leftarrow 1, 2, \dots, 2^{54}$  do
3:   Query  $P_i$  to the encryption oracle and obtain the corresponding ciphertext  $C_i$ ;
4:   Query  $P'_i = P_i \oplus \Delta P$  to the encryption oracle and obtain the corresponding ciphertext
    $C'_i$ ;
5:   if  $C_i \oplus C'_i = \Delta S_3^{\text{AK}}$  then
6:     return 0;           // The oracle is the AES reduced to 3 rounds.
7:   end if
8: end for
9: return 1;           // The oracle is a random permutation.
```

---

the attacker aims to distinguish which is implemented in the oracle. The output of the distinguisher is a single bit called a **determining bit**. If the attacker finds that AES reduced to three rounds is implemented, the distinguishing attack outputs a single bit 0 as the determining bit. If the attacker finds that a random permutation is implemented, the distinguishing attack outputs a single bit 1 as the determining bit.

In the attack procedure, the attacker queries  $2^{54}$  plaintext pairs with a difference  $\Delta P$  in Figure 4.14, and then checks whether or not one of the corresponding ciphertext pairs has a difference  $\Delta S_3^{\text{AK}}$  in Figure 4.14.

- If the oracle implements AES reduced to three rounds, one of the pairs is expected to satisfy the differential characteristic with probability  $2^{-54}$ .
- If the oracle implements a random permutation, the probability that the ciphertext difference becomes  $\Delta S_3^{\text{AK}}$  is  $2^{-128}$  per pair. With a negligible probability, an event with probability  $2^{-128}$  is satisfied by examining  $2^{54}$  pairs.

Therefore, if one of the ciphertext pairs has the difference  $\Delta S_3^{\text{AK}}$ , the attack returns 0 as the determining bit. Otherwise, it returns 1 as the determining bit. In Algorithm 4.4, the attack procedure is given in an algorithmic format.

The attack complexity of the distinguishing attack is evaluated as follows.

- The data complexity is  $2 \times 2^{54} = 2^{55}$  chosen plaintexts.
- The time complexity is  $2^{54}$  XOR computational cost.
- The attack does not need to store intermediate values during the attack but for 1 ciphertext  $C_i$ . Thus, the memory complexity is 1, or negligible.

Note that when the event with probability  $2^{-54}$  is examined by  $2^{54}$  trials, the expectation number of the success is 1, but it does not ensure the success with probability 1. The success probability of the distinguishing attack can increase by spending more data and time complexities.

**Exercise 4.9** Compute the success probability of the above distinguishing attack with  $2^{54}$  pairs of plaintexts.

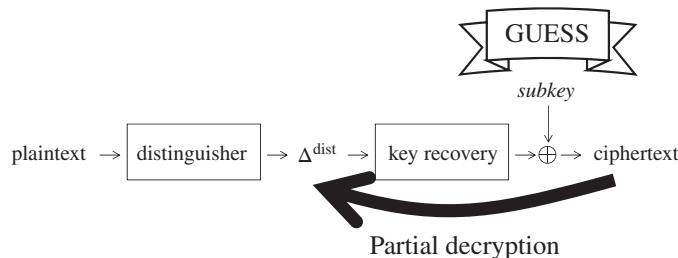
**Exercise 4.10** In order to increase the success probability of the above distinguishing attack to 95%, how many plaintexts are required?

#### 4.2.9 Key Recovery Attack after Differential Characteristic

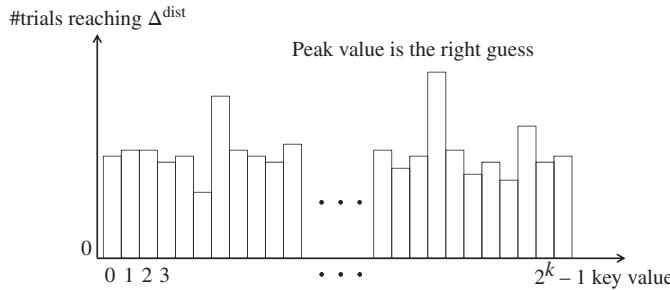
Although the security of block cipher can be broken by the distinguishing attack, the attacker cannot recover the key value with the distinguishing attack. The differential cryptanalysis can also be used to recover the key. In general, compared to distinguishing attacks, key recovery attacks can attack more rounds while the attack complexity is higher.

The attacker appends a few more rounds to the end of the distinguisher, and the attacker aims to recover the last subkey value. The framework of the key recovery attack is illustrated in Figure 4.15. The key recovery attack starts from collecting several pairs of plaintexts satisfying the differential characteristic. Because a few more rounds are appended to the end of the distinguisher, the attacker cannot identify the pairs satisfying the differential characteristic. Hence, all the pairs are analyzed to recover the key. The pairs satisfying the differential characteristic are called **right pairs**, while the pairs not satisfying the differential characteristic are called **wrong pairs**. At the end of the differential characteristic, right pairs have the pre-specified difference denoted by  $\Delta^{\text{dist}}$ . The attacker exhaustively guesses the last subkey value and performs the partial decryption until the output state of the differential characteristic. The attacker then checks whether or not the difference of the intermediate state values is  $\Delta^{\text{dist}}$ .

Here, the analytic results behave differently for right pairs and wrong pairs. If the pair is the right one and the subkey guess is right, the results of the state difference are always  $\Delta^{\text{dist}}$ .



**Figure 4.15** Framework of the key recovery attack



**Figure 4.16** Histogram of subkey guess reaching  $\Delta^{\text{dist}}$

In any other combination of right/wrong pair and right/wrong guess, such a deterministic event does not occur. The result of the partial decryption happens to be  $\Delta^{\text{dist}}$  probabilistically, which is much lower probability than 1. Therefore, by analyzing many pairs including right pairs and wrong pairs, the right guess reaches  $\Delta^{\text{dist}}$  more than any other wrong guess, which allows the attacker to detect the right subkey value. The analysis is summarized below.

Right pair with  $\Delta^{\text{dist}}$ :

- Right guess always reaches  $\Delta^{\text{dist}}$ .
- Wrong guess reaches  $\Delta^{\text{dist}}$  only probabilistically.

Wrong pair:

- Right guess reaches  $\Delta^{\text{dist}}$  only probabilistically.
- Wrong guess reaches  $\Delta^{\text{dist}}$  only probabilistically.

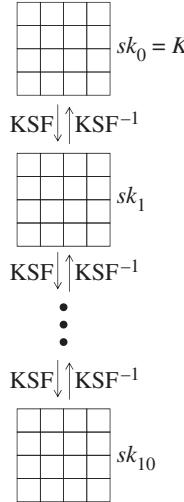
As an exact procedure to recover the last subkey, the attacker draws the histogram describing how many times each key guess reaches  $\Delta^{\text{dist}}$  after the partial decryption. The histogram is illustrated in Figure 4.16. Because the right guess reaches  $\Delta^{\text{dist}}$  more than any other wrong guess, it makes the peak in the histogram, which shows the subkey value to the attacker.

Note that as shown in Figure 4.16, there might exist wrong key guesses that are close to the peak value. Moreover, the right guess might not be the peak value. However, this is not a big issue for the attack. The attacker can test several key candidates that are in a high position in the histogram. This method is called the **ranking test**. By using the ranking test, identifying several key candidates that are likely to be a correct key is sufficient.

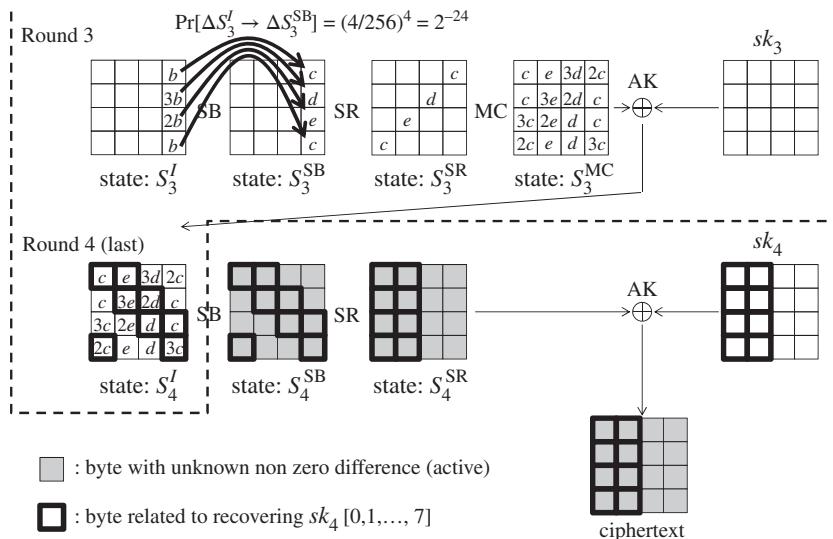
From the recovered last subkey, the attacker recovers the original key  $K$ . AES uses the original key  $K$  as the first subkey  $sk_0$ . Because the key schedule function, KSF, is easily invertible, the attacker can compute the corresponding value of  $sk_0$  from any subkey by computing the inverse of the key schedule function as illustrated in Figure 4.17. Note that in many block ciphers, the key schedule function is designed in order to avoid the loss of the computation efficiency. Thus, the key schedule function is usually invertible, and recovering a subkey is sufficient to recover the original key  $K$  in many block ciphers including AES.

#### 4.2.10 Basic Differential Cryptanalysis for Four-Round AES †

This section explains the key recovery attack against AES reduced to four rounds. One round is appended to the end of the three-round differential characteristic in Figure 4.14. The key



**Figure 4.17** Converting subkey value to original key value



**Figure 4.18** Key recovery attack against four-round AES

recovery part of this attack is depicted in Figure 4.18. Note that to describe the subkey byte positions in detail, the figure of the round function changes from Figure 4.14. In addition, note that the last round function does not compute the MixColumns operation.

All bytes in the SubBytes operation in round 4 are active. After the SubBytes operation in round 4, the output difference is nonzero. The attacker does not know the exact difference but only knows that those bytes are active. In Figure 4.18, these type of bytes are filled in gray.

#### 4.2.10.1 Filtering for Collecting Pairs

The first phase of the attack is collecting plaintext pairs that may follow the three-round differential characteristic in Figure 4.14. Because the probability of the three-round differential characteristic is  $2^{-54}$ , making at least  $2^{54}$  pairs of queries is necessary.

Here, the technique called **filtering** is introduced in order to maximize the attack efficiency. The purpose of filtering is to remove the pairs that cannot satisfy the differential characteristic with probability 1, which contributes to reducing the number of wrong pairs. To apply the filtering technique, more detailed properties of the S-box need to be considered.

Lemma 4.2.8 in Section 4.2.5 shows that for any fixed input difference, 129 output differences are never produced. In other words, only 127 output differences can be produced. As shown in Figure 4.18, the output difference of the SubBytes operation in round 4, that is,  $\Delta S_4^{\text{SB}}$  can be obtained directly from the ciphertext difference. Then, with the property of the S-box, the attacker can check whether or not the ciphertext difference in each byte is included in the 127 possible output differences from the S-box. The probability that each byte difference is included in the 127 possible output differences is  $127/256$ . Thus, with 16 bytes, the probability is

$$\left(\frac{127}{256}\right)^{16} \approx 2^{-16}. \quad (4.72)$$

The probability that a randomly generated ciphertext pair is a candidate of the one satisfying the differential characteristic is called **filtering power**, which is often denoted by a variable  $\beta$ . As discussed earlier, the filtering power  $\beta$  of this attack is

$$\beta \approx 2^{-16}. \quad (4.73)$$

#### 4.2.10.2 Signal-to-Noise Ratio

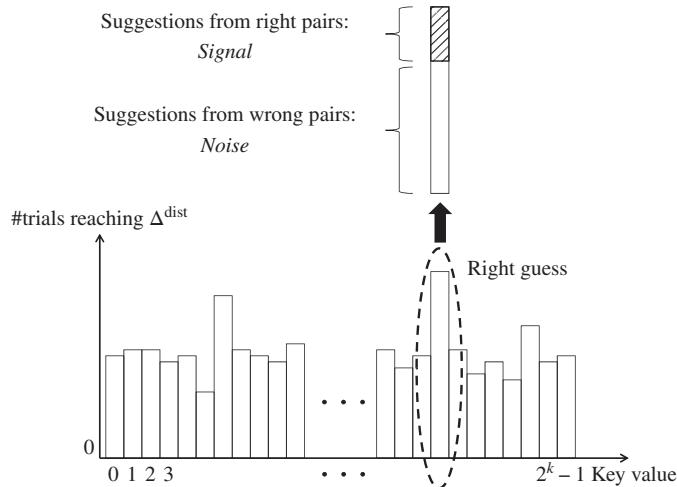
The number of necessary right pairs to recover the key depends on the value called **signal-to-noise ratio**, which is often denoted by  $S/N$ .<sup>1</sup> In the histogram of the key suggestion in Figure 4.16, regarding the correct key, the number of key suggestions consists of two contributions. One is suggestion from the right pairs, which is called signal, and the other is suggestion from the wrong pairs, which is called noise. When the ratio of the signal to noise is big enough, the key is recovered. The concept of the signal-to-noise ratio is depicted in Figure 4.19.

Suppose that the attacker queries  $N$  chosen plaintext pairs, and the probability of the characteristic is  $p$ . In addition, suppose that the bit size of the subkey guess is  $k$  and the filtering power of the attack is  $\beta$ . In the following, the amount of signal and noise is evaluated and their ratio is calculated. Note that the numbers for the attack in Figure 4.18 are  $p = 2^{-54}$ ,  $k = 64$ , and  $\beta = 2^{-16}$ .

**Signal:** Signal is the right key suggestion from right pairs. For each right pair, the partial decryption with the right guess is performed once. Hence, the amount of signal is equal

---

<sup>1</sup> In Chapter 5, another  $S/N$  will be introduced for describing the quality of the digital measurements of the side-channel information, whose noise mainly depends on the measurement environment and setup.



**Figure 4.19** Signal-to-noise ratio

to the amount of the right pair. Because  $N$  pairs are queried and the probability of the characteristic is  $p$ , the amount of signal is

$$N \cdot p. \quad (4.74)$$

**Noise:** Noise is the right key suggestion from wrong pairs. (In the attack in Figure 4.18, the combination of the right pair and the wrong guess never suggests the right key.) The number of wrong pairs is

$$N \cdot (1 - p) \approx N \quad (4.75)$$

for small  $p$ . After the filtering technique is applied,

$$N \cdot \beta \quad (4.76)$$

wrong pairs remain. Then, the number of key suggestions (including both right and wrong ones) from each remaining wrong pair is counted. By following the convention, let  $\alpha$  be this number. The total number of key suggestions from all wrong pairs can be evaluated as

$$N \cdot \beta \cdot \alpha. \quad (4.77)$$

When it comes to the attack in Figure 4.18, the evaluation of  $\alpha$  is done in each byte. From Lemma 4.2.8 in Section 4.2.5, for 126 difference values, two key values are suggested, and for one difference value, four key values are suggested. Roughly speaking, two keys are suggested for each byte. In the attack in Figure 4.18, 8 subkey bytes are guessed. Thus,  $\alpha$  is the number of all the combinations of two suggestions for each of the 8 bytes, that is,

$$\alpha = 2^8. \quad (4.78)$$

Finally, the amount of the right key suggestions out of  $N \cdot \beta \cdot \alpha$  wrong suggestions is counted. It is assumed that wrong suggestions are generated according to the uniform distribution. Because the bit size of guessed subkeys is  $k$ , the probability that a randomly generated wrong suggestion is for the right key is  $2^{-k}$ . All in all, the amount of noise is represented by

$$N \cdot \beta \cdot \alpha \cdot 2^{-k}. \quad (4.79)$$

**S/N:** By calculating the ratio of the signal to noise,  $S/N$  is represented as follows:

$$S/N = \frac{N \cdot p}{N \cdot \beta \cdot \alpha \cdot 2^{-k}} \quad (4.80)$$

$$= \frac{2^k \cdot p}{\alpha \cdot \beta}. \quad (4.81)$$

It is known that when  $S/N$  is bigger than 1, collecting two right pairs is enough to identify the right key by using the ranking test.

#### 4.2.10.3 Collecting Pairs

The signal-to-noise ratio of the attack in Figure 4.18 is as follows:

$$S/N = \frac{2^k \cdot p}{\alpha \cdot \beta} \quad (4.82)$$

$$= \frac{2^{64} \cdot 2^{-54}}{2^8 \cdot 2^{-16}} \quad (4.83)$$

$$= 2^{18}. \quad (4.84)$$

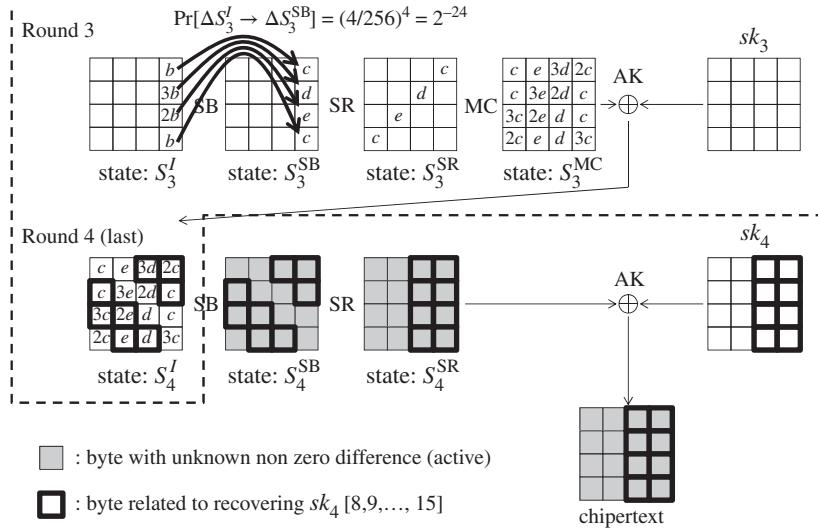
Because  $S/N$  is big enough, two right pairs are enough to recover the key. From the condition  $N \cdot 2^{-54} \geq 2$ , the number of queried messages is  $2^{55}$  pairs, which is  $2^{56}$  encryption oracle calls.

For the obtained ciphertext pairs, the filtering technique is applied, that is, the attacker confirms that the ciphertext difference in each byte can be produced from  $\Delta S_4^I$ . The pairs that do not satisfy the filtering are immediately discarded. The filtering power  $\beta$  is  $2^{-16}$ . After the filtering, the number of remaining pairs becomes

$$2^{55} \cdot 2^{-16} = 2^{39}. \quad (4.85)$$

#### 4.2.10.4 Recovering the Last Subkey

Let  $(C_i, C'_i)$  for  $i = 1, 2, \dots, 2^{39}$  be the obtained ciphertext pairs. The attacker exhaustively guesses the left two columns of the last subkey value  $sk_4[0, 1, \dots, 7]$ , and partially decrypts each ciphertext pair up to state  $S_4^I$ . With the guessed value of 8 bytes of  $sk_4$ , 8 bytes of  $S_4^{SR}[0, 1, \dots, 7]$  can be computed by taking XOR with the ciphertext pair. The ShiftRows operation can be easily inverted because it only changes the byte positions. For the 8 computed bytes in  $S_4^{SB}[0, 3, 4, 5, 9, 10, 14, 15]$ , the SubBytes operation can be inverted by looking up the inverse S-box. As a result, 8 bytes of  $S_4^I[0, 3, 4, 5, 9, 10, 14, 15]$  can be computed. The computed byte positions are stressed by bold line in Figure 4.18.



**Figure 4.20** Recovery of the right half of  $sk_4$

To generate a histogram of the key suggestion, the attacker prepares a memory to store counter values for all the  $2^{64}$  8-byte values of  $sk_4[0, 1, \dots, 7]$ . All the counter values are initialized to 0 at the beginning. If the difference of the partially decrypted 8 bytes in  $S_4^I[0, 3, 4, 5, 9, 10, 14, 15]$  matches the output difference from the differential characteristic, the attacker increases the counter for the corresponding 8-byte value of  $sk_4$ . After the analysis for all the pairs and all the 8-byte guesses of  $sk_4$ , the guess of  $sk_4$  with the highest counter value is the right value. Hence, 8 bytes of  $sk_4[0, 1, \dots, 7]$  are successfully recovered.

#### 4.2.10.5 Recovery of the Key

Independently, the attack applies the last subkey recovery procedure for the right two columns of  $sk_4$ , that is,  $sk_4[8, 9, \dots, 15]$ . The correspondence of the guessed byte positions and the ones in  $S_4^I$  is shown in Figure 4.20. The pair collection part can be shared with the analysis for the left half of  $sk_4$ . Only the subkey recovery part is independently processed of the left half.

By combining the recovered values of the left half and the right half,  $sk_4$  is fully recovered. Then, by computing  $sk_0$  from  $sk_4$  with the inverse of the key schedule function as illustrated in Figure 4.17, the original key  $K$  is successfully recovered.

#### 4.2.10.6 Attack Procedure

The entire attack procedure is described in Algorithm 4.5. Here,  $G_L$  and  $G_R$  are variables to denote 8-byte guesses for the left half of  $sk_4$  and the right half of  $sk_4$ , respectively.

**Algorithm 4.5** Key Recovery Attack against AES Reduced to 4 Rounds**Input:** differential characteristic**Output:**  $K (= sk_0)$ 

- 1: Initialize two 64-bit counters  $cnt_L[i]$  and  $cnt_R[i]$  to 0 where  $i = 0, 1, \dots, 2^{64} - 1$ ;
- Collecting Pairs // application of filtering technique**
- 2: Choose  $2^{55}$  plaintext pairs  $(P_i, P'_i)$  for  $i = 1, 2, \dots, 2^{55}$  such that  $P_i \oplus P'_i = \Delta P$ ;
- 3: **for**  $i \leftarrow 1, 2, \dots, 2^{55}$  **do**
- 4:   Query  $P_i$  and  $P'_i$  to obtain  $C_i$  and  $C'_i$ ;
- 5:   Compute  $\Delta C_i \leftarrow C_i \oplus C'_i$ , and then  $\Delta S_4^{\text{SB}}$ ;
- 6:   **if** all bytes of  $\Delta S_4^{\text{SB}}$  can be produced from  $\Delta S_4^I$  **then**
- 7:     Store the pair in a list  $L$ ;
- 8:   **end if**
- 9: **end for**
- 10:  $2^{39}$  pairs are expected. Label them as  $(C_i, C'_i)$  for  $i = 1, 2, \dots, 2^{39}$ ;
- Recovering Left Half of  $sk_4$  //  $sk_4[0, 1, \dots, 7]$**
- 11: **for**  $G_L \leftarrow 0, 1, \dots, 2^{64} - 1$  **do**
- 12:   **for**  $i \leftarrow 1, 2, \dots, 2^{39}$  **do**
- 13:      $\text{tmp} \leftarrow \text{SB}^{-1} \circ \text{SR}^{-1}(C_i \oplus G_L)$  in 8 byte positions [0, 3, 4, 5, 9, 10, 14, 15];
- 14:      $\text{tmp}' \leftarrow \text{SB}^{-1} \circ \text{SR}^{-1}(C'_i \oplus G_L)$  in 8 byte positions [0, 3, 4, 5, 9, 10, 14, 15];
- 15:     **if**  $\text{tmp} \oplus \text{tmp}' = \Delta S_4^I$  in 8 byte positions [0, 3, 4, 5, 9, 10, 14, 15] **then**
- 16:        $cnt_L[G_L] \leftarrow cnt_L[G_L] + 1$ ;
- 17:     **end if**
- 18:   **end for**
- 19: **end for**
- 20: Pick up  $G_L$  with the highest counter (or several  $G_L$  for the ranking test);
- Recovering Right Half of  $sk_4$  //  $sk_4[8, 9, \dots, 15]$**
- 21: **for**  $G_R \leftarrow 0, 1, \dots, 2^{64} - 1$  **do**
- 22:   **for**  $i \leftarrow 1, 2, \dots, 2^{39}$  **do**
- 23:      $\text{tmp} \leftarrow \text{SB}^{-1} \circ \text{SR}^{-1}(C_i \oplus G_R)$  in 8 byte positions [1, 2, 6, 7, 8, 11, 12, 13];
- 24:      $\text{tmp}' \leftarrow \text{SB}^{-1} \circ \text{SR}^{-1}(C'_i \oplus G_R)$  in 8 byte positions [1, 2, 6, 7, 8, 11, 12, 13];
- 25:     **if**  $\text{tmp} \oplus \text{tmp}' = \Delta S_4^I$  in 8 byte positions [1, 2, 6, 7, 8, 11, 12, 13] **then**
- 26:        $cnt_R[G_R] \leftarrow cnt_R[G_R] + 1$ ;
- 27:     **end if**
- 28:   **end for**
- 29: **end for**
- 30: Pick up  $G_R$  with the highest counter (or several  $G_R$  for the ranking test);
- Recovering  $K$  and its Verification**
- 31:  $sk_4 \leftarrow G_L \| G_R$ ;
- 32:  $sk_0 \leftarrow \text{KSF}^{-1} \circ \text{KSF}^{-1} \circ \text{KSF}^{-1} \circ \text{KSF}^{-1}(sk_4)$ ;
- 33: Choose one pair of plaintext  $(P^*, C^*)$  previously obtained from the oracle;
- 34: **if**  $C^* = \text{AES}_{sk_0}(P^*)$  **then**
- 35:   **return**  $sk_0$ ;
- 36: **end if**
- 37: Try other candidates of  $G_L$  and  $G_R$  by following the ranking test;

#### 4.2.10.7 Complexity Evaluation

In Algorithm 4.5, queries are made only in step 4, which is  $2^{55}$  pairs of  $(P_i, P'_i)$ . Thus, the number of query is  $2^{56}$  chosen plaintexts.

Regarding the pair collection phase, the computational cost is  $2^{54}$  XOR computations. The memory requirement is storing  $2^{39}$  pairs of  $(C_i, C'_i)$ , which is  $2^{40}$  AES state values. The computational cost and the memory requirement are negligibly small compared to the subkey recovery part.

Regarding the left half of the  $sk_4$  recovery phase, the computational cost is  $2^{64} \cdot 2^{39} \cdot 2 = 2^{104}$  AES round function computations. The memory requirement is storing counter values for  $2^{64}$  subkey candidates. The same computational cost is required for the recovery of the right half of  $sk_4$ . No additional memory is required for the right half of  $sk_4$  because the memory used for the analysis on the left half can be overwritten.

Recovering  $K$  only requires a negligible cost compared to the other steps. In the end, the computational cost for the entire attack is  $2^{104}$  AES round function computations for the left half of  $sk_4$  and  $2^{104}$  AES round function computations for the right half of  $sk_4$ , which is  $2^{105}$  AES round function computations. The computational cost is often measured by how many times the entire encryption algorithm is computed. Because the entire structure consists of four rounds, the cost of one round is 1/4 of the entire structure. The attack complexity is  $2^{105}/4 = 2^{103}$  computations of AES reduced to four rounds.

In summary, the attack complexity is as follows.

$$(Data, Time, Memory) = (2^{56}, 2^{103}, 2^{64}). \quad (4.86)$$

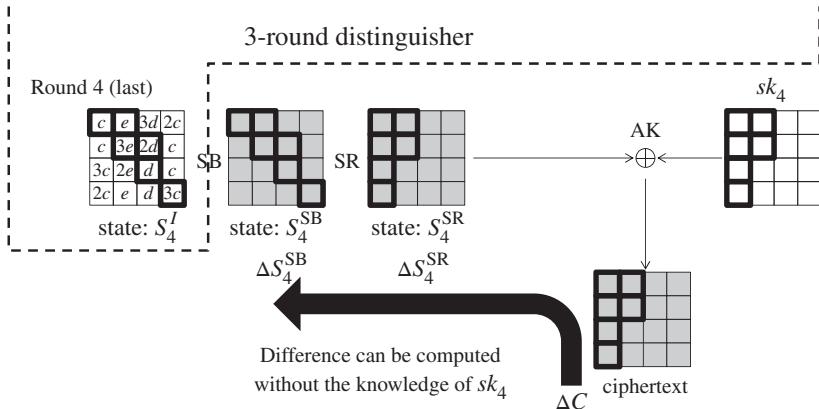
This is significantly smaller than the generic attack complexity in Equation (4.6), that is,  $(2^{128}, 2^{128}, 2^{128})$ . Therefore, AES reduced to four rounds are vulnerable against the differential cryptanalysis.

#### 4.2.11 Advanced Differential Cryptanalysis for Four-Round AES †

The above simple differential cryptanalysis can be improved so that the attack requires a much smaller cost. The bottleneck of the attack complexity lies in recovering  $sk_4$ . Two techniques to improve this part are introduced in this section.

##### 4.2.11.1 Minimizing Guessed Subkey Size

To recover the key in the differential cryptanalysis, keeping the signal-to-noise ratio,  $S/N$ , high enough is important. The number of the guessed subkey bytes is heavily related to the derivation of  $S/N$ . Recall Equations (4.82) and (4.83).  $S/N > 1$  is enough to make the attack efficient, while the current  $S/N$  ratio is too high. Hence, the number of the guessed subkey bytes can be reduced, which contributes to reducing the attack complexity. In Figure 4.18, 8 bytes of  $sk_4[0, 1, \dots, 7]$  are guessed, that is,  $k = 64$ . Here, the new attack guesses only 6 bytes of  $sk_4[0, 1, \dots, 5]$ . The guessed subkey byte positions and the partial decryption up to  $S_4^I$  are shown in Figure 4.21.



**Figure 4.21** Key recovery with 6-byte guess of  $sk_4$

The new  $S/N$  ratio becomes

$$S/N = \frac{2^k \cdot p}{\alpha \cdot \beta} \quad (4.87)$$

$$= \frac{2^{48} \cdot 2^{-54}}{2^6 \cdot 2^{-16}} \quad (4.88)$$

$$= 2^4, \quad (4.89)$$

which is bigger than 1.

Another improvement is that after the 6 bytes of  $sk_4[0, 1, \dots, 5]$  are recovered, the attack does not have to iterate the same procedure for the other 10 bytes. The recovered 6 bytes of  $sk_4[0, 1, \dots, 5]$  can be used to filter out wrong pairs. Given the fixed subkey value  $sk_4[j], 0 \leq j \leq 5$ , the probability that  $S^{-1}(C[j] \oplus sk_4[j]) \oplus S^{-1}(C'[j] \oplus sk_4[j])$  becomes the target byte difference at  $S_4^I$  is  $2/256 = 2^{-7}$ . Thus, the number of wrong pairs after the filtering with recovered 6 subkey bytes is

$$2^{30} \cdot (2^{-7})^6 = 2^{-3}, \quad (4.90)$$

which indicates that only two right pairs will survive after the filtering. Therefore, once 6 bytes of  $sk_4[0, 1, \dots, 5]$  are recovered, the other 10 bytes of  $sk_4$  can be recovered trivially, and its complexity can be ignored.

**Exercise 4.11** *The key recovery attack will not work successfully with  $2^{56}$  chosen plaintext queries when the number of guessed subkey bytes is 5 or less. Answer the reason.*

#### 4.2.11.2 Deriving Key Suggestions without Guess

In the basic attack, 8 bytes of  $sk_4$  are exhaustively guessed, and then the corresponding differences at  $S_4^I$  are computed. If the computed differences at  $S_4^I$  do not match the target, no key suggestion can be obtained. In other words, this is a waste of the computation power.

This procedure can be modified so that only computations contributing to the key suggestion are performed.

The procedure is the same until  $2^{39}$  wrong pairs are obtained after the filtering technique. Recall the procedure of applying the filtering technique. For each ciphertext pair, the attacker computes the corresponding difference at state  $S_4^{\text{SB}}$  and checks if all byte differences can be produced by  $\Delta S_4^I$  specified by the differential characteristic. An important fact here is that both of the input and output difference for S-boxes in round 4 are determined without determining the subkey values  $sk_4$ .

Recall Lemmas 4.2.8 and 4.2.9, which specify the property of the AES S-box. When an input difference  $\Delta \text{IN}$  and an output difference  $\Delta \text{OUT}$  for the S-box with nonzero probability are given, they have two solutions (values satisfying the differential propagation of the S-box) with probability 126/127 and four solutions with probability 1/127. Here, the new strategy is preparing the look-up table  $T_{up}$  that takes a pair of  $(\Delta \text{IN}, \Delta \text{OUT})$  as input and returns all values  $x$  such that  $S(x) \oplus S(x \oplus \Delta \text{IN}) = \Delta \text{OUT}$  as shown in Figure 4.22.

The procedure to obtain key suggestions for a pair of ciphertexts  $(C, C')$  is explained below.

1. The difference is computed from  $\Delta C$  to  $\Delta S_4^{\text{SB}}$ , and the filtering technique is applied to eliminate wrong pairs.
2. With the look-up table  $T_{up}$ , obtain all combinations of solutions for each S-box. Because each byte has about two solutions, the attacker obtains about  $2^6$  combinations of the solutions for each byte.
3. For each of the  $2^6$  combined solutions, the values of the 6 bytes of  $S_4^{\text{SR}}[0, 1, \dots, 5]$  are fixed. The key suggestions for 6 bytes of  $sk_4[0, 1, \dots, 5]$  are obtained by computing

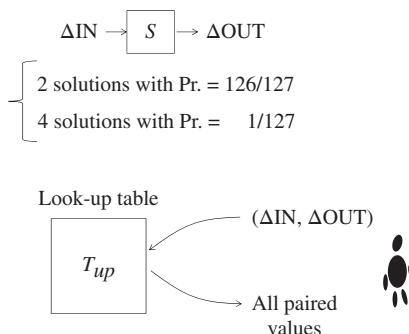
$$sk_4[0, 1, \dots, 5] \leftarrow S_4^{\text{SR}}[0, 1, \dots, 5] \oplus C[0, 1, \dots, 5]. \quad (4.91)$$

Note that  $S_4^{\text{SR}}[0, 1, \dots, 5] \oplus C'[0, 1, \dots, 5]$  suggests the same key value, thus does not have to be computed.

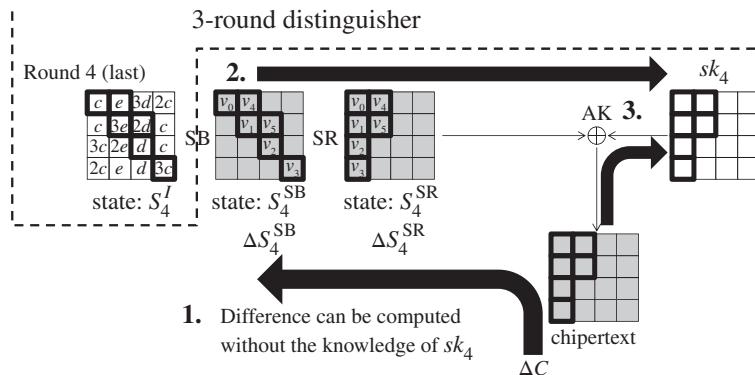
The procedure is illustrated in Figure 4.23.

#### 4.2.11.3 Attack Procedure

The updated attack procedure is described in Algorithm 4.6. The pair collection phase is not included in Algorithm 4.6. Instead, the attacker derives the suggestions as soon as each pair



**Figure 4.22** Look-up table returning all solutions for S-box



**Figure 4.23** Efficient key suggestions derivation

satisfying the filtering condition is obtained. This is another optimization of the attack. Unfortunately, because the pair collection phase is not the bottleneck of the attack complexity, this optimization does not contribute to reducing the entire attack complexity. However, the technique is useful when the fault analysis is improved with theoretical cryptanalytic techniques in Chapter 6.

#### 4.2.11.4 Complexity Evaluation

In Algorithm 4.6, queries are made in the same manner as Algorithm 4.5. Thus, the number of queries is the same as Algorithm 4.5, which is  $2^{56}$  chosen plaintexts.

Regarding the procedure for recovering the first 6 bytes of  $sk_4$ , the computational cost is  $2^{54}$  XOR computations in step 5. After the filtering technique is applied in step 6, only  $2^{39}$  pairs are examined. Therefore, step 10 requires  $2^{39} \cdot 2^6 = 2^{45}$  AES one-round computation. The memory requirement is for storing  $2^{48}$  counter values and  $2^{39}$  pairs of  $(C_i, C'_i)$ .

The procedure after recovering  $sk_4[0, 1, \dots, 5]$  is at most  $2 \cdot 2^{39}$ , which is much smaller than the other part. Thus, the computational cost for the entire attack is  $2^{55}$  XOR computations.

In summary, the attack complexity is as follows.

$$(Data, Time, Memory) = (2^{56}, 2^{55}, 2^{48}). \quad (4.92)$$

Compared to the attack complexity for the basic attack in Equation (4.86), the complexity of the advanced attack is improved significantly.

#### 4.2.12 Preventing Differential Cryptanalysis †

The essence of the differential cryptanalysis is a differential characteristic satisfied with a high probability. In other words, by avoiding a high probability differential characteristic, the differential cryptanalysis can be prevented. As explained so far, the probability of the differential characteristic decreases only when a difference goes through the S-box. Let  $2^{-\delta}$  be the maximum probability of the differential propagation in a single S-box. Let  $N_S$  be the total number

**Algorithm 4.6** Advanced Key Recovery Attack against AES Reduced to 4 Rounds**Input:** differential characteristic**Output:**  $K (= sk_0)$ 

1: Initialize a 48-bit counters  $cnt[i]$  to 0 where  $i = 0, 1, \dots, 2^{48} - 1$ ;

**Recovering First 6 Bytes of  $sk_4$**  //  $sk_4[0, 1, \dots, 5]$

2: Choose  $2^{55}$  plaintext pairs  $(P_i, P'_i)$  for  $i = 1, 2, \dots, 2^{55}$  such that  $P_i \oplus P'_i = \Delta P$ ;

3: **for**  $i \leftarrow 1, 2, \dots, 2^{55}$  **do**

4:   Query  $P_i$  and  $P'_i$  to obtain  $C_i$  and  $C'_i$ ;

5:   Compute  $\Delta C_i \leftarrow C_i \oplus C'_i$ , and then  $\Delta S_4^{\text{SB}}$ ;

6:   **if** all bytes of  $\Delta S_4^{\text{SB}}$  can be produced from  $\Delta S_4^I$  (with  $\Pr = 2^{-16}$ ) **then**

7:     Store the pair in a list  $L$ ;

8:     Derive all the solutions of the 6 S-boxes denoted by  $v_0, v_1, \dots, v_5$  with  $T_{up}$ ;

9:     **for** all (about  $2^6$ ) possible combinations of  $v_0, v_1, \dots, v_5$  **do**

10:        $\text{tmp} \leftarrow S_4^{\text{SR}}[0, 1, \dots, 5] \oplus C_i[0, 1, \dots, 5]$ ;

11:        $cnt[\text{tmp}] \leftarrow cnt[\text{tmp}] + 1$ ;

12:     **end for**

13:   **end if**

14: **end for**

15: Set  $sk_4[0, 1, \dots, 5]$  to the most suggested value;

**Filtering with Recovered 6 Bytes of  $sk_4$** 

16: Label  $2^{39}$  expected pairs in the list  $L$  as  $(C_i, C'_i)$  for  $i = 1, 2, \dots, 2^{39}$ ;

17: **for**  $i \leftarrow 1, 2, \dots, 2^{39}$  **do**

18:    $\text{tmp} \leftarrow \text{SB}^{-1} \circ \text{SR}^{-1}(C_i[j] \oplus sk_4[j])$  for  $j = 0, 1, \dots, 5$ ;

19:    $\text{tmp}' \leftarrow \text{SB}^{-1} \circ \text{SR}^{-1}(C'_i[j] \oplus sk_4[j])$  for  $j = 0, 1, \dots, 5$ ;

20:   **if**  $\text{tmp} \oplus \text{tmp}' \neq \Delta S_4^I$  in at least 1 byte **then**

21:     Remove  $(C_i, C'_i)$  from the list  $L$ ;

22:   **end if**

23: **end for**

24: **for** remaining 2 ciphertext pairs **do**

25:   **for** all (about  $2^{10}$ ) possible combinations of  $v_6, v_7, \dots, v_{15}$  **do**

26:     Obtain key suggestions of  $sk_4$  by  $S_4^{\text{SR}}[6, 7, \dots, 15] \oplus C_i[6, 7, \dots, 15]$ ;

27:      $sk_0 \leftarrow \text{KSF}^{-1} \circ \text{KSF}^{-1} \circ \text{KSF}^{-1} \circ \text{KSF}^{-1}(sk_4)$ ;

28:     Choose one pair of plaintext  $(P^*, C^*)$  previously obtained from the oracle;

29:     **if**  $C^* = \text{AES}_{sk_0}(P^*)$  **then**

30:       **return**  $sk_0$ ;

31:     **end if**

32: **end for**

33: **end for**

of active S-boxes in a characteristic. Then, the maximum probability of the differential characteristic is bounded by

$$(2^{-\delta})^{N_S} = 2^{-\delta \cdot N_S}. \quad (4.93)$$

In AES, from Lemma 4.2.8, the maximum probability of the differential propagation in a single S-box is  $4/256 = 2^{-6}$ . Thus, the entire probability of the differential characteristic

is bounded by

$$2^{-6N_S}. \quad (4.94)$$

On the one hand, the differential cryptanalysis needs to collect at least one pair of plaintext to satisfy the differential characteristic. Thus, the attack on AES needs to query at least  $2^{6N_S}$  pairs. On the other hand, the number of texts that the attack can query is upper-bounded by  $2^b$ , where  $b$  is the bit-length of the block size. For AES,  $b$  is 128. Therefore, satisfying the differential characteristic of AES becomes impossible if the number of active S-boxes,  $N_S$ , satisfies the following condition:

$$2^{6N_S} > 2^{128} \quad (4.95)$$

$$\Rightarrow N_S \geq 22. \quad (4.96)$$

AES was designed so that

$$N_S \geq 25 \quad (4.97)$$

is guaranteed for four rounds. Therefore, the full-round (10-round) AES is expected to resist the differential cryptanalysis even if a few rounds are appended for the key recovery part around the differential characteristic. The strategy of proving the number of minimum active S-boxes in AES is called a **wide trail strategy**.

#### 4.2.12.1 Wide Trail Strategy for Four-Round AES

The rough sketch of the wide trail strategy is introduced below. A **differential propagation for four rounds**, from round  $i$  to round  $i + 3$ , is considered. Those four rounds are illustrated in Figure 4.24. As long as **two different plaintexts are encrypted** under the same key, each internal state has nonzero **difference in at least 1 byte**.

The analysis starts from 1 active column for the MixColumns operation in round  $i + 1$ . To complete the proof, four cases starting from 1 to 4 active columns must be evaluated. In this book, only the case with one **active column is explained**. The analysis is irrespective of the position of the one active column. In Figure 4.24, the second column is chosen as an example.

From the **property of the MDS matrix** in the MixColumns operation, the **sum of the number of active bytes before and after the MixColumns operation is at least 5**. Let  $\alpha$  and  $\beta$  be the number of active bytes before the MixColumns operation,  $S_{i+1}^{\text{SR}}$ , and after the **MixColumns operation**,  $S_{i+1}^{\text{MC}}$ , respectively. Then, the following condition is obtained.

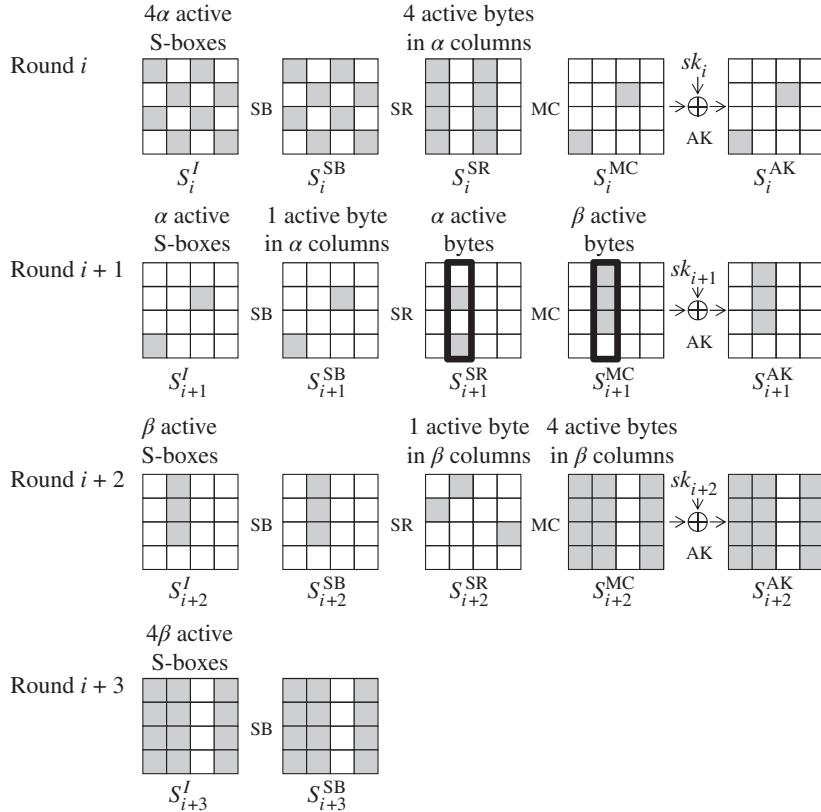
$$\alpha + \beta \geq 5. \quad (4.98)$$

Those derive  $\alpha$  active S-boxes for the SubBytes operation in round  $i + 1$  and  $\beta$  active S-boxes for the SubBytes operation in round  $i + 2$ . At this stage, together with Equation (4.98),

$$N_S \geq \alpha + \beta \geq 5 \quad (4.99)$$

is proven for AES reduced to two rounds.

Consider propagating the difference at  $S_{i+1}^{\text{MC}}$  in the forward direction.  $\beta$  active bytes in  $S_{i+1}^{\text{MC}}$  are located in one column. Those move **to different columns by the next ShiftRows operation** in round  $i + 2$ . In other words, the probability that several active bytes move to the same column is ensured to be 0. Therefore, at state  $S_{i+2}^{\text{SR}}$ , there are  $\beta$  active columns with 1 active byte. After



**Figure 4.24** Proof of minimum number of active S-boxes for AES four rounds. Gray byte shows an example of the differential propagation for  $\alpha = 2$  and  $\beta = 3$

the subsequent MixColumns operation, those make 4 bytes active in  $\beta$  columns. Finally,  $4\beta$  active bytes are ensured for the SubBytes operation in round  $i + 3$ .

Consider propagating the difference at  $S_{i+1}^{SR}$  in the backward direction.  $\alpha$  active bytes in  $S_{i+1}^{SR}$  are located in one column. Those move to different columns by the next inverse ShiftRows operation in round  $i + 1$ . Therefore, at state  $S_i^{MC}$ , there are  $\alpha$  active columns with 1 active byte. After the next inverse MixColumns operation, those make 4 bytes active in  $\alpha$  columns. Finally,  $4\alpha$  active columns are ensured for the SubBytes operation in round  $i$ .

As a result, the number of active S-boxes,  $N_S$ , for AES reduced to four rounds is at least

$$4\alpha + \alpha + \beta + 4\beta. \quad (4.100)$$

From Equation (4.98), the following result is obtained.

$$N_S \geq 5(\alpha + \beta) \geq 25. \quad (4.101)$$

As explained before, it ensures that the maximum number of rounds of the differential characteristic for AES that can be used for the attack is 3. The differential characteristic in

Figure 4.14 is an example of this case. It also indicates that breaking full-round AES with the differential cryptanalysis is almost impossible.

**Exercise 4.12** In the earlier discussion, the condition  $N_S \geq 22$  only guarantees the security of a single key. Given the fact that  $N_S \geq 25$  for four-round AES, what security incidents occur when there are  $2^{32}$  users with  $2^{32}$  different keys?

#### 4.2.12.2 Recent Design Trend and Role of Differential Cryptanalysis

Most of the recent block cipher designs have some security proof so that the cipher can resist the basic differential cryptanalysis. Hence, the theoretical attack with the differential cryptanalysis is impossible for those designs.

The differential cryptanalysis is a base of many other advanced cryptanalytic techniques. Thus, learning its mechanism is useful to learn other attacks. Moreover, when the attacker can exploit the fault injections, the differential cryptanalysis becomes one of the most powerful approaches owing to its simplicity. Differential cryptanalysis combined with fault analysis will be explained in Chapter 6.

### 4.3 Impossible Differential Cryptanalysis

#### 4.3.1 Basic Concept and Definition

There are several cryptanalyses using an impossible event of the encryption or decryption algorithm. Cryptanalysis with impossible differential propagation was independently found by Knudsen and Biham and Shamir. It exploits a differential propagation that is never satisfied.

**Definition 4.3.1 (Impossible Differential Characteristic)** Let  $\Delta x$  and  $\Delta y$  be the input and output difference of function  $F$ . A pair of  $(\Delta x, \Delta y)$  is an impossible differential characteristic with respect to  $F$  if

$$\Pr[\Delta x \xrightarrow{F} \Delta y] = 0. \quad (4.102)$$

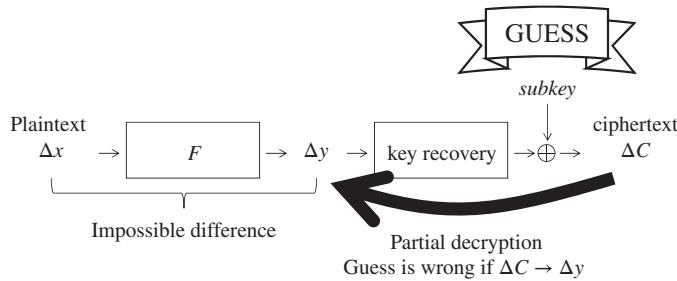
An example of impossible differential characteristic is zero input difference and nonzero output difference, that is,

$$0 \xrightarrow{F} y, \text{ for } y \neq 0. \quad (4.103)$$

Similarly, when  $F$  is a bijective function, the differential propagation from nonzero input difference to zero output difference is impossible, that is,

$$x \xrightarrow{F} 0, \text{ for } x \neq 0. \quad (4.104)$$

The basic idea to utilize an impossible differential characteristic in order to recover the key is as follows, which is also illustrated in Figure 4.25. Suppose that there is a plaintext pair  $(P, P \oplus \Delta x)$  and the corresponding ciphertext pair  $(C, C')$ . In addition, suppose that



**Figure 4.25** Mechanism of impossible differential cryptanalysis

$\Pr[\Delta x \xrightarrow{F} \Delta y] = 0$ , that is,  $(\Delta x, \Delta y)$  is an impossible differential characteristic with respect to  $F$ . To recover the key, the attacker exhaustively guesses the last subkey value and partially decrypts  $C$  and  $C'$  until the output state of  $F$ . If the difference of the partially decrypted results becomes  $\Delta y$ , the subkey guess is wrong, and it can be removed from subkey candidates. The attack recovers the right key by eliminating all wrong subkey guesses.

A comparison of the differential cryptanalysis and the cryptanalysis with impossible differential characteristic is given below.

- In the differential cryptanalysis, the attacker aims to construct a differential characteristic that is satisfied with a high probability, and aims to detect the right key from the obtained key suggestions.
- In the cryptanalysis with impossible differential propagation, the attacker aims to construct a differential characteristic that is satisfied with probability 0, and aims to discard all the wrong guesses from the obtained key suggestions.

The analysis is often termed **impossible differential cryptanalysis**. In this book, the term impossible differential cryptanalysis is used by following the convention of the cryptographic community.

As shown in Figure 4.25, the attack consists of two parts:

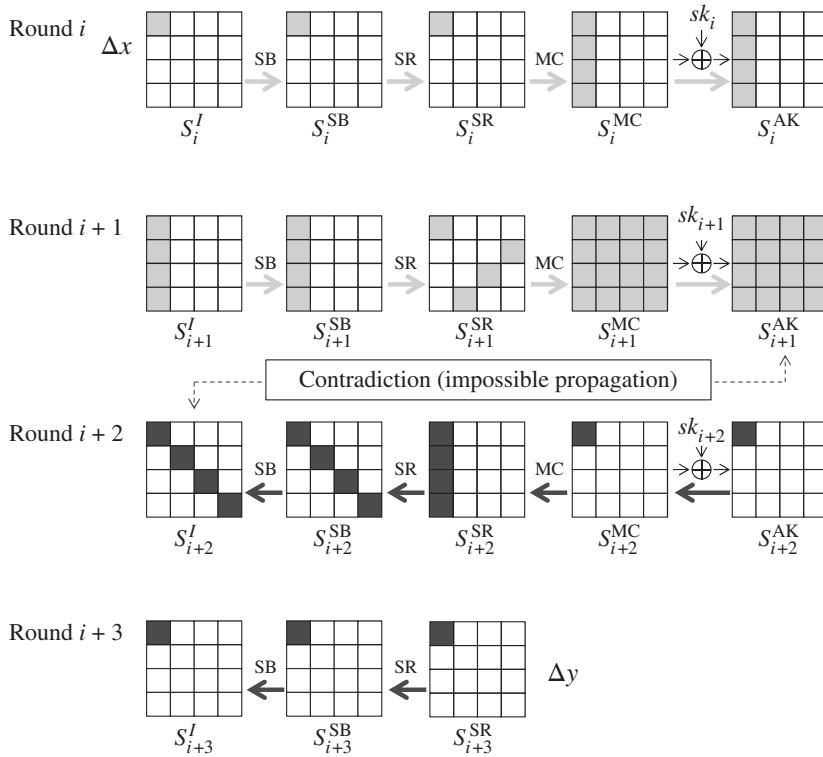
- constructing impossible differential characteristic, and
- appending several rounds for recovering subkey value.

### 4.3.2 Impossible Differential Characteristic for 3.5-round AES

Impossible differential characteristics can be constructed for 3.5 rounds of AES. Here, 0.5 round represents a composition of the SubBytes and ShiftRows operations. 3.5 rounds cover from state  $S_i^I$  to  $S_{i+3}^{SR}$  as shown in Figure 4.26.

The input difference  $\Delta x$  to 3.5 rounds has nonzero difference only in 1 byte. The active byte can be located in any position. In Figure 4.26, the byte position 0 is chosen as an example. The input difference  $\Delta x$  is described as follows:

$$\Delta x = (\Delta, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0), \quad (4.105)$$



**Figure 4.26** Impossible differential characteristic for 3.5-round AES. Gray bytes denote active bytes. During the differential trace in forwards, active bytes are colored in light gray. During the differential trace in backwards, active bytes are colored in dark gray

where  $\Delta$  represents a nonzero difference, that is,  $\Delta \in \{1, 2, \dots, 255\}$ . Similarly to the differential cryptanalysis, a byte with nonzero difference is called **active**.

The output difference  $\Delta y$  from 3.5 rounds also has nonzero difference only in 1 byte. The active byte can be located in any position. In Figure 4.26, the byte position 0 is chosen as an example. The output difference  $\Delta y$  is described as follows.

$$\Delta y = (\Delta', 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0), \quad (4.106)$$

where  $\Delta'$  represents a nonzero difference independent of  $\Delta$ .

The impossible differential characteristic in Figure 4.26 can be formally summarized in the following lemma.

**Lemma 4.3.2**  $\Pr[\Delta x \xrightarrow{\text{AES } 3.5R} \Delta y] = 0$ .

*Proof.* To prove the lemma, the difference of each byte in each state is evaluated. Here, the attacker focuses on if the difference of each byte is zero or nonzero, that is, active or inactive.

Consider propagating the input difference  $\Delta x$  in the forward direction by two rounds. After the SubBytes operation in round  $i$ , inactive bytes are still inactive. For the active byte with

difference  $\Delta$ , the attacker does not know the difference after the S-box application. On the other hand, nonzero difference never becomes zero. Thus, the attacker knows that the byte is still active even after the S-box application, which leads to

$$\Delta S_i^{\text{SB}} = (\Delta, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0). \quad (4.107)$$

By the next ShiftRows operation, the active-byte position in the top row does not change. Thus,

$$\Delta S_i^{\text{SR}} = (\Delta, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0). \quad (4.108)$$

By the next MixColumns operation, 1 active byte enters the MixColumns operation. Owing to the property of the MixColumns operation, all bytes in the active column become active. Thus,

$$\Delta S_i^{\text{MC}} = (\Delta, \Delta, \Delta, \Delta, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0). \quad (4.109)$$

Because the difference of the active byte is not fixed, considering exact coefficients of the matrix does not help the attack. Hence, coefficients are omitted and those bytes are simply marked as active, that is, each difference is one of  $\{1, 2, \dots, 255\}$ . After the AddRoundKey operation, the difference does not change.

$$\Delta S_i^{\text{AK}} = \Delta S_{i+1}^I = (\Delta, \Delta, \Delta, \Delta, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0). \quad (4.110)$$

Owing to the similar discussion as round  $i$ , the difference is propagated as

$$\Delta S_{i+1}^{\text{SB}} = (\Delta, \Delta, \Delta, \Delta, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0), \quad (4.111)$$

$$\Delta S_{i+1}^{\text{SR}} = (\Delta, 0, 0, 0, 0, 0, 0, \Delta, 0, 0, \Delta, 0, 0, \Delta, 0, 0). \quad (4.112)$$

Because all columns have 1 active byte at  $S_{i+1}^{\text{SR}}$ , from the property of the MixColumns operation, all bytes become active at  $S_{i+1}^{\text{MC}}$ .

$$\Delta S_{i+1}^{\text{MC}} = (\Delta, \Delta, \Delta), \quad (4.113)$$

$$\Delta S_{i+1}^{\text{AK}} = (\Delta, \Delta, \Delta). \quad (4.114)$$

In summary, the input difference  $\Delta x$  makes all bytes active after two rounds of AES. In Figure 4.26, the differential propagation in the first two rounds is drawn by light gray.

Consider propagating the output difference  $\Delta y$  in the backward direction by 1.5 rounds. The inverse ShiftRows, inverse SubBytes, and inverse AddRoundKey operations do not change the active-byte position in the top row. Thus, the difference is propagated as follows:

$$\Delta S_{i+3}^{\text{SB}} = (\Delta', 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0), \quad (4.115)$$

$$\Delta S_{i+2}^{\text{AK}} = (\Delta', 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0), \quad (4.116)$$

$$\Delta S_{i+2}^{\text{MC}} = (\Delta', 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0). \quad (4.117)$$

The inverse MixColumns operation makes 4 bytes in the active column active, which derives

$$\Delta S_{i+2}^{\text{SR}} = (\Delta', \Delta', \Delta', \Delta', 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0). \quad (4.118)$$

The subsequent inverse ShiftRows operation moves 4 active bytes in one column to all columns, and the same form is preserved even after the inverse SubBytes operation.

$$\Delta S_{i+2}^{\text{SB}} = (\Delta', 0, 0, 0, 0, \Delta', 0, 0, 0, 0, \Delta', 0, 0, 0, 0, \Delta'), \quad (4.119)$$

$$\Delta S_{i+2}^I = (\Delta', 0, 0, 0, 0, \Delta', 0, 0, 0, 0, \Delta', 0, 0, 0, 0, \Delta'). \quad (4.120)$$

In summary, the output difference  $\Delta y$  makes 1 byte of each column active after 1.5 inverse AES rounds. In Figure 4.26, the differential propagation in the last 1.5 rounds is drawn by dark gray.

The states  $S_{i+1}^{\text{AK}}$  and  $S_{i+2}^I$  are identical. The forward propagation for two rounds indicates that all bytes are active, while the backward propagation for 1.5 rounds indicates that only 4 bytes are active, and the other 12 bytes are always inactive. This is a contradiction. In other words, the input difference  $\Delta x$  cannot be the output difference  $\Delta y$  after 3.5 rounds of the AES computation.  $\square$

#### 4.3.2.1 Another Impossible Differential Characteristic for 3.5-Round AES

The form of the output difference of the 3.5-round impossible differential characteristic  $\Delta y$  can be chosen from many other choices. For example, the number of active bytes can be increased up to 12 bytes:

$$\Delta y = (0, \Delta', \Delta', \Delta', \Delta', \Delta', \Delta', 0, \Delta', \Delta', 0, \Delta', \Delta', 0, \Delta', \Delta'). \quad (4.121)$$

The characteristic is described in Figure 4.27. Owing to the similarity to the previous 3.5-round impossible differential characteristic, the detailed proof is omitted. In short, the 1.5-round computation in the backward direction from  $S_{i+3}^{\text{SR}}$  always makes 4 bytes inactive, which contradicts to the result of the two-round computation in the forward direction from state  $S_i^I$ .

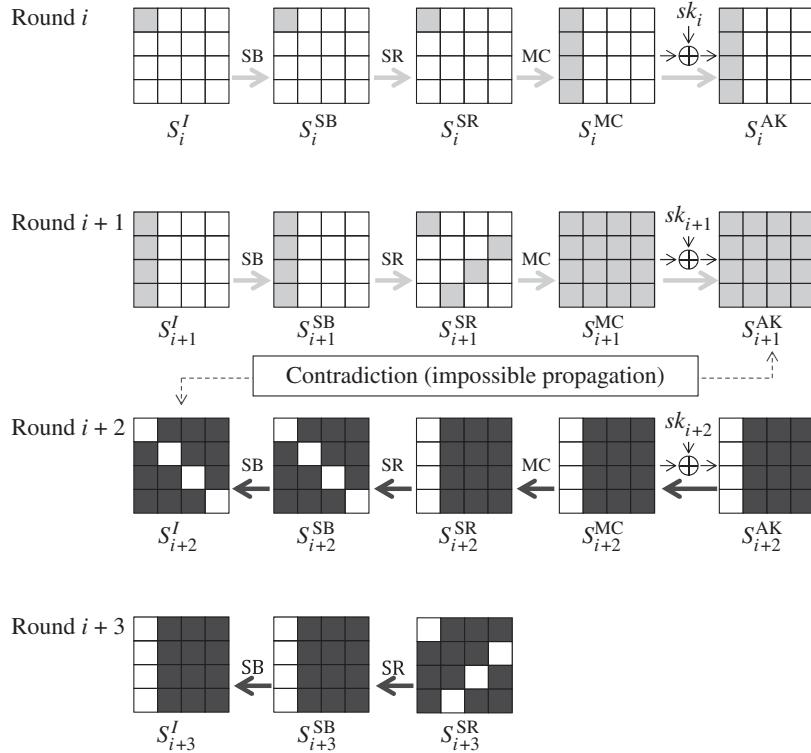
It is hard to conclude which of the 3.5-round impossible differential characteristics is stronger. The attacker needs to carefully choose which 3.5-round impossible differential characteristic is used to optimize the entire attack.

#### Exercise 4.13

- (a) Describe an example of 3.5-round impossible differential characteristic with four active S-boxes in the input and 12 active S-boxes in the output.
- (b) Describe an example of 3.5-round impossible differential characteristic with eight active S-boxes in the input and three active S-boxes in the output.

#### 4.3.3 Key Recovery Attacks for Five-Round AES

To demonstrate how the key recovery part works, the simple attack for AES reduced to five rounds is explained. In this attack, one round is added as the key recovery part before the



**Figure 4.27** Another impossible differential characteristic for 3.5-round AES

3.5-round impossible differential characteristic in Figure 4.27. The attack supposes that the AES last round function is used in the last round of the reduced-round variant. The attack structure is described in Figure 4.28. The attack first aims to recover 4 bytes of the first subkey  $sk_0[0, 5, 10, 15]$ .

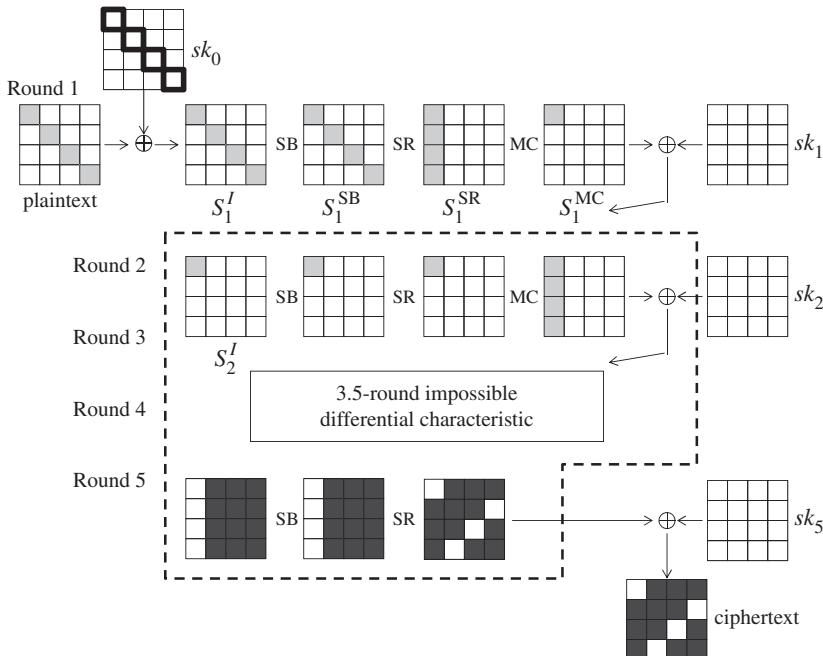
In Figure 4.28, the AddRoundKey operation is added to the end of the 3.5-round distinguisher. Because the AddRoundKey operation does not change the difference, the 3.5-round impossible differential characteristic can be extended to include the ciphertext difference of the form

$$\Delta C = (0, \Delta', \Delta', \Delta', \Delta', \Delta', \Delta', 0, \Delta', \Delta', 0, \Delta', \Delta', 0, \Delta', \Delta'). \quad (4.122)$$

The input difference to the 3.5-round impossible differential characteristic is propagated in the backward direction by one round. It determines the active-byte positions of the plaintext, which are represented as

$$\Delta P = (\Delta, 0, 0, 0, 0, \Delta, 0, 0, 0, 0, \Delta, 0, 0, 0, 0, \Delta). \quad (4.123)$$

The exact value of  $\Delta$  can be any nonzero value, and can be determined independently for 4 bytes. For simplicity, the following discussion assumes that  $\Delta P$  is fixed to one choice of the above form, and the attack is later optimized by relaxing this assumption.



**Figure 4.28** Key recovery attack for five-round AES

#### 4.3.3.1 Collecting Pairs

The first phase of the attack is collecting many pairs of plaintexts whose corresponding ciphertext difference satisfies the form of  $\Delta C$ . For simplicity, the attack assumes the chosen plaintext attack model so that the attacker can always choose the pair of plaintexts with the specified difference  $\Delta P$ . Let  $2^N$  be the number of plaintext pairs queried to the encryption oracle. The probability that a randomly chosen pair has the difference  $\Delta C$  at the ciphertext is  $2^{-32}$  because inactive 4 bytes (32 bits) must have difference zero. Thus, the number of pairs satisfying  $\Delta C$  is

$$2^N \cdot 2^{-32} = 2^{N-32}. \quad (4.124)$$

#### 4.3.3.2 Recovery of Subkey $sk_0[0, 5, 10, 15]$

The attacker aims to derive wrong key suggestions that generate  $\Delta S_2^I$  at the input state to the 3.5-round impossible differential characteristic. Because the AddRoundKey operation does not change the difference, computing  $\Delta S_1^{MC}$  is sufficient, which avoids guessing the subkey value  $sk_1$ .

The simple method is guessing the 4 bytes of  $sk_0[0, 5, 10, 15]$ , and applying the partial encryption from  $\Delta P$  to  $\Delta S_1^{MC}$ . However, as explained in the advanced differential cryptanalysis, wrong key suggestions can be derived more efficiently. Because the AddRoundKey operation with  $sk_0$  does not change the difference,  $\Delta P = \Delta S_1^I$ .  $\Delta P$  is chosen by the attacker, thus  $\Delta S_1^I$  is known to the attacker for each pair.

The attacker then chooses the difference of the active byte at state  $S_1^{\text{MC}}$ . It can be any value but for 0, thus chosen from 255 choices. For each of the 255 choices of  $\Delta S_1^{\text{MC}}$ , the corresponding difference  $\Delta S_1^{\text{SB}}$  is uniquely computed by applying the inverse MixColumns and the inverse ShiftRows operations. The 255 results of  $\Delta S_1^{\text{SB}}$  are stored in a look-up table  $T[i]$ , where  $i = 1, 2, \dots, 255$ . More precisely,  $T[i]$  contains a 4-byte difference of  $\Delta S_1^{\text{SB}}[0, 5, 10, 15]$ , which is derived from  $\Delta S_1^{\text{MC}}$  with active-byte difference value  $i$ .

The values of the pair that maps the fixed  $\Delta S_1^I$  to each of the 255  $\Delta S_1^{\text{SB}}$  stored in  $T[i]$  can be easily obtained by exploiting the property of the AES S-box. From Lemmas 4.2.8 and 4.2.9, for a randomly fixed input difference and an output difference of the S-box, the number of paired values that can achieve this propagation is as follows.

- two paired values with probability 126/256,
- four paired values with probability 1/256, and
- the propagation is impossible (0 paired value) with probability 129/256.

For each plaintext pair with a fixed  $\Delta S_1^I$ , the match with  $T[i]$  is examined for  $i = 1, 2, \dots, 255$  with respect to four active S-boxes. The expected number of solutions for one pair is

$$\left(2 \times \frac{126}{256} + 4 \times \frac{1}{256}\right)^4 = 1. \quad (4.125)$$

By examining 255  $T[i]$ , 255 paired values that map the fixed  $\Delta P$  to the desired  $\Delta S_1^{\text{SB}}$  are expected.

With the plaintext pair  $(P, P \oplus \Delta P)$  and the state value pair  $S_1^I, S_1^I \oplus \Delta P$ , the corresponding subkey value  $sk_0[0, 5, 10, 15]$  can be computed by simply XORing those values, that is,

$$sk_0[0, 5, 10, 15] \leftarrow P[0, 5, 10, 15] \oplus S_1^I[0, 5, 10, 15], \quad (4.126)$$

$$sk_0[0, 5, 10, 15] \leftarrow P[0, 5, 10, 15] \oplus S_1^I[0, 5, 10, 15] \oplus \Delta P[0, 5, 10, 15]. \quad (4.127)$$

Those two values lead to the impossible differential characteristic, and thus known to be wrong. The attacker can discard those two values from candidate values for  $sk_0[0, 5, 10, 15]$ . Note that the above-mentioned procedure derives  $255 \cdot 2$  values of  $sk_0[0, 5, 10, 15]$ . However, each value is counted twice. Thus, the number of distinct values of  $sk_0[0, 5, 10, 15]$  is 255. By using 255 paired values of the internal state  $S_1^I$ , 255 wrong key values are discarded. The idea of the efficient key derivation is summarized in Figure 4.29.

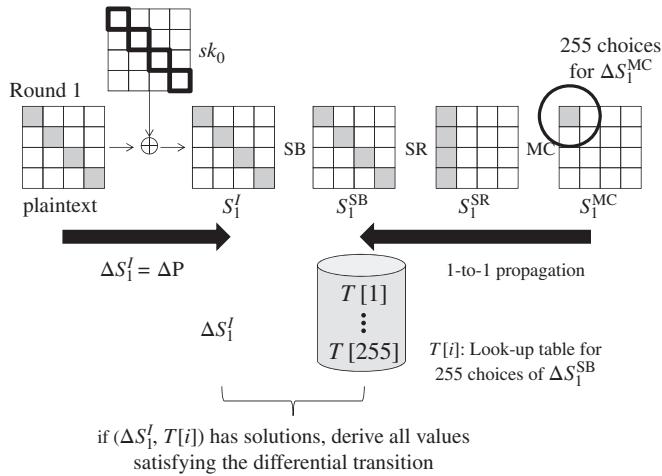
#### 4.3.3.3 The Number of Plaintext Pairs

After the pair collection phase, the attacker obtains  $2^{N-32}$  pairs that also satisfy the output difference  $\Delta C$ . Because each pair can derive 255 wrong key suggestions of  $sk_0[0, 5, 10, 15]$ ,

$$2^{N-32} \cdot 255 \approx 2^{N-24} \quad (4.128)$$

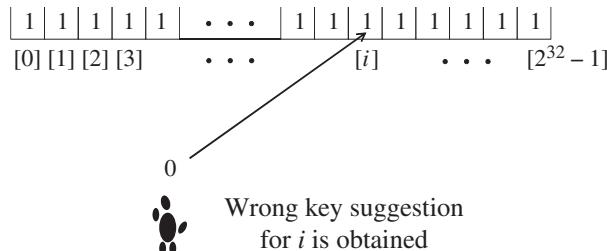
wrong key suggestions are obtained in total.

What is the minimum value of  $N$  to identify the right value of 32-bit subkey  $sk_0[0, 5, 10, 15]$ ? Obtaining  $2^{32}$  wrong key suggestions, that is,  $N = 56$ , is not sufficient



**Figure 4.29** Efficient derivation of wrong subkey suggestions

Key space :  $K$



**Figure 4.30** Reducing key space

because the same wrong key can be suggested more than once. More detailed analysis is necessary.

In the impossible differential attack, the attacker first prepares a set of all candidate values for the target subkey. In this attack, the attacker prepares a memory  $\kappa$  of  $2^{32}$  entries for 32-bit subkey  $sk_0[0, 5, 10, 15]$ , which is used to record the correctness and incorrectness of each subkey value.  $\kappa$  is often called **key space** or **subkey space** depending on which is the target of the impossible differential attack.  $\kappa$  is first initialized so that all  $2^{32}$  values of  $sk_0[0, 5, 10, 15]$  can be right values. If the attacker obtains wrong key suggestions, the corresponding values are removed from the key space  $\kappa$ . The analysis is iterated until the size of the key space  $\kappa$  becomes 1.

The procedure of reducing the key space is illustrated in Figure 4.30. At first, all entries of  $\kappa$  are initialized to a single bit “1,” which represents that the value can be a right key. If the attacker obtains wrong key suggestions, the corresponding bit is replaced with “0,” which represents that the value is not a right key.

At the initial stage, the size of the remaining key space is  $2^{32}$ . By deriving one wrong key suggestion, the size of the remaining key space becomes

$$2^{32} - 1. \quad (4.129)$$

This is a precise evaluation, however, using this metric (with subtraction) for counting the remaining subkey space is hard to deal with the event in which the same wrong key value is suggested more than once. To solve this issue, the impossible differential cryptanalysis often regards that obtaining one wrong key suggestion reduces the subkey space by a factor of

$$1 - \frac{1}{2^{32}}. \quad (4.130)$$

With this metric, the size of the remaining key space after deriving 1 wrong key suggestion is represented as

$$2^{32} \cdot \left(1 - \frac{1}{2^{32}}\right). \quad (4.131)$$

After discarding  $r$  wrong key suggestions, the size of the remaining key space becomes

$$2^{32} \cdot \left(1 - \frac{1}{2^{32}}\right)^r. \quad (4.132)$$

With  $2^{32}$  wrong key suggestions, the size of the remaining key space is not sufficiently reduced as shown below:

$$2^{32} \cdot \left(1 - \frac{1}{2^{32}}\right)^{2^{32}} \approx 2^{32} \cdot \frac{1}{e} \approx 2^{30.56}. \quad (4.133)$$

When the number of wrong key suggestions is  $2^{32} \cdot r$ , the size of the remaining key space becomes

$$2^{32} \cdot \left(1 - \frac{1}{2^{32}}\right)^{2^{32} \cdot r} \approx 2^{32} \cdot \left(\frac{1}{e}\right)^r. \quad (4.134)$$

Equation (4.134) takes  $2^{8.92}$  for  $r = 16$  and  $2^{-14.16}$  for  $r = 32$ . Therefore, the size of the key space is reduced to 1 by obtaining  $2^{32} \cdot 32 = 2^{37}$  wrong key suggestions. The suitable choice of the number of plaintexts pairs,  $N$ , can be obtained by the following inequation:

$$2^{N-24} \geq 2^{37}, \quad (4.135)$$

which concludes that  $N = 61$  is sufficient, namely, that the number of plaintext pairs should be  $2^{61}$ .

**Exercise 4.14** In the evaluation with Equation (4.134), the value of  $r$  does not have to be restricted to a multiple of 2 but can be any integer. Calculate the minimum number of  $r$  that makes the remaining key space less than 1.

**Algorithm 4.7** Impossible Differential Attack against AES Reduced to 5 Rounds

**Input:**  $\Delta P$  and  $\Delta C$  including a impossible differential characteristic,  $N = 2^{61}$

**Output:**  $sk_0[0, 5, 10, 15]$

**Initialization**

- 1: Initialize a 32-bit counter  $\kappa[i]$  to 1 where  $i = 0, 1, \dots, 2^{32} - 1$ ;
- 2: **for** 255 non-zero differences at  $S_1^{\text{MC}}$  **do**
- 3:   Compute  $\Delta S_1^{\text{SB}} \leftarrow \text{SR}^{-1} \circ \text{MC}^{-1}(\Delta S_1^{\text{MC}})$ ;
- 4:   Store  $\Delta S_1^{\text{SB}}$  in a look-up table  $T[i]$ , where  $i = 1, 2, \dots, 255$ ;
- 5: **end for**

**Collecting Pairs**

- 6: Choose  $2^{61}$  plaintext pairs  $(P_i, P'_i)$  for  $i = 1, 2, \dots, 2^{61}$  such that  $P_i \oplus P'_i = \Delta P$ ;
- 7: **for**  $i \leftarrow 1, 2, \dots, 2^{61}$  **do**
- 8:   Query  $P_i$  and  $P'_i$  to obtain  $C_i$  and  $C'_i$ ;
- 9:   Compute  $\Delta C_i \leftarrow C_i \oplus C'_i$ ;
- 10:   **if**  $\Delta C_i$  is 0 in all the 4 inactive bytes specified in  $\Delta C$  **then**
- 11:     Store the plaintext pair in a list  $L$ ;
- 12:   **end if**
- 13: **end for**
- 14: Label  $2^{29}$  expected pairs in the list  $L$  as  $(C_i, C'_i)$  for  $i = 1, 2, \dots, 2^{29}$ ;

**Recovery of  $sk_0[0, 5, 10, 15]$** 

- 15: **for**  $i \leftarrow 1, 2, \dots, 2^{29}$  **do**
- 16:   **for**  $j \leftarrow 1, 2, \dots, 255$  **do**
- 17:     Find all solutions of 4 active S-boxes satisfying  $\Delta P \xrightarrow{\text{SB}} T[j]$  in round 1, which are denoted by  $v_1, v_2, v_3, v_4$ ;
- 18:     **for** all combinations of  $v_1, v_2, v_3, v_4$  **do**
- 19:        $\text{tmp} \leftarrow P[0, 5, 10, 15] \oplus (v_1, v_2, v_3, v_4)$ ;
- 20:        $\kappa[\text{tmp}] \leftarrow 0$ ;
- 21:     **end for**
- 22:   **end for**
- 23: **end for**
- 24: **return** the index of  $\kappa$  whose entry is still 1;

**4.3.3.4 Attack Procedure for Recovering  $sk_0[0, 5, 10, 15]$** 

The description of the attack in an algorithmic form is given in Algorithm 4.7.

**4.3.3.5 Complexity Evaluation**

In Algorithm 4.7, queries are made only in step 8, which makes queries of  $2^{61}$  pairs of chosen plaintexts. The time complexity for the initialization part is at most  $2^{32}$ , for collecting pairs is  $2^{61}$  XOR operations, and for recovery of  $sk_0[0, 5, 10, 15]$  is at most  $2^{37}$ . Hence, the entire time complexity is  $2^{32} + 2^{61} + 2^{37} \approx 2^{61}$  XOR computations, which is less than  $2^{61}$  five-round AES computations. The memory complexity is  $2^{32}$  32-bit values for  $\kappa$ ,  $2^8$  32-bit values for  $T[i]$ , and  $2^{29}$  pairs of plaintexts for  $L$ . Therefore, the entire memory complexity is less than  $2^{32}$  128-bit

values. In summary, the attack complexity for recovering  $sk_0[0, 5, 10, 15]$  is as follows:

$$(Data, Time, Memory) = (2^{62}, 2^{61}, 2^{32}). \quad (4.136)$$

#### 4.3.3.6 Recovering Other Bytes of $sk_0$

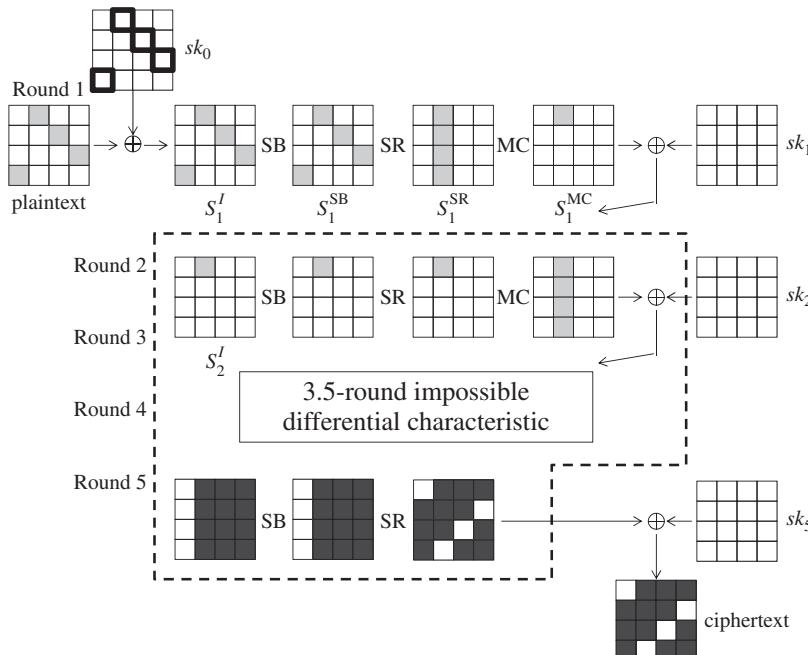
After 4 bytes of  $sk_0[0, 5, 10, 15]$  are recovered, the attacker needs to recover the other 12 bytes, or 96 bits. The straightforward method is performing the exhaustive search on the unknown 96 bits. However, this requires  $2^{96}$  computations, which is much higher than the complexity of the recovery of  $sk_0[0, 5, 10, 15]$ . Although it is already faster than the original exhaustive search on 128 bits, the attack complexity can be further optimized.

The strategy is to change the 3.5-round impossible differential characteristic so that the other 4 bytes of subkey  $sk_0$  can be recovered. Recall Figure 4.26. For the input difference of the 3.5-round impossible differential characteristic, the attacker can choose any active-byte position. Then, the attacker chooses the byte position 4 as the active-byte position, that is,

$$\Delta S_2^I = (0, 0, 0, 0, \Delta, 0, 0, 0, 0, 0, 0, 0, 0, 0). \quad (4.137)$$

This modification impacts the five-round key recovery attack. In detail, the active-byte positions of the plaintext become [3, 4, 9, 14] as depicted in Figure 4.31.

Consider applying the same attack as the subkey recovery for  $sk_0[0, 5, 10, 15]$  with slightly modifying the active-byte positions of the plaintext according to the 4 bytes in Figure 4.31. The attack procedure is almost the same as the one for  $sk_0[0, 5, 10, 15]$ , and thus the details



**Figure 4.31** Key recovery attack for five-round AES with different active-byte positions

are omitted. However, in this time, 4 bytes of  $sk_0[3, 4, 9, 14]$  are recovered instead of  $sk_0[0, 5, 10, 15]$ . The attack complexity to recover 4 bytes of  $sk_0[3, 4, 9, 14]$  is also the same as the one for  $sk_0[0, 5, 10, 15]$ , which is  $(Data, Time, Memory) = (2^{62}, 2^{61}, 2^{32})$ .

By applying the same discussion, using the following  $\Delta S_2^I$ , the other 4 bytes of  $sk_0[2, 7, 8, 13]$  can be recovered with the same complexity.

$$\Delta S_2^I = (0, 0, 0, 0, 0, 0, 0, \Delta, 0, 0, 0, 0, 0, 0). \quad (4.138)$$

After 12 bytes of  $sk_0$  are recovered, the remaining 4 bytes of  $sk_0[1, 6, 11, 12]$  can also be recovered by changing the active-byte position of  $\Delta S_2^I$  to the first low of the last column. However, applying the exhaustive search on the remaining 4 bytes is faster than iterating the same analysis one more time. In the end, the last 4 bytes can be recovered only with  $2^{32}$  computations without any additional data and memory complexities.

In summary, 12 bytes of  $sk_0$  are recovered by iterating the analysis on 4 bytes three times. This increases the data and time complexities three times compared to the 4-byte recovery. The required memory amount does not change because the memory used to recovery the first 4 bytes can be reused during the analysis on the other 4 bytes. After the 12 bytes are recovered, the cost of the exhaustive search on the remaining 4 bytes is negligibly smaller compared to the 12-byte recovery. The entire attack complexity becomes as follows:

$$(Data, Time, Memory) = (3 \times 2^{62}, 3 \times 2^{61}, 2^{32}). \quad (4.139)$$

#### 4.3.3.7 Structure Technique to Reduce the Number of Queries

Let us go back to the 4-byte subkey recovery of  $sk_0[0, 5, 10, 15]$  described in Figure 4.28. The attack was first explained under the assumption that the plaintext difference in the 4-byte positions  $[0, 5, 10, 15]$  was fixed to a single choice. Under this assumption, the attack requires to query  $2^N = 2^{61}$  chosen plaintext pairs.

There is a widely known technique called **structure** to reduce the data complexity by relaxing this assumption. Remember that the attack principally can work for any differences in the active 4-byte positions  $[0, 5, 10, 15]$ , that is, the differential form is as follows:

$$\Delta P = (\Delta, 0, 0, 0, 0, \Delta, 0, 0, 0, 0, \Delta, 0, 0, 0, 0, \Delta). \quad (4.140)$$

The idea of the structure technique is using a set of plaintexts that is closed with respect to the XOR operation. More precisely, the following set  $\mathcal{X}$  consisting of  $2^{32}$  plaintexts is considered:

$$\mathcal{X} \triangleq (A, C, C, C, C, A, C, C, C, C, A, C, C, C, C, A), \quad (4.141)$$

where 12 bytes, denoted by “C” (for constant), are fixed to a prespecified value, say 0, and 4 bytes, denoted by “A” (for active), take all the  $2^{32}$  possibilities. Suppose that all the  $2^{32}$  plaintexts included in the set  $\mathcal{X}$  are queried to the encryption oracle. Then, how many plaintext pairs following the differential form of Equation (4.140) are generated? The difference of any two plaintexts can satisfy the differential form of Equation (4.140) but for the small probability that the difference of at least 1 active byte happens to 0. Here, for the sake of simplicity, such a small probability event is ignored. The number of plaintext pairs generated from  $2^{32}$  plaintexts is

$$\binom{2^{32}}{2} \approx \frac{2^{64}}{2} = 2^{63}. \quad (4.142)$$

Hence, up to  $2^{63}$  pairs can be generated only with  $2^{32}$  queries. This is much better than the single difference case that requires  $2^{N+1}$  queries to obtain  $2^N$  pairs.

In the case of  $sk_0[0, 5, 10, 15]$  recovery attack,  $2^N = 2^{61}$  plaintext pairs are required. Taking  $2^{31}$  plaintexts from the set  $\mathcal{X}$  and making  $2^{31}$  queries is sufficient. This reduces the data complexity of the  $sk_0[0, 5, 10, 15]$  in Equation (4.136) from  $2^{62}$  to  $2^{31}$ . Moreover, the time complexity of the collecting pair part becomes at most  $2^{31}$ , which is no longer a bottleneck of the attack. The bottleneck is  $2^{37}$  complexity for obtaining the wrong key suggestions. In summary, the attack complexity in Equation (4.136) is updated as follows:

$$(Data, Time, Memory) = (2^{31}, 2^{37}, 2^{32}). \quad (4.143)$$

Similarly, the attack complexity for recovering all the 16 bytes is improved as follows:

$$(Data, Time, Memory) = (3 \times 2^{31}, 3 \times 2^{37}, 2^{32}). \quad (4.144)$$

**Exercise 4.15** Replace the 3.5-round impossible differential characteristic in the five-round attack (Figure 4.28) with the one described in Figure 4.26.

- (a) Explain the reason why the same attack strategy as the above five-round key recovery attack cannot work.
- (b) Show another example of 3.5-round impossible differential characteristic achieving the five-round key recovery attack in which
  1. the active-byte position in the input state is the same as the one in Figure 4.26, and
  2. the number of active bytes in the output state is minimized.

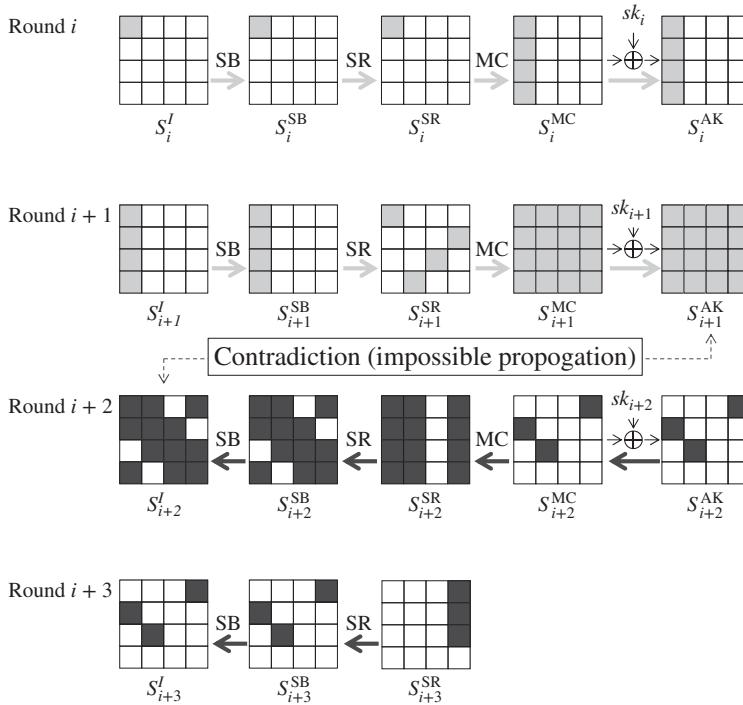
#### 4.3.4 Key Recovery Attacks for Seven-Round AES †

The impossible differential cryptanalysis can be extended to seven rounds. The essential difference from the five-round attack is that the 3.5-round impossible differential characteristic is carefully chosen so that the attack complexity can be below  $2^{128}$ . Compared to the five-round attack, more optimization techniques are introduced. In particular, several input and output differences are considered simultaneously to minimize the data complexity, and the round function for the key recovery is equivalently transformed into another representation to minimize the time complexity.

##### 4.3.4.1 Base of Impossible Differential Characteristics

Several 3.5-round impossible differential characteristics are used for the seven-round attack. One of the choices is shown in Figure 4.32. This is useful to understand the concept of the multiple impossible differential characteristics that will be explained later. Hereafter, the impossible differential characteristic in Figure 4.32 is called **basic impossible characteristic**.

The input difference  $\Delta x$  to the basic impossible characteristic is the same as the previously used ones, that is, it has nonzero difference only in 1 byte. In Figure 4.32, the byte position 0



**Figure 4.32** 3.5-Round basic impossible differential characteristic for seven-round attack

is chosen. The input difference  $\Delta x$  is described as follows:

$$\Delta x = (\Delta, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0). \quad (4.145)$$

The output difference  $\Delta y$  from the basic impossible characteristic has nonzero difference in 3 bytes of a single column. The active column can be located in any column position, and 3 active bytes can be located in any byte position inside the selected column. In Figure 4.32, the rightmost column is selected and the first 3 bytes are active. The output difference  $\Delta y$  is described as follows.

$$\Delta y = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \Delta', \Delta', \Delta', 0), \quad (4.146)$$

where  $\Delta'$  represents a nonzero difference independent of  $\Delta$ .

The detailed proof of the impossibility of the differential propagation  $\Delta x \xrightarrow{3.5R} \Delta y$  is omitted. In short,  $\Delta x$  makes all bytes active in two rounds, while 4 bytes are ensured to be inactive after 1.5-round decryption from  $\Delta y$ .

#### 4.3.4.2 Multiple Impossible Differential Characteristics

The concept of the **multiple impossible differential characteristics** considers a set of input difference  $\Delta x$ -set and a set of output difference  $\Delta y$ -set such that the differential propagation from any element in the  $\Delta x$ -set to any element in the  $\Delta y$ -set is impossible.

In the impossible differential cryptanalysis, key guesses generating input and output differences for the impossible differential characteristic are discarded. Hence, by considering multiple input and output differences, wrong key values can be discarded more quickly than the single difference case.

In the basic impossible differential characteristic, the active-byte position of  $\Delta x$  is fixed to the byte position 0. Here, even if the active-byte position of  $\Delta x$  is modified, the modified  $\Delta x$  and  $\Delta y$  are impossible differential transition. This property allows the attacker to consider the  $\Delta x$ -set consisting of the 16 differences. It does not mean that all of the 16 elements in the  $\Delta x$ -set can contribute to reducing the data complexity. Each element can reduce the data complexity only when they can speed up wrong key suggestions in the subkey recovery part. The seven-round attack uses the following  $\Delta x$ -set consisting of four input differences:

$$\begin{aligned}\Delta x\text{-set} \triangleq & \{(\Delta, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0), \\ & (0, \Delta, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0), \\ & (0, 0, \Delta, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0), \\ & (0, 0, 0, \Delta, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)\}.\end{aligned}\quad (4.147)$$

The active-byte positions of  $\Delta y$  can also be modified. As long as 1 byte is inactive in  $\Delta y$ , it forms the impossible differential propagation against any element in the  $\Delta x$ -set. The seven-round attack uses the following  $\Delta y$ -set consisting of four output differences:

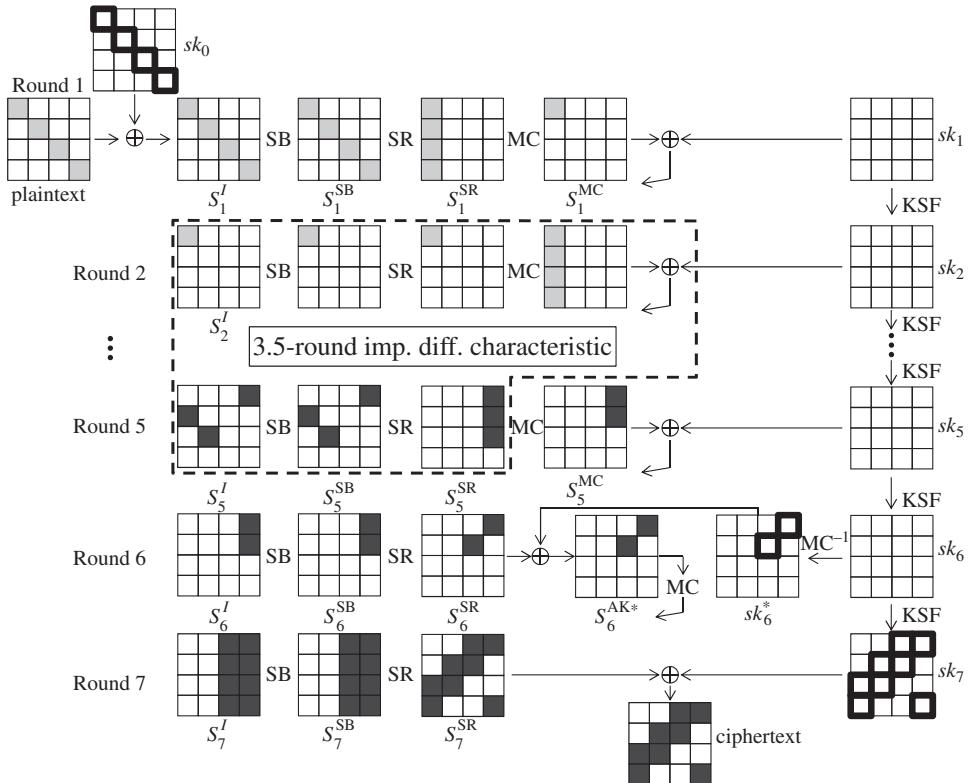
$$\begin{aligned}\Delta y\text{-set} \triangleq & \{(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \Delta', \Delta', \Delta', 0), \\ & (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \Delta', \Delta', 0, \Delta'), \\ & (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \Delta', 0, \Delta', \Delta'), \\ & (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \Delta', \Delta', \Delta')\}.\end{aligned}\quad (4.148)$$

**Exercise 4.16** Verify that any element in the  $\Delta x$ -set and  $\Delta y$ -set forms a 3.5-round impossible differential characteristic.

#### 4.3.4.3 Appending Subkey Recovery Part

To attack seven rounds, one round and two rounds are added before and after the 3.5-round characteristic, respectively. This locates the 3.5-round characteristic from state  $S_2^I$  to state  $S_5^{SR}$ . The entire structure is shown in Figure 4.33. It assumes the basic impossible differential characteristic in the middle 3.5 rounds. The added rounds can be used for any choice of the input and output differences in the  $\Delta x$ -set and  $\Delta y$ -set.

The backward extension by one round is straightforward and the same as the one in the five-round attack shown in Figure 4.28. It derives 4 active bytes in the plaintext, and 4 bytes of subkey  $sk_0$  are related to the formulation of the impossible differential characteristic from the plaintext difference. The forward extension by two rounds is more complicated. Given



**Figure 4.33** Key recovery attack for seven-round AES

the difference  $\Delta S_5^{SR}$ , the difference  $\Delta S_5^{MC}$  after the next MixColumns operation can take the following three possibilities.

1. The number of active bytes in the active column of  $\Delta S_5^{MC}$  is 2.
2. The number of active bytes in the active column of  $\Delta S_5^{MC}$  is 3.
3. The number of active bytes in the active column of  $\Delta S_5^{MC}$  is 4.

The seven-round attack restricts the form of  $\Delta S_5^{SR}$  to the case 1 for minimizing the number of involved subkey bytes to be recovered. Indeed by allowing more than 2 active bytes for  $\Delta S_5^{MC}$ , the attack complexity exceeds  $2^{128}$ , and thus cannot be faster than the exhaustive search for the original 128-bit key. In Figure 4.33, the active-byte positions are the first 2 bytes of the active column.

After  $\Delta S_5^{MC}$  is set to case 1, the remaining differential propagation until the ciphertext is straightforward. It derives two fully active columns after the MixColumns operation in round 6 and then two fully active diagonals at the ciphertext. In the end, the byte positions 2, 3, 5, 6, 8, 9, 12, 15 of the ciphertext are active.

How many subkey bytes are related to the formulation of the multiple impossible differential characteristics from the ciphertext difference? To minimize the attack complexity, the number of related subkey bytes needs to be minimized as much as possible. The seven-round attack

requires to introduce another technique for this purpose, which is an **equivalent transformation of the subkey addition**. Note that the seven-round structure in Figure 4.33 has already applied the technique in round 6.

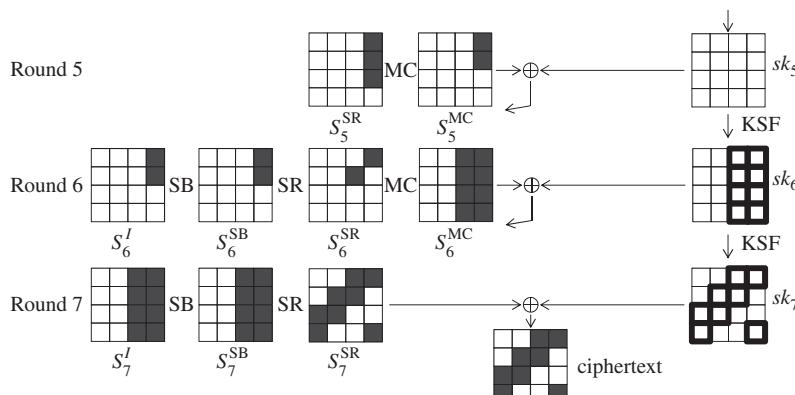
#### 4.3.4.4 Equivalent Transformation of Subkey Addition

This technique represents the computations of the AES round function in an alternative method. In particular, the order of the linear computations (ShiftRows, MixColumns, AddRoundKey) is exchanged.

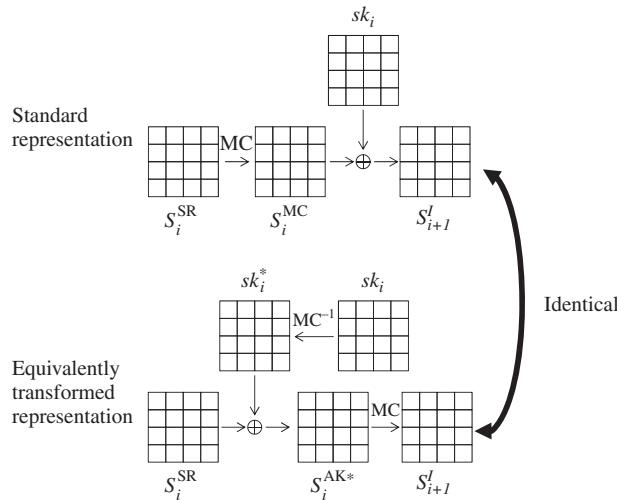
To understand the motivation of introducing this technique, it is useful to check a simple extension of two rounds to the end of the 3.5-round distinguisher, and simulate how the subkey recovery part would work. The computation structure is depicted in Figure 4.34. To derive wrong key suggestions, the attacker guesses part of subkeys  $sk_7$  and  $sk_6$ , and checks if the difference after the partial decryption up to  $S_5^{\text{SR}}$  satisfies any of the output difference in the  $\Delta y$ -set. If the number of guessed subkey bytes is large, the attack complexity is also large. Thus, keeping the number of guessed subkey bytes low is important for the attacker.

Owing to the linearity of the MixColumns and AddRoundKey operations,  $\Delta S_5^{\text{SR}}$  can be computed by  $\text{MC}^{-1}(\Delta S_6^I)$ , which implies that any value of  $sk_5$  is not related to the partial decryption up to  $\Delta S_5^{\text{SR}}$ , thus the attacker does not have to guess the value of  $sk_5$ . Here, how many subkey bytes need to be guessed to compute  $\Delta S_6^I$  from the ciphertext difference  $\Delta C$ ? In a simple method, 8 bytes of  $sk_7$  and 8 bytes of  $sk_6$ , in total 16 bytes, need to be guessed as stressed in Figure 4.34. However, guessing 16 bytes requires  $2^{128}$  trials, which is the same cost as the exhaustive search of the original 128-bit key. The motivation of the equivalent transformation is to avoid guessing 8 bytes of  $sk_6$ . Indeed, the number of guessed subkey bytes can be reduced to 2 by guessing linearly converted value of  $sk_6$  as implied in Figure 4.33.

The idea to reduce the number of guessed subkey bytes is changing the place of the AddRoundKey operation. More precisely, the attacker exchanges the order of the MixColumns and AddRoundKey operations so that the subkey addition is performed before the number of active bytes increases with the MixColumns operation. Because both operations are linear, such a transformation is possible. In Figure 4.35, the transformation is detailed. The top figure describes that the MixColumns operation is applied to state  $S_i^{\text{SR}}$ , and then the subkey



**Figure 4.34** Two-round simple extension after the distinguisher



**Figure 4.35** Equivalent transformation of subkey addition. The order of MixColumns and AddRoundKey is exchanged

$sk_i$  is XORed to the state. The bottom figure applies a slightly different operation, in which the inverse MixColumns operation is applied to the subkey, and the transformed subkey is XORed to the state before the MixColumns operation. Both computations result in the same state  $S_{i+1}^I$ .

To verify the equivalence of two computations, converting the bottom computation to the top one is easier. The bottom computation in Figure 4.35 is represented as follows:

$$S_{i+1}^I = \text{MC} (S_i^{\text{SR}} \oplus \text{MC}^{-1}(sk_i)) . \quad (4.149)$$

For any linear function LF and two input values  $x, y$ , the value of  $\text{LF}(x \oplus y)$  is equal to the value of  $\text{LF}(x) \oplus \text{LF}(y)$ . Therefore,

$$S_{i+1}^I = \text{MC}(S_i^{\text{SR}}) \oplus \text{MC} (\text{MC}^{-1}(sk_i)) \quad (4.150)$$

$$= \text{MC}(S_i^{\text{SR}}) \oplus sk_i, \quad (4.151)$$

which is equivalent to the top computation in Figure 4.35. As shown in Figure 4.35, hereafter the internal state after XOR of  $\text{MC}^{-1}(sk_i)$  and before the MixColumns operation is denoted by  $S_i^{\text{AK}*}$ .

Finally, the last 1.5 rounds of the seven-round attack become as shown in Figure 4.33. By guessing the 8 bytes of  $sk_7[2, 3, 5, 6, 8, 9, 12, 15]$  and 2 bytes of  $sk_6^*[9, 12]$ , the corresponding difference at  $S_5^{\text{SR}}$  can be computed from each ciphertext pair.

#### 4.3.4.5 Attack Procedure for Recovering Subkey

##### Collecting Plaintext Pairs

The attack first collects many pairs of plaintexts and the corresponding ciphertexts whose differences satisfy the form of  $\Delta P$  and  $\Delta C$  in Figure 4.33 by using the structure technique.

As previously explained, a structure  $\mathcal{X}$  that consists of  $2^{32}$  plaintexts is generated for a fixed 12-byte constant  $C$ :

$$\mathcal{X} \triangleq (A, C, C, C, C, A, C, C, C, C, A, C, C, C, C, A). \quad (4.152)$$

Then,  $2^{63}$  plaintext pairs are generated per structure.

Up to  $2^{12 \cdot 8} = 2^{96}$  structures can be constructed by changing the value of 12-byte constant  $C$ . Let  $2^N$  be the number of generated structures. Then,  $2^{N+63}$  plaintext pairs are generated with  $2^{N+32}$  queries. The exact value of  $N$  is later determined so that the entire attack complexity is optimized.

The probability that a randomly chosen pair has the difference  $\Delta C$  at the ciphertext is  $2^{-64}$  because inactive 8 bytes (64 bits) must have difference zero. Thus, the number of pairs satisfying  $\Delta C$  is

$$2^{N+63} \cdot 2^{-64} = 2^{N-1}. \quad (4.153)$$

### **Deriving Wrong Key Suggestions by Constructing Look-Up Table**

From each of the  $2^{N-1}$  obtained pairs, the attacker derives wrong key suggestions about the following 14 subkey bytes ( $sk_0[0, 5, 10, 15], sk_6^*[9, 12], sk_7[2, 3, 5, 6, 8, 9, 12, 15]$ ). Wrong key suggestions are efficiently derived by generating the look-up table  $T$ .

As explained in Figure 4.29, a wrong key suggestion for  $sk_0[0, 5, 10, 15]$  is obtained for each 1 active-byte difference of  $\Delta S_1^{\text{MC}}$  by checking the solutions of the differential propagation  $\Delta P \xrightarrow{\text{SB}} \Delta S_1^{\text{SB}}$ . For each element in the  $\Delta x$ -set, there are 255 choices of  $\Delta S_1^{\text{MC}}$ . Because the  $\Delta x$ -set includes 4 elements, there are  $4 * 255 \approx 2^{10}$  choices of  $\Delta S_1^{\text{MC}}$  in this attack.

With the same reason, a wrong key suggestion for  $sk_7[2, 3, 5, 6, 8, 9, 12, 15]$  is obtained for each 2 active-byte difference of  $\Delta S_6^{\text{AK}*}[9, 12]$ . The attacker, without the knowledge of the key, can compute the difference  $\Delta S_7^{\text{SB}}$  from ciphertext difference for each pair. For each 2 active-byte difference of  $\Delta S_6^{\text{AK}*}$ , the corresponding  $\Delta S_7^I$  is computed. By checking the solutions of the differential propagation  $\Delta S_7^I \xrightarrow{\text{SB}} \Delta S_7^{\text{SB}}$ , the value of the 8 active bytes is fixed, and thus the corresponding 8-byte value of  $sk_7$  is obtained. After 8 bytes of  $sk_7$  are recovered, each ciphertext pair can be decrypted until 2 bytes of  $S_6^{\text{AK}*}[9, 12]$ .

Similarly, a wrong key suggestion for  $sk_6^*[9, 12]$  can be efficiently derived from the 2-byte difference  $\Delta S_5^{\text{MC}}[12, 13]$  and 2 byte values and their difference of  $S_6^{\text{AK}*}[9, 12]$ . Here, 2-byte difference  $\Delta S_5^{\text{MC}}[12, 13]$  cannot be chosen from all the  $2^{16}$  possibilities. The 2-byte difference  $\Delta S_5^{\text{MC}}[12, 13]$  must be chosen so that  $\text{MC}^{-1}(\Delta S_5^{\text{MC}}[12], \Delta S_5^{\text{MC}}[13], 0, 0)$  is included in the  $\Delta y$ -set, that is, the result has zero difference in 1 byte position. The explanation for the basic impossible differential characteristic is given below, in which the inactive byte is located in  $\Delta S_5^{\text{SR}}[15]$ . By looking inside the inverse MixColumns operation, the difference of  $\Delta S_5^{\text{SR}}[15]$  is computed as follows:

$$\Delta S_5^{\text{SR}}[15] = (\text{B} \cdot \Delta S_5^{\text{MC}}[12]) \oplus (\text{D} \cdot \Delta S_5^{\text{MC}}[13]) \oplus (\text{9} \cdot 0) \oplus (\text{E} \cdot 0). \quad (4.154)$$

For each  $2^8$  choice of  $\Delta S_5^{\text{MC}}[12]$ , there is only one choice of  $\Delta S_5^{\text{MC}}[13]$  to satisfy  $\Delta S_5^{\text{SR}}[15] = 0$ . Therefore, the number of 2-byte difference  $\Delta S_5^{\text{MC}}[12, 13]$  satisfying  $\Delta S_5^{\text{SR}}[15] = 0$  is only 255  $\approx 2^8$ . Because the  $\Delta y$ -set contains four elements, the same discussion can be applied to the other three elements. In the end, the number of valid choices of  $\Delta S_5^{\text{MC}}[12, 13]$  is  $4 * 2^8 = 2^{10}$ . Once  $\Delta S_5^{\text{MC}}[12, 13]$  is fixed from those  $2^{10}$  choices,

the corresponding difference  $\Delta S_6^I[9, 12]$  is also fixed. By checking the solutions of the differential propagation  $\Delta S_6^I \xrightarrow{\text{SB}} \Delta S_6^{\text{SB}}$ , the value of the 2 active bytes is fixed, and thus the corresponding 2-byte value of  $sk_6^*$  is obtained.

In summary, the attacker first prepares the look-up table  $T$  consisting of  $2^{36}$  elements, which contains the information of 14-byte difference of  $\Delta S_1^{\text{SB}}[0, 5, 10, 15]$ ,  $\Delta S_7^I[8, 9, 10, 11, 12, 13, 14, 15]$ , and  $\Delta S_6^I[12, 13]$  for  $2^{10}$  choices of  $\Delta S_1^{\text{MC}}$ ,  $2^{16}$  choices of  $\Delta S_6^{\text{AK}*}[9, 12]$ , and  $2^{10}$  choices of  $\Delta S_5^{\text{MC}}[12, 13]$ .

#### **Evaluation of the Number of Structures**

From each of the  $2^{N-1}$  obtained pairs and each of the  $2^{36}$  choices in the look-up table  $T$ , a wrong key suggestion for the target 14 bytes is obtained. Therefore,  $2^{N+35}$  wrong key suggestions are obtained. If  $2^{N+35}$  suggestions are enough to reduce the 14-byte (112-bit) subkey space, the target 14 subkey bytes are recovered.

The proper choice of  $N$  is now evaluated. From Equation (4.134), with  $2^{N+35}$  wrong key suggestions, the size of the remaining key space for the 112-bit subkey bytes becomes

$$2^{112} \cdot \left(1 - \frac{1}{2^{112}}\right)^{2^{N+35}} \quad (4.155)$$

$$= 2^{112} \cdot \left(1 - \frac{1}{2^{112}}\right)^{2^{112} \cdot 2^{N-77}} \quad (4.156)$$

$$\approx 2^{112} \cdot \left(\frac{1}{e}\right)^{2^{N-77}}. \quad (4.157)$$

Equation (4.157) takes  $2^{19.67}$  for  $N = 83$  and  $2^{-72.67}$  for  $N = 84$ . Therefore, the 112-bit subkey space is reduced to 1 by obtaining all wrong key suggestions for  $2^N = 2^{84}$  structures.

#### **4.3.4.6 Complexity Evaluation**

The attack generates  $2^N = 2^{84}$  structures, in which each structure requires queries of  $2^{32}$  chosen plaintexts. Therefore, the data complexity of the attack is  $2^{84+32} = 2^{116}$  chosen plaintexts.

Regarding the computational cost, the attack needs to deal with  $2^{116}$  ciphertexts received by the encryption oracle for collecting pairs. The cost for generating look-up table  $T$  is around  $2^{36}$  computations, which is negligibly small compared to the other part. The attack derives  $2^{84+35} = 2^{119}$  wrong key suggestions. The cost for deriving one suggestion is about one-round encryption and two round decryptions, which is about  $3/7$  seven-round AES computations. Therefore, the cost for deriving  $2^{119}$  wrong key suggestions is less than  $2^{118}$  seven-round AES computations, which is the bottleneck of the computational cost.

The largest memory amount is for recording the validity/invalidity of each subkey value in the 112-bit subkey space, which is less than  $2^{112}$  128-bit state values.

In summary, the attack complexity for recovering 112-bit subkey value is as follows:

$$(Data, Time, Memory) = (2^{116}, 2^{118}, 2^{112}). \quad (4.158)$$

#### 4.3.4.7 Recovering Original Secret Key

The procedure for recovering the original 128-bit key is very simple. After the above subkey recovery, 64 bits of  $sk_7$  are recovered. The number of remaining unknown key bits of  $sk_7$  is 64 bits. The attacker can perform the exhaustive search on those 64 bits of  $sk_7$ . For each guess, the original 128-bit secret key  $sk_0$  can be computed with the inverse of the key schedule function, and the correctness of the guess can be confirmed with a pair of plaintext and ciphertext. The cost for the exhaustive search on the 64 bits of  $sk_7$  is around  $2^{64}$  seven-round AES computations, which is negligibly small compared to the 112-bit subkey recovery attack.

### 4.4 Integral Cryptanalysis

The origin of the integral cryptanalysis was the one proposed by Daemen *et al.* against Square block cipher. The design of AES is similar to the one for the Square block cipher. In this section, the integral cryptanalysis against AES is explained.

#### 4.4.1 Basic Concept

Remember that the differential cryptanalysis is motivated by the property that the difference of two values never changes by XORing an identical unknown secret value, and the propagation of the difference through the linear computations can be simulated with probability 1. Those properties allow a particular difference to appear with higher probability than other differences.

The motivation of the integral cryptanalysis is exploiting different properties that can hold with a high probability with respect to the frequently used computations in block ciphers.

AES is a byte-oriented cipher, which means that the computation inside the round function is defined bytewise. On the basis of this fact, the integral analysis considers processing a set of 256 plaintexts in which 1 byte takes all possibilities among 256 plaintexts, and all the other bytes are fixed to an identical value among 256 plaintexts. First of all, a set  $\mathcal{P}$  consisting of 256 plaintexts are defined as follows:

$$\begin{aligned} \mathcal{P} &\triangleq \{P_0, P_1, P_2, \dots, P_{255}\} \\ P_0 &= (0, c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8, c_9, c_{10}, c_{11}, c_{12}, c_{13}, c_{14}, c_{15}), \\ P_1 &= (1, c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8, c_9, c_{10}, c_{11}, c_{12}, c_{13}, c_{14}, c_{15}), \\ P_2 &= (2, c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8, c_9, c_{10}, c_{11}, c_{12}, c_{13}, c_{14}, c_{15}), \\ &\vdots \\ P_{255} &= (255, c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8, c_9, c_{10}, c_{11}, c_{12}, c_{13}, c_{14}, c_{15}), \end{aligned} \tag{4.159}$$

where each of  $c_1, c_2, \dots, c_{15} \in \{0, 1\}^8$  is a constant value fixed before the analysis. The set  $\mathcal{P}$  is also depicted in Figure 4.36. The set  $\mathcal{P}$  is an unordered set, that is, the order of each element in the set is not distinguished. For example, the sets  $\{P_0, P_1, P_2, \dots, P_{255}\}$  and  $\{P_1, P_0, P_2, \dots, P_{255}\}$  are regarded to be the same. The only important point is that each of  $P_i, i = 0, 1, 2, \dots, 255$  is included in the set  $\mathcal{P}$  exactly once.

$$\mathcal{P} = \{P_0, P_1, P_2, \dots, P_{255}\}$$

	$i$	$c_4$	$c_8$	$c_{12}$
$P_i$				
	$c_1$	$c_5$	$c_9$	$c_{13}$
	$c_2$	$c_6$	$c_{10}$	$c_{14}$
	$c_3$	$c_7$	$c_{11}$	$c_{15}$

**Figure 4.36** Basic set of plaintexts for integral cryptanalysis

$$\mathcal{P}^{\text{AK}} = \{P_0 \oplus sk_0, P_1 \oplus sk_0, P_2 \oplus sk_0, \dots, P_{255} \oplus sk_0\}$$

$$0 \leq i \leq 255$$



$i \oplus sk_0[0]$	$c_4 \oplus sk_0[4]$	$c_8 \oplus sk_0[8]$	$c_{12} \oplus sk_0[12]$
$c_1 \oplus sk_0[1]$	$c_5 \oplus sk_0[5]$	$c_9 \oplus sk_0[9]$	$c_{13} \oplus sk_0[13]$
$c_2 \oplus sk_0[2]$	$c_6 \oplus sk_0[6]$	$c_{10} \oplus sk_0[10]$	$c_{14} \oplus sk_0[14]$
$c_3 \oplus sk_0[3]$	$c_7 \oplus sk_0[7]$	$c_{11} \oplus sk_0[11]$	$c_{15} \oplus sk_0[15]$

$\mathcal{A}$	$C$	$C$	$C$
$C$	$C$	$C$	$C$
$C$	$C$	$C$	$C$
$C$	$C$	$C$	$C$

**Figure 4.37** Plaintexts set after XORing subkey  $sk_0$

In the integral cryptanalysis, the byte in which all values appear exactly once among all the texts in the set is called the **all property**, and is often denoted by  $\mathcal{A}$ . Similarly, the byte in which all texts in the set have an identical value is called the **constant property**, and is often denoted by  $\mathcal{C}$ . With those notations, the property of the set  $\mathcal{P}$  is written as  $(\mathcal{A}, \mathcal{C}, \mathcal{C})$ .

The set  $\mathcal{P}$  shows several interesting properties when it is processed by the computations adopted in AES under the same key, which will be detailed below.

#### 4.4.2 Processing $\mathcal{P}$ through Subkey XOR

Because the first operation of AES is an XOR of the plaintext and the first subkey, the status of the set  $\mathcal{P}$  after the subkey XOR is firstly analyzed. Namely, the subkey  $sk_0$  is XORed to each of the 256 plaintexts in the set  $\mathcal{P}$ . Here, it is assumed that the key value does not change for encrypting 256 plaintexts in the set. This updates the set  $\mathcal{P}$  to  $\mathcal{P}^{\text{AK}} \triangleq \{P_0 \oplus sk_0, P_1 \oplus sk_0, P_2 \oplus sk_0, \dots, P_{255} \oplus sk_0\}$ .  $\mathcal{P}^{\text{AK}}$  is also described in Figure 4.37.

Regarding the byte position  $j$ , where  $1 \leq j \leq 15$ , the 256 plaintexts originally have the same value  $c_j$ . Now the value is updated to  $c_j \oplus sk_0[j]$ . The important property is that  $c_j \oplus sk_0[j]$

is a fixed value, although its value is unknown to the attacker because of  $sk_0[j]$ . For example, the value of  $c_j \oplus sk_0[j]$  can be represented by another constant  $c'_j$ . Clearly, all the  $j$ th byte positions ( $1 \leq j \leq 15$ ) in the updated state satisfy the constant property.

Regarding the byte position 0, the 256 plaintexts originally vary to take all the 256 values (all property). By XORing an unknown constant  $sk_0[0]$  to all of the 256 texts, the property that all the 256 values appear among 256 texts still holds with probability 1. Remember that the set considered in the integral analysis is an unordered set. Thus, changing the order of elements inside the set does not affect the analysis. It is concluded that the byte position 0 in the update state satisfies the all property.

**Lemma 4.4.1** *By XORing an unknown or known constant for each of the texts in the set,*

- *the byte with all property still satisfies the all property, and*
- *the byte with constant property still satisfies the constant property.*

#### 4.4.3 Processing $\mathcal{P}$ through SubBytes Operation

The next operation of AES is the SubBytes operation, which applies the 8-bit to 8-bit S-box to each byte of the state. Here, the status of the set  $\mathcal{P}$  after the SubBytes operation is analyzed. This updates the set  $\mathcal{P}$  to  $\mathcal{P}^{\text{SB}} \triangleq \{S(P_0), S(P_1), S(P_2), \dots, S(P_{255})\}$ .  $\mathcal{P}^{\text{SB}}$  is also described in Figure 4.38.

Two properties of the S-box are related to the analysis.

- S-box is a bijective mapping from  $\{0, 1\}^8$  to  $\{0, 1\}^8$ .
- S-box is a deterministic mapping. It always maps the same input value to the same output value.

Regarding the byte positions  $j$ , where  $1 \leq j \leq 15$ , the 256 plaintexts originally have the same value  $c_j$ . Now the value is updated to  $S(c_j)$ . Because the S-box is a deterministic

$$\mathcal{P}^{\text{SB}} = \{\text{SB}(P_0), \text{SB}(P_1), \text{SB}(P_2), \dots, \text{SB}(P_{255})\}$$

$$0 \leq i \leq 255$$

$S(i)$	$S(c_4)$	$S(c_8)$	$S(c_{12})$
$S(c_1)$	$S(c_5)$	$S(c_9)$	$S(c_{13})$
$S(c_2)$	$S(c_6)$	$S(c_{10})$	$S(c_{14})$
$S(c_3)$	$S(c_7)$	$S(c_{11})$	$S(c_{15})$



$\mathcal{A}$	$C$	$C$	$C$
$C$	$C$	$C$	$C$
$C$	$C$	$C$	$C$
$C$	$C$	$C$	$C$

**Figure 4.38** Plaintexts set after the SubBytes operation

mapping,  $S(c_j)$  is a fixed value among all the 256 texts in the set. All the  $j$ th byte positions ( $1 \leq j \leq 15$ ) in the updated state satisfy the constant property.

Regarding the byte position 0, the 256 plaintexts originally vary to take all the 256 values (all property). Owing to the bijective property of the S-box, the output of the S-box also varies to take all the 256 values, although the order of the values is changed. It is concluded that the byte position 0 in the update state satisfies the all property.

**Lemma 4.4.2** *By applying the S-box for each of the texts in the set,*

- *the byte with all property still satisfies the all property, and*
- *the byte with constant property still satisfies the constant property.*

#### 4.4.4 Processing $\mathcal{P}$ through ShiftRows Operation

The ShiftRows operation of AES only exchanges the byte positions. It does not operate the data inside each byte. Because the integral analysis only exploits the property inside each byte, the ShiftRows operation never affects the property used in the integral cryptanalysis.

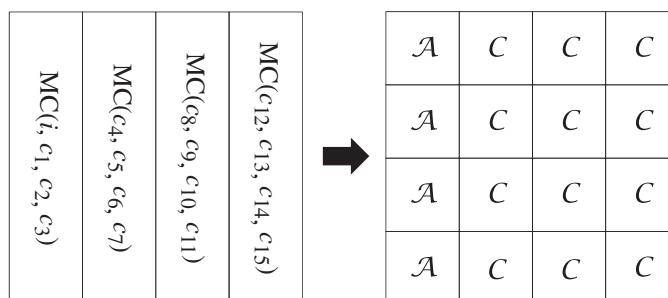
#### 4.4.5 Processing $\mathcal{P}$ through MixColumns Operation

Because the MixColumns operations is not closed in each byte, that is, it mixes the values of 4 bytes, the analysis is not simple in general. If there is at most 1 byte with the all property in each column, the analysis is simple. Here, the status of the set  $\mathcal{P}$  after the MixColumns operation is first analyzed. This updates the set  $\mathcal{P}$  to  $\mathcal{P}^{\text{MC}} \triangleq \{\text{MC}(P_0), \text{MC}(P_1), \text{MC}(P_2), \dots, \text{MC}(P_{255})\}$ .  $\mathcal{P}^{\text{MC}}$  is also described in Figure 4.39.

Regarding the second, third, and the fourth columns, the input value to the MixColumns operation is a fixed value among all the 256 texts. Therefore, the output of the MixColumns operation is also a fixed value among all the 256 texts. For all bytes in the second, third, and the fourth columns, the updated state satisfies the constant property.

$$\mathcal{P}^{\text{MC}} = \{\text{MC}(P_0), \text{MC}(P_1), \text{MC}(P_2), \dots, \text{MC}(P_{255})\}$$

$$0 \leq i \leq 255$$



**Figure 4.39** Plaintexts set  $\mathcal{P}$  after the MixColumns operation

Regarding the first column, according to the specification of the MixColumns operation, the 4 output bytes are computed as follows:

$$\begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} i \\ c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} 2i \\ i \\ i \\ 3i \end{bmatrix} \oplus \begin{bmatrix} 3c_1 \oplus c_2 \oplus c_3 \\ 2c_1 \oplus 3c_2 \oplus c_3 \\ c_1 \oplus 2c_2 \oplus 3c_3 \\ c_1 \oplus c_2 \oplus 2c_3 \end{bmatrix}. \quad (4.160)$$

The 4 output bytes are composed of the impact from  $i$  and the impact from the other three constant values ( $c_1, c_2, c_3$ ). Among the 256 texts, the impact from the constant values, which is the second term on the right-hand side of Equation (4.160), is a fixed value. As summarized in Lemma 4.4.2, XORing the constant does not change the all property and constant property. Hence, the property of the output value only depends on the impact from  $i$  (the first term on the right-hand side of Equation (4.160)). In the input 256 texts, the value of  $i$  varies to take all the 256 values (all property). This makes the values of  $i, 2i$ , and  $3i$  vary to take all the 256 values, although the order of the values changes. It is concluded that the 4 output bytes from the first column will have the all property.

The analysis cannot be simple when the number of bytes with the all property is more than 1. The analysis will be explained later when the integral property of AES three rounds is explained.

#### 4.4.6 Integral Property of AES Reduced to 2.5 Rounds

By using the earlier discussions, a certain property can be constructed until 2.5 rounds. The overall structure is given in Figure 4.40. In the following, the transition of the property is explained state by state.

##### Round 1

**Plaintexts:** The analysis starts from the set of plaintexts  $\mathcal{P}$ , which is defined in Equation (4.159).

**State  $S_1^I$ :** From Lemma 4.4.1, the subkey XOR maintains the all and constant properties.

**State  $S_1^{SB}$ :** From Lemma 4.4.2, the SubBytes operation maintains the all and constant properties.

**State  $S_1^{SR}$ :** The byte positions are moved according to the ShiftRows operation (no movement in the first row).

**State  $S_1^{MC}$ :** As shown in Figure 4.39, all bytes in the right three columns maintain the constant property and a single byte with the all property in the input state expands to 4 bytes in the first column.

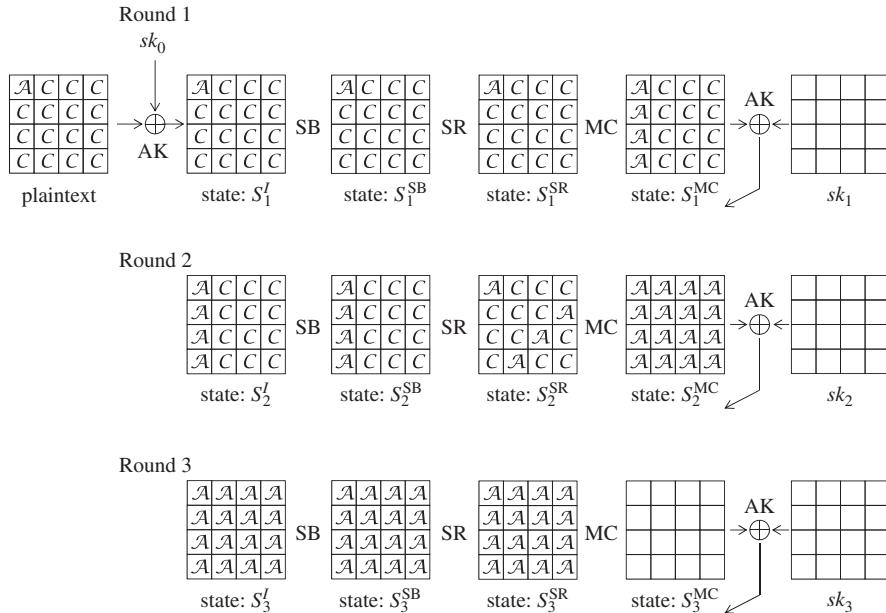
**State  $S_2^I$ :** From Lemma 4.4.1, the subkey XOR maintains the all and constant properties.

##### Round 2

**State  $S_2^{SB}$ :** From Lemma 4.4.2, the SubBytes operation maintains the all and constant properties.

**State  $S_2^{SR}$ :** The byte positions are moved according to the ShiftRows operation. All columns come to contain a byte with the all property.

**State  $S_2^{MC}$ :** As the same principle of the first column in Figure 4.39, for all columns, a single byte with the all property in the input state expands to 4 bytes in the first column. As a result, bytes with the constant property disappear and all bytes will show the all property.



**Figure 4.40** Integral property for 2.5-round AES

**State  $S_3^I$ :** From Lemma 4.4.1, the subkey XOR maintains the all property.

#### Round 2.5

**State  $S_3^{SB}$ :** From Lemma 4.4.2, the SubBytes operation maintains the all property.

**State  $S_3^{SR}$ :** The byte positions are moved according to the ShiftRows operation.

Finally, it was shown that the set of plaintexts  $\mathcal{P}$  will provide the all property in all 16 bytes after 2.5 rounds.

#### 4.4.7 Balanced Property

In Figure 4.40, the following question naturally rises.

Does any property remain after the MixColumns operation is applied to state  $S_3^{SR}$  in Figure 4.40?

Actually, the answer is “Yes.” However it is no longer the all property or the constant property. The computation from the state  $S_3^{SR}$  to  $S_3^{MC}$  is identical for all columns, thus analyzing that a single column is enough. Moreover, the analysis is almost the same among all the 4 output bytes of the column. Here, the property of the first output byte is analyzed in detail. According to the specification of the MixColumns operation,  $S_3^{MC}[0]$  is computed by the following equation:

$$S_3^{MC}[0] = 2 \cdot S_3^{SR}[0] \oplus 3 \cdot S_3^{SR}[1] \oplus S_3^{SR}[2] \oplus S_3^{SR}[3]. \quad (4.161)$$

Remember that the analysis initially starts from a set of 256 plaintexts  $P_i$ , where  $i = 0, 1, \dots, 255$ . Let  $S_{3,i}^{\text{SR}}[0, 1, 2, 3]$  be 4 input bytes to the MixColumns operation corresponding to  $P_i$ . Owing to the 2.5-round integral property, it is ensured that all of the 4 bytes  $S_{3,i}^{\text{SR}}[0, 1, 2, 3]$  show the all property, that is, the value of  $S_{3,i}^{\text{SR}}[0]$  takes all possibilities by collecting all  $i = 0, 1, \dots, 255$ , and the same is applied to  $S_{3,i}^{\text{SR}}[1], S_{3,i}^{\text{SR}}[2]$ , and  $S_{3,i}^{\text{SR}}[3]$ . The important fact here is that the all property for each of 4 bytes was derived independently from each other. In other words, no relation is guaranteed among the values of different byte positions in the same text.

There is some probability that 4 independent input bytes for different texts  $i_1$  and  $i_2$  will result in the same output value. Hence, the “all” property cannot be ensured for  $S_3^{\text{MC}}[0]$ . It is also easy to see that the constant property cannot be ensured for  $S_3^{\text{MC}}[0]$ .

The idea of the new property is computing the XOR sum of all the 256 texts, that is,  $\bigoplus_{i=0}^{255} S_{3,i}^{\text{MC}}[0]$ . The details of this computation are as follows:

$$\bigoplus_{i=0}^{255} S_{3,i}^{\text{MC}}[0] = \bigoplus_{i=0}^{255} (2 \cdot S_{3,i}^{\text{SR}}[0] \oplus 3 \cdot S_{3,i}^{\text{SR}}[1] \oplus S_{3,i}^{\text{SR}}[2] \oplus S_{3,i}^{\text{SR}}[3]) \quad (4.162)$$

$$= \bigoplus_{i=0}^{255} (2 \cdot S_{3,i}^{\text{SR}}[0]) \oplus \bigoplus_{i=0}^{255} (3 \cdot S_{3,i}^{\text{SR}}[1]) \oplus \bigoplus_{i=0}^{255} S_{3,i}^{\text{SR}}[2] \oplus \bigoplus_{i=0}^{255} S_{3,i}^{\text{SR}}[3] \quad (4.163)$$

$$= 0 \oplus 0 \oplus 0 \oplus 0 = 0. \quad (4.164)$$

In the third equality of the above transformation, the fact that each byte satisfies the all property and the fact that  $0 \oplus 1 \oplus \dots \oplus 255 = 0$  are used.

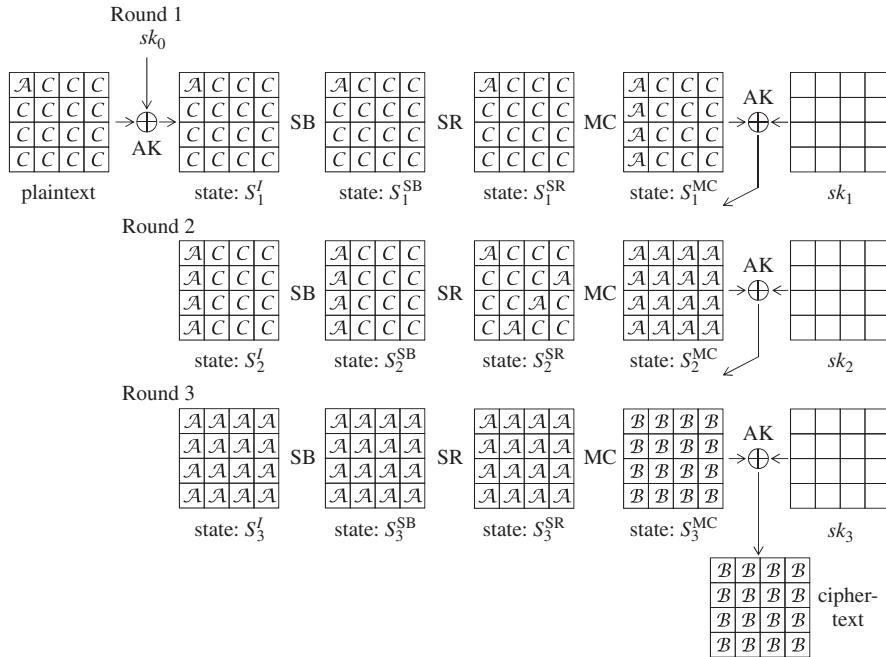
In summary,  $S_3^{\text{MC}}[0]$  satisfies the property that the XOR sum of all the 256 texts is 0, that is,  $\bigoplus_{i=0}^{255} S_{3,i}^{\text{MC}}[0] = 0$ . This property is called **balanced property**, and often denoted by  $\mathcal{B}$ . The same discussion can be applied to the other bytes of  $S_3^{\text{MC}}$ . Thus, all the bytes of  $S_3^{\text{MC}}$  satisfy the balanced property.

#### 4.4.8 Integral Property of AES Reduced to Three Rounds and Distinguishing Attack

By using the balanced property, the integral property for three-round AES can be constructed, which is shown in Figure 4.41.

From Figure 4.40, the MixColumns operation is applied to  $S_3^{\text{SR}}$ , making all the bytes of  $S_3^{\text{MC}}$  balanced. With the last AddRoundKey operation for  $sk_3$ , the XOR sum of each byte is further XORed by  $sk_3$  256 times. As long as the number of texts in the original set is an even number, the effect of XORing the same constant is canceled out. In the end, the last AddRoundKey operation does not break the balanced property, making all bytes of the ciphertext balanced.

The distinguishing attack is now easily constructed for AES three rounds in the chosen plaintext model. The goal of the attacker is identifying which of the AES three rounds or a random permutation is implemented by the oracle. The attacker prepares the set of 256 plaintexts  $\mathcal{P}$ , and queries all of the 256 plaintexts to the oracle. The attacker then computes the XOR sum of each byte of the received ciphertexts and checks whether the sum is 0 or not. If all the bytes satisfy the balanced property, the oracle implements the AES reduced to three rounds. Otherwise, it implements the random permutation.



**Figure 4.41** Integral property for three-round AES

---

#### Algorithm 4.8 Distinguishing Attack with Integral Cryptanalysis against 3-Round AES

---

**Input:** A variable  $\text{tmp}$  for counting the XOR sum of returned texts from the oracle

**Output:** Determining bit 0 (AES 3 rounds) or 1 (a random permutation)

- 1: Choose constant values  $c_1, c_2, \dots, c_{15}$  for the 15 bytes in the plaintext;
  - 2: Initialize  $\text{tmp}$  to 0;
  - 3: **for**  $i = 0, 1, \dots, 255$  **do**
  - 4:   Query  $P_i$  to the oracle and obtain the corresponding ciphertext  $C_i$ ;
  - 5:    $\text{tmp} \leftarrow \text{tmp} \oplus C_i$ ;
  - 6: **end for**
  - 7: **if**  $\text{tmp}$  is 0 in all bytes **then**
  - 8:   **return** 0;       // The oracle is AES 3 rounds.
  - 9: **end if**
  - 10: **return** 1;      // The oracle is random permutation.
- 

#### Attack Procedure

The attack procedure in the algorithmic form is described in Algorithm 4.8.

#### Complexity Evaluation

The attack makes 256 queries. Thus, the data complexity of this attack is 256 chosen plaintexts. The computational cost is updating  $\text{tmp}$  value 256 times, which is 256 XOR operations.

The required memory is only for storing `tmp` value, which is negligibly small. In summary, the complexity of this attack is  $(Data, Time, Memory) = (256, 256, negl.)$ , where “*negl.*” stands for negligible.

### **Success Probability**

As long as the oracle implements the AES three rounds, the attack successfully returns the determining bit 0 with probability 1.

When the oracle implements a random permutation, the XOR sum of each byte may happen to be 1 with a low probability. The probability that the XOR sum of randomly generated 256 byte values happens to be 0 is  $2^{-8}$ . The probability that this event happens for all the 16 bytes at the same time is  $2^{-8 \cdot 16} = 2^{-128}$ , which is negligibly small.

Hence, the distinguishing attack successfully distinguishes the AES three rounds from a random permutation.

#### **4.4.9 Key Recovery Attack with Integral Cryptanalysis for Five Rounds**

On the basis of the three-round integral property, a key recovery attack can be mounted against AES reduced to five rounds. The attack appends two rounds after the three-round integral property, which makes the structure shown in Figure 4.42. The attacker guesses the part of the last two subkeys  $sk_5$  and  $sk_4$  and performs the partial decryption up to  $S_4^I$ . The correct guess always achieves the balanced property in all bytes of  $S_4^I$ , which enables the attacker to reduce the key space.

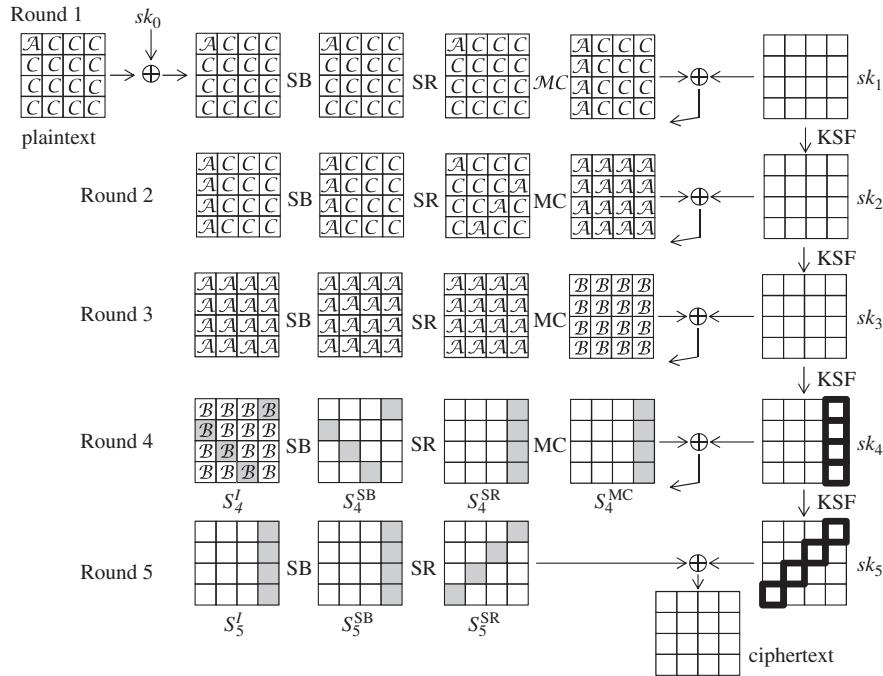
#### **Collecting Plaintexts**

The attacker prepares sets of 256 plaintexts  $\mathcal{P}$ . As explained later, the analysis guesses 64 bits of subkeys, and each set of 256 plaintexts  $\mathcal{P}$  can reduce the subkey space by a factor of  $2^{32}$ . In order to reduce the subkey space to 1, two sets of 256 plaintexts  $\mathcal{P}$  are required. Those  $2 \cdot 256 = 512$  plaintexts are passed to the encryption oracle, and the attacker obtains the corresponding two sets of 256 ciphertexts.

#### **Recovery of $sk_5$ and $sk_4$**

With the obtained two sets of 256 ciphertexts, the attacker recovers the values of  $sk_5$  and  $sk_4$ . The 4 bytes of  $sk_5[3, 6, 9, 12]$  and the 4 bytes of  $sk_4[12, 13, 14, 15]$ , in total 64-bit subkey space, are reduced to 32 bits with the analysis of the first set of 256 plaintexts. Then, the subkey space is further reduced to 1 by using the second set of 256 plaintexts. The guessed subkey bytes are marked by bold line in Figure 4.42.

With a guess of  $sk_5[3, 6, 9, 12]$  and  $sk_4[12, 13, 14, 15]$ , the attacker can compute the corresponding 4 bytes of  $S_5^{\text{SR}}$  by taking XOR of the ciphertext and the guess of  $sk_5[3, 6, 9, 12]$  for all the 256 ciphertexts in the first set. The inverse of the ShiftRows operation moves the computed byte positions into 12, 13, 14, 15, and the corresponding 4 bytes of  $S_5^I$  can be computed with the inverse of the S-box. The guessed  $sk_4[12, 13, 14, 15]$  allows the attacker to compute the corresponding 4 bytes of  $S_4^{\text{MC}}$ . Because all values in a column are computed, the corresponding 4 bytes of  $S_4^{\text{SR}}$  are computed with the inverse of the MixColumns operation. The next inverse of the ShiftRows operation moves the computed byte positions into 1, 6, 11, 12, and the corresponding 4 bytes of  $S_4^I$  can be computed with the inverse of the S-box.



**Figure 4.42** Key recovery attack against five-round AES. Guessed 4 bytes of  $sk_3$  and 4 bytes of  $sk_4$  are stressed by bold lines. With those guesses, several bytes of the internal state marked by light gray color can be computed

The value of  $S_4^I[1, 6, 11, 12]$  is computed for all the 256 texts in the first set. To check if the balanced property is satisfied, the XOR sum of all the 256 texts is computed for 4 bytes of  $S_4^I[1, 6, 11, 12]$ . If the results are 0 in all bytes, that is,

$$\bigoplus S_4^I[1] = \bigoplus S_4^I[6] = \bigoplus S_4^I[11] = \bigoplus S_4^I[12] = 0, \quad (4.165)$$

the subkey guess of  $sk_5[3, 6, 9, 12], sk_4[12, 13, 14, 15]$  is a correct subkey candidate. For this case, the guess is stored in a list  $\kappa$ . If at least 1 byte of  $S_4^I[1, 6, 11, 12]$  does not satisfy the balanced property, the subkey guess is wrong. For this case, the subkey guess is not stored and is discarded immediately.

The correct guess always satisfies Equation (4.165). Besides the correct guess, several wrong guesses happen to satisfy Equation (4.165) probabilistically. The probability that randomly chosen 4 byte values become 0 is  $(2^{-8})^4 = 2^{-32}$ . Therefore, with trying  $2^{64}$  guesses,  $2^{64} \cdot 2^{-32} = 2^{32}$  wrong guesses are stored in the correct subkey candidates list  $\kappa$ . In other words, with the analysis of the first set, the subkey space is reduced from  $2^{64}$  to  $2^{32}$ , by 32 bits.

The same analysis is iterated again by using the 256 ciphertexts in the second set. This time, the subkey guess of  $sk_5[3, 6, 9, 12], sk_4[12, 13, 14, 15]$  is chosen from the list  $\kappa$ , thus the

number of guesses is  $2^{32}$  rather than  $2^{64}$  in the analysis for the first set. The subkey space is reduced by another 32 bits, which makes the number of subkey candidates about 1.

### **Recovery of Original Key**

The same analysis can be performed to recover other subkey bytes of  $sk_5$  and  $sk_4$ . With iterating the analysis twice,

- 8 bytes of  $sk_5[2, 5, 8, 15], sk_4[8, 9, 10, 11]$  and
- 8 bytes of  $sk_5[1, 4, 11, 14], sk_4[4, 5, 6, 7]$

are recovered. As explained in the previous section, the original key  $sk_0$  can be recovered with any other subkey. The remaining unknown bytes of  $sk_4$  are the 4 bytes of  $sk_4[0, 1, 2, 3]$ . The attacker recovers those 4 bytes with the exhaustive search. Note that, if the subkey candidates for  $sk_4[12, 13, 14, 15], sk_4[8, 9, 10, 11]$ , and  $sk_4[4, 5, 6, 7]$  were not reduced to 1, but several candidates are remaining, the correct candidates can be guessed together with the exhaustive search of  $sk_4[0, 1, 2, 3]$ .

### **Attack Summary**

The attack is a chosen plaintext attack. The data complexity of this attack is  $2 \cdot 256 = 2^9$  chosen plaintexts. The computational cost for constructing two sets of 256 plaintexts is 512 computations, which is negligible compared to the key recovery part. For recovering  $sk_5[3, 6, 9, 12], sk_4[12, 13, 14, 15]$ , in the analysis of the first set, the two-round decryption is performed for each of the  $2^{64}$  subkey guesses and  $2^8$  ciphertexts in the set. Hence, the computational cost for analyzing the first set is  $2 \cdot 2^{64+8} = 2^{73}$  round function computations, which corresponds to  $2^{73}/5 \approx 2^{70.7}$  five-round AES computations. The computational cost for the second set is cheaper than the one for the first set by a factor of  $2^{32}$  because only about  $2^{32}$  candidates remain in  $\kappa$ . The computational cost for the recovery of  $sk_0$  is two more iterations of the analysis for different columns and the exhaustive search on 32 bits and several remaining candidates. The exhaustive search on 32 bits is much cheaper than the cost for the other part. As a result, the time complexity of this attack is  $3 \times 2^{70.7} \approx 2^{72.3}$  five-round AES computations.

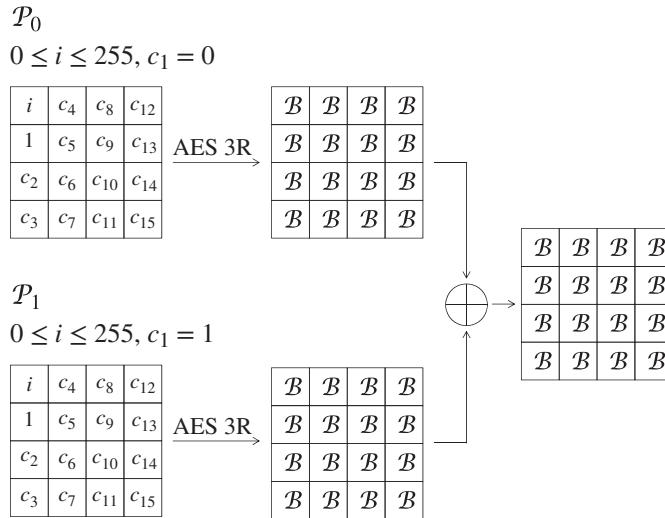
The reduced subkey space  $\kappa$  requires to store  $2^{32}$  8-byte information, which is equivalent to  $2^{31}$  AES states. The other part is negligibly small. As a result, the memory complexity of this attack is  $2^{31}$  AES states.

In summary, the complexity of this attack is  $(Data, Time, Memory) = (2^9, 2^{72.3}, 2^{31})$ .

#### **4.4.10 Higher-Order Integral Property †**

The integral property for three rounds only uses 256 plaintexts. A natural approach to improve the attack is increasing the number of plaintexts to attack more rounds. The approach is called **higher-order integral cryptanalysis**.

Remember that the values of the 15 bytes with the constant property in the set of plaintexts  $\mathcal{P}$  in Figure 4.36 can be fixed to any value. The higher-order integral cryptanalysis generates several sets of plaintexts by chaining the values of bytes with the constant property. Suppose



**Figure 4.43** Idea of the higher-order (second-order) integral property

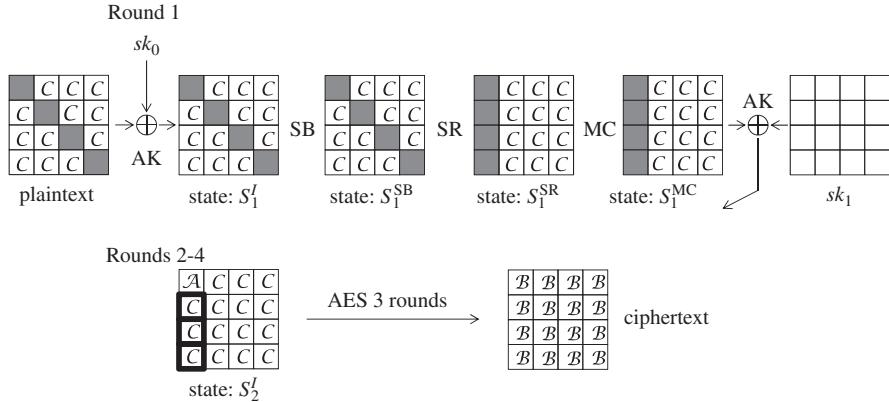
that two sets of 256 plaintexts  $\mathcal{P}_1$  and  $\mathcal{P}_2$  are generated. For the first set  $\mathcal{P}_1$ , the byte position 1 is fixed to 0. For the second set  $\mathcal{P}_2$ , the byte position 2 is fixed to 0. Then, those 512 plaintexts are queried to the oracle with AES reduced to three rounds, and the attacker obtains the corresponding 512 ciphertexts.

Suppose that the order of the 512 ciphertexts is mixed in an unknown manner to the attacker, that is, the attacker does not know which of the first or second set each ciphertext belongs to. What property can be ensured for 512 ciphertexts in this situation?

The higher-order integral property computes the XOR sum of those 512 ciphertexts. As discussed in the previous section, 256 ciphertexts belonging to the first set make the balanced property after AES three rounds that is, the XOR sum for those 256 ciphertexts is 0. Similarly, the XOR sum for the 256 ciphertexts belonging to the second set is 0. Thus, the XOR sum of the 512 ciphertexts always becomes 0 even without knowing which sets each ciphertext belongs to. The analysis is shown in Figure 4.43.

#### 4.4.10.1 Higher-Order Integral Property of AES Reduced to Four Rounds

To extend the number of attacked rounds by one round,  $2^{24}$  sets of 256 plaintexts ( $\mathcal{P}_0, \mathcal{P}_1, \dots, \mathcal{P}_{2^{24}-1}$ ) are generated by changing the values of  $(c_1, c_2, c_3)$ . To make it precise, for  $\mathcal{P}_0$ , the values of  $(c_1 \| c_2 \| c_3)$  are fixed to 0 and the byte position 0 takes all the 256 possibilities. For  $\mathcal{P}_1$ , the values of  $(c_1 \| c_2 \| c_3)$  are fixed to 1 and the byte position 0 takes all the 256 possibilities. For  $\mathcal{P}_{2^{24}-1}$ , the values of  $(c_1 \| c_2 \| c_3)$  are fixed to  $2^{24} - 1$  and the byte position 0 takes all the 256 possibilities.  $2^{24} \cdot 256 = 2^{32}$  texts are required to construct  $(\mathcal{P}_0, \mathcal{P}_1, \dots, \mathcal{P}_{2^{24}-1})$ . Owing to the same discussion as the two sets case, all  $2^{32}$  ciphertexts after three rounds satisfy the balanced property, that is, the XOR sum of all  $2^{32}$  ciphertexts is always 0. Then, those  $2^{32}$  texts are computed in the backward direction by one round. The analysis is shown in Figure 4.44.



**Figure 4.44** Higher-order integral property for four-round AES  $2^{24}$  sets of 256 plaintexts are generated with  $2^{24}$  values of  $(c_1, c_2, c_3)$  at state  $S_2^I$ . This involves all the  $2^{32}$  values for the first column at state  $S_2^I$

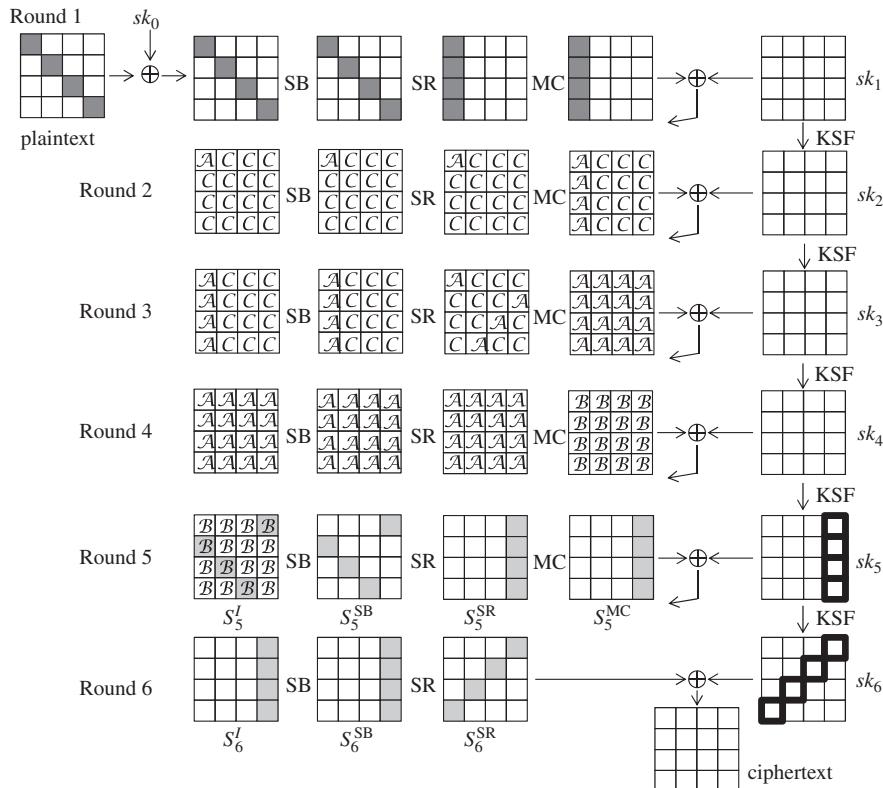
The  $2^{32}$  texts are involving all the  $2^{32}$  values for the first column at the state  $S_2^I$ . In Figure 4.44, 4 gray bytes in the state represent that all  $2^{32}$  possible values are included in the set of  $2^{32}$  plaintexts. The same text structure is preserved at the state  $S_1^{\text{MC}}$  after the inverse of the AddRoundKey operation. The MixColumns operation is a bijective mapping from 32 bits to 32 bits. Hence, the state  $S_1^{\text{SR}}$  after the inverse of the MixColumns operation has the same property, that is, all the  $2^{32}$  values appear in the first column of  $S_1^{\text{SR}}$ . The byte positions are moved at the state  $S_1^{\text{SB}}$  owing to the inverse of the ShiftRows operation. From Lemmas 4.4.1 and 4.4.2, the same property is preserved until the plaintext. In summary, the following property can be obtained for four-round AES.

**Lemma 4.4.3** Let  $\mathcal{P}$  be a set of  $2^{32}$  plaintexts in which the byte positions 0, 5, 10, and 15 take all possible values, and the other bytes take a fixed value. The corresponding  $2^{32}$  states  $C_0, C_1, \dots, C_{2^{32}-1}$  after four AES rounds have the balanced property in all the 16 bytes that is,

$$\bigoplus_{i=0}^{2^{32}-1} C_i[j] = 0, \quad 0 \leq j \leq 15. \quad (4.166)$$

#### 4.4.11 Key Recovery Attack with Integral Cryptanalysis for Six Rounds †

On the basis of the four-round higher-order integral property, a key recovery attack can be mounted against AES reduced to six rounds. The attack appends two rounds after the four-round higher-order integral property, which makes the structure shown in Figure 4.45. As explained in Figure 4.44,  $2^{32}$  plaintexts that take all  $2^{32}$  possibilities of the byte position 0, 5, 10, 15 will make all bytes at state  $S_5^I$  satisfy the balanced property after four rounds. The attacker guesses the part of the last two subkeys  $sk_6$  and  $sk_5$  and performs the partial decryption up to  $S_5^I$ . The correct guess always achieves the balanced property in all bytes of  $S_5^I$ , which enables the attacker to reduce the key space.



**Figure 4.45** Key recovery attack against six-round AES. Guessed 4 bytes of  $sk_6$  and 4 bytes of  $sk_5$  are stressed by bold lines. With those guesses, several bytes of the internal state marked by light gray color can be computed

### Collecting Plaintexts

The attacker chooses fixed values for 12 bytes of the plaintext in the byte positions 1, 2, 3, 4, 6, 7, 8, 9, 11, 12, 13, 14. The attacker then chooses  $2^{32}$  plaintexts in which the 4 bytes in 0, 5, 10, 15 take all possibilities, and the other bytes are fixed to the chosen values. Those  $2^{32}$  plaintexts are passed to the encryption oracle and the attacker obtains the corresponding  $2^{32}$  ciphertexts.

For the key recovery phase, the attacker prepares another set of  $2^{32}$  plaintexts by choosing different fixed values of the plaintext in the byte positions 1, 2, 3, 4, 6, 7, 8, 9, 11, 12, 13, 14. Those  $2^{32}$  plaintexts are also passed to the encryption oracle and the attacker obtains the corresponding  $2^{32}$  ciphertexts. In total, the attacker has two sets of  $2^{32}$  ciphertexts in the key recovery phase.

### Recovery of $sk_6$ and $sk_5$

With the obtained two sets of  $2^{32}$  ciphertexts, the attacker recovers the value of  $sk_6$  and  $sk_5$ . The 4 bytes of  $sk_6[3, 6, 9, 12]$  and 4 bytes of  $sk_5[12, 13, 14, 15]$ , in total 64-bit subkey space,

are reduced to 32 bits with the analysis of the first set of  $2^{32}$  plaintexts. Then, the subkey space is further reduced to 1 by using the second set of  $2^{32}$  plaintexts. The guessed subkey bytes are marked by bold line in Figure 4.45.

With a guess of  $sk_6[3, 6, 9, 12]$  and  $sk_5[12, 13, 14, 15]$ , the attacker can compute the corresponding 4 bytes of  $S_6^{\text{SR}}$  by taking XOR of the ciphertext and the guess of  $sk_6[3, 6, 9, 12]$  for all of  $2^{32}$  ciphertexts in the first set. The inverse of the ShiftRows operation moves the computed byte positions into 12, 13, 14, 15, and the corresponding 4 bytes of  $S_6^I$  can be computed with the inverse of the S-box. The guessed  $sk_5[12, 13, 14, 15]$  allows the attacker to compute the corresponding 4 bytes of  $S_5^{\text{MC}}$ . Because all values in a column are computed, the corresponding 4 bytes of  $S_5^{\text{SR}}$  are computed with the inverse of the MixColumns operation. The next inverse of the ShiftRows operation moves the computed byte positions into 1, 6, 11, 12, and the corresponding 4 bytes of  $S_5^I$  can be computed with the inverse of the S-box.

The value of  $S_5^I[1, 6, 11, 12]$  is computed for all the  $2^{32}$  texts in the first set. To check if the balanced property is satisfied, the XOR sum of all the  $2^{32}$  texts is computed for 4 bytes of  $S_5^I[1, 6, 11, 12]$ . If the results are 0 in all bytes, that is,

$$\bigoplus S_5^I[1] = \bigoplus S_5^I[6] = \bigoplus S_5^I[11] = \bigoplus S_5^I[12] = 0 \quad (4.167)$$

the subkey guess of  $sk_6[3, 6, 9, 12], sk_5[12, 13, 14, 15]$  can be correct. Then, the guess is stored in a list  $\kappa$ . If at least 1 byte of  $S_5^I[1, 6, 11, 12]$  does not satisfy the balanced property, the subkey guess is wrong. The subkey guess is not stored and discarded immediately.

The correct guess always satisfies Equation (4.167). Besides the correct guess, several wrong guesses happen to satisfy Equation (4.167) probabilistically. The probability that randomly chosen 4 byte values become 0 is  $(2^{-8})^4 = 2^{-32}$ . Therefore, with trying  $2^{64}$  guesses,  $2^{64} \cdot 2^{-32} = 2^{32}$  wrong guesses are stored in the correct subkey candidates list  $\kappa$ . In other words, with the analysis of the first set, the subkey space is reduced from  $2^{64}$  to  $2^{32}$ , by 32 bits.

The same analysis is iterated again by using the  $2^{32}$  ciphertexts in the second set. This time, the subkey guess of  $sk_6[3, 6, 9, 12], sk_5[12, 13, 14, 15]$  is chosen from the list  $\kappa$ , thus the number of guesses is  $2^{32}$  rather than  $2^{64}$  in the analysis for the first set. The subkey space is reduced by another 32 bits, which makes the number of subkey candidates about 1.

### ***Recovery of Original Key***

The same analysis can be performed to recover other subkey bytes of  $sk_6$  and  $sk_5$ . With iterating the analysis twice,

- 8 bytes of  $sk_6[2, 5, 8, 15], sk_5[8, 9, 10, 11]$
- 8 bytes of  $sk_6[1, 4, 11, 14], sk_5[4, 5, 6, 7]$

are recovered. As explained in the previous section, the original key  $sk_0$  can be recovered with any other subkey. The remaining unknown bytes of  $sk_5$  are the 4 bytes of  $sk_5[0, 1, 2, 3]$ . The attacker recovers those 4 bytes with the exhaustive search. Note that, if the subkey candidates for  $sk_5[12, 13, 14, 15], sk_5[8, 9, 10, 11]$ , and  $sk_5[4, 5, 6, 7]$  were not reduced to 1, but several candidates are remaining, the correct candidates can be guessed together with the exhaustive search of  $sk_5[0, 1, 2, 3]$ .

**Algorithm 4.9** Key Recovery with Integral Cryptanalysis against 6-Round AES

**Input:** A variable  $\text{tmp}$  for counting the XOR sum of returned texts from the oracle

**Output:** The original key  $K (= sk_0)$

**Construction of 2 Sets of Plaintexts**

- 1: Choose fixed value  $const_0$  for the 12 bytes in the plaintext;
- 2: Choose another fixed value  $const_1$  for the 12 bytes in the plaintext;
- 3: **for**  $i = 0, 1, \dots, 2^{32} - 1$  **do**
- 4:   Set 4 bytes of the plaintext  $P_i$  to  $i$  and combine it with  $const_0$ ;
- 5:   Query  $P_i$  to the oracle and add the corresponding ciphertext  $C_i$  to the set  $\mathcal{S}_0$ ;
- 6:   Set 4 bytes of the plaintext  $P'_i$  to  $i$  and combine it with  $const_1$ ;
- 7:   Query  $P'_i$  to the oracle and add the corresponding ciphertext  $C'_i$  to the set  $\mathcal{S}_1$ ;
- 8: **end for**

**Recovery of  $sk_6[3, 6, 9, 12], sk_5[12, 13, 14, 15]$** 

- 9: **for** Exhaustive guess of  $sk_6[3, 6, 9, 12], sk_5[12, 13, 14, 15]$  **do**
- 10:   **for** all  $2^{32}$  ciphertexts in the set  $\mathcal{S}_0$  **do**
- 11:     Compute the 4-byte value of  $S_5^I[1, 6, 11, 12]$ ;
- 12:      $\text{tmp} \leftarrow \text{tmp} \oplus S_5^I[1, 6, 11, 12]$ ;
- 13:   **end for**
- 14: **end for**
- 15: **if**  $\text{tmp}$  is 0 in 4 bytes **then**
- 16:   Add the guess to  $\kappa$ ;
- 17: **end if**
- 18: **for** Exhaustive guess of the candidates in  $\kappa$  **do**
- 19:   **for** all  $2^{32}$  ciphertexts in the set  $\mathcal{S}_1$  **do**
- 20:     Compute the 4-byte value of  $S_5^I[1, 6, 11, 12]$ ;
- 21:      $\text{tmp} \leftarrow \text{tmp} \oplus S_5^I[1, 6, 11, 12]$ ;
- 22:   **end for**
- 23: **end for**
- 24: **if**  $\text{tmp}$  is 0 in 4 bytes **then**
- 25:   Add the guess to  $\kappa'$ ;
- 26: **end if**

**Recovery of  $sk_0$** 

- 27: By applying the procedure from step 9 to step 26 for different guessed byte positions, recover the 8 bytes of  $sk_6[2, 5, 8, 15], sk_5[8, 9, 10, 11]$  and 8 bytes of  $sk_6[1, 4, 11, 14], sk_5[4, 5, 6, 7]$ ;
- 28: **for** Exhaustive guess of  $sk[0, 1, 2, 3]$  and remaining candidates of  $\kappa'$  **do**
- 29:   Compute  $sk_0$  by inverting the key schedule function;
- 30:   **if**  $C$  and  $\text{AES}_{sk_0}(P)$  match for a pair of plaintext  $P$  and ciphertext  $C$  **then**
- 31:     **return**  $sk_0$ ;
- 32:   **end if**
- 33: **end for**

### Attack Procedure

The attack procedure is described in an algorithmic form in Algorithm 4.9.

### Attack Evaluation

The attack is a chosen plaintext attack. Queries are made in steps 5 and 7, which requires  $2^{32}$  queries for each. Hence, the data complexity of this attack is  $2 \cdot 2^{32} = 2^{33}$  chosen plaintexts.

The computational cost for constructing two sets of  $2^{32}$  plaintexts is  $2^{33}$  computations, which is negligible compared to the key recovery part. For recovering  $sk_6[3, 6, 9, 12], sk_5[12, 13, 14, 15]$ , in the analysis of the first set, the two-round decryption is performed for each of the  $2^{64}$  subkey guesses and  $2^{32}$  ciphertexts in the set. Hence, the computational cost for analyzing the first set (from step 9 to step 17) is  $2^{64+32} \cdot 2 = 2^{97}$  round function computations, which is  $2^{97}/6$  six-round AES computations. The computational cost for the second set is cheaper than the one for the first set by a factor of  $2^{32}$  because only about  $2^{32}$  candidates remain in  $\kappa$ . The computational cost for the recovery of  $sk_0$  is two more iterations of the 8-byte subkey recovery in different byte positions and the exhaustive search on 32 bits and several remaining candidates in  $\kappa'$ .

The exhaustive search on 32 bits is much cheaper than the cost for the other part. As a result, the time complexity of this attack is  $3 \cdot 2^{97}/6 = 2^{96}$  six-round AES computations.

Both the sets,  $\mathcal{S}_0$  and  $\mathcal{S}_1$ , require a memory to store  $2^{32}$  AES states. The reduced subkey space  $\kappa$  also requires to store  $2^{32}$  8-byte information, which is equivalent to  $2^{31}$  AES states. The other parts are negligibly small. As a result, the memory complexity of this attack is  $2^{32} + 2^{32} + 2^{31} \approx 2^{33.3}$  AES states.

In summary, the complexity of this attack is  $(Data, Time, Memory) = (2^{33}, 2^{96}, 2^{33.3})$ .

**Exercise 4.17** In the six-round attack described in Figure 4.45, the computational cost of the key recovery attack can be improved significantly by exchanging the order of the MixColumns operation and the AddRoundKey operation with the technique introduced in Section 4.3.4.

- (a) Describe the procedure to recover the key with a smaller computational cost.
- (b) Evaluate the attack complexity and compare its cost with the one in the above-mentioned attack.

## Further Reading

- Bahrak B and Aref MR 2008 Impossible differential attack on seven-round AES-128. *IET Information Security* 2(2), 28–32.
- Biham E, Biryukov A, Dunkelman O, Richardson E and Shamir A 1999 Initial observations on Skipjack: cryptanalysis of Skipjack-3XOR In *Selected Areas in Cryptography 1998* (ed. Tavares S and Meijer H), vol. 1556 of *Lecture Notes in Computer Science*, pp. 362–375. Springer-Verlag.
- Biham E and Keller N 1999 *Cryptanalysis of Reduced Variants of Rijndael. The Second AES (Advanced Encryption Standard) Conference*.
- Biham E and Shamir A 1993 *Differential Cryptanalysis of the Data Encryption Standard*. Springer-Verlag.

- Daemen J, Knudsen LR and Rijmen V 1997 The block cipher Square In *Fast Software Encryption 1997* (ed. Biham E), vol. 1267 of *Lecture Notes in Computer Science*, pp. 149–165. Springer-Verlag.
- Knudsen LR 1998 DEAL – A 128-bit cipher. Technical Report, Department of Informatics, University of Bergen, Norway.
- Knudsen LR and Wagner D 2002 Integral cryptanalysis In *Fast Software Encryption 2002* (ed. Daemen J and Rijmen V), vol. 2365 of *Lecture Notes in Computer Science*, pp. 112–127. Springer-Verlag.
- Suzaki T, Minematsu K, Morioka S and Kobayashi E 2013 TWINE: a lightweight block cipher for multiple platforms In *Selected Areas in Cryptography, 19th International Conference, SAC 2012, Windsor, ON, Canada, August 15–16, 2012, Revised Selected Papers* (ed. Knudsen LR and Wu H), vol. 7707 of *Lecture Notes in Computer Science*, pp. 339–354. Springer-Verlag.

# 5

# Side-Channel Analysis and Fault Analysis on Block Ciphers

## 5.1 Introduction

In this chapter, **side-channel analysis** and **fault analysis** on block ciphers are introduced. They belong to the **physical attacks** or the **implementation attacks**, in which the attackers attempt to recover the secret information from a cryptographic algorithm implemented in a physical device. A straightforward implementation of the block ciphers can be easily broken by physical attacks without leaving any attack evidence.

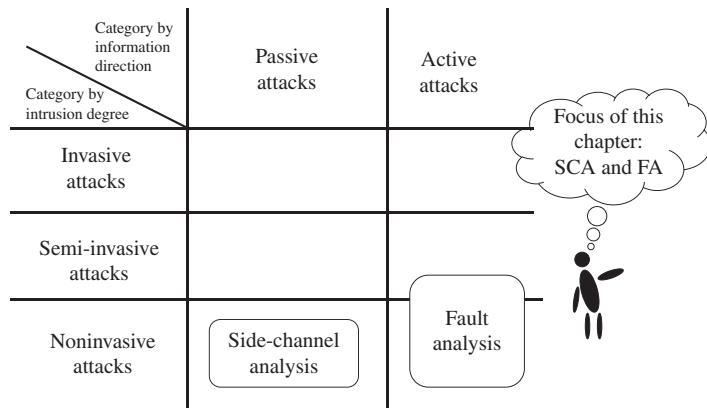
In cryptanalysis, it is assumed that the **intermediate values**, that is, the intermediate calculation result during the cryptographic operation is unavailable to the attackers. In contrast, in the physical attacks, the attackers can obtain information such as the Hamming weight (HW) of an intermediate value by physically manipulating and observing the device. Such information enables the attackers to efficiently recover the secret key of a block cipher.

### 5.1.1 Intrusion Degree of Physical Attacks

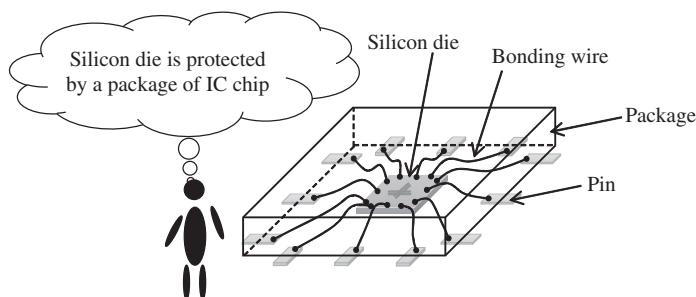
The category of the physical attacks is shown in Figure 5.1. According to the physical intrusion degree to the target device, the physical attacks are sorted into three categories; invasive, semi-invasive, and noninvasive attacks. Generally speaking, the higher the intrusion degree is, the more powerful the attack becomes. However, a deep intrusion is likely to link to a high cost in implementing the attack and may result in leaving much evidence of the performed attacks.

Figure 5.2 shows a general structure of an IC chip. Inside the package of the IC chip, the silicon die, on which the gates, wires, memory, and so on are fabricated, is in the center. The IC chip pins are connected to the silicon die via the bonding wires. Without depackaging the chip, the silicon die cannot be contacted directly. Many physical attacks with a high intrusion degree assume a direct contact with the silicon die, which implies that the package of the IC chip must be removed in the attacks.

1. **Invasive attacks** invade the silicon die in the target IC chip by removing the package material using chemicals. Well-known invasive attack is based on an observation of the signals



**Figure 5.1** Category of physical attacks



**Figure 5.2** General structure of IC chip

on wires or memory cells using a special instrument such as an electronic microscope. This type of attack is very powerful as the intermediate value in the digital circuit becomes transparent to the attackers. On the other hand, performing this type of attack requires relatively expensive equipment and special expertise. Furthermore, the risk of a permanent damage to the target device is relatively high.

2. **Semi-invasive attacks** still require removing the package to disclose the silicon die. However, different from the invasive attacks, the semi-invasive attacks require no direct electrical contact to the internal wires on the silicon die. Therefore, the possibility of the physical damage of the silicon die is largely reduced compared to the invasive attack.

In the semi-invasive attack, the attackers rather change the intermediate values than observing it directly. For example, it has been reported that a high-energy light or an optical laser beam can be used for altering the value in the memory under the semi-invasive attack scenario. The attack utilizing the alternation of the intermediate value is explained in more details later in this chapter.

3. **Noninvasive attacks** do not require the depackaging step. They only require some contact to the pins of the IC chip. Side-channel analysis and fault analysis are included in this type of attacks.

In side-channel analysis, attackers observe physical information leaked from the target device via the side channel. The attackers analyze the so-called **side-channel information** that contains sensitive information of the intermediate value of a cryptographic operation. As a result, if the attackers know the relationship between intermediate values and side-channel information in some way, the intermediate value could be disclosed, which often leads to the secret key recovery of a cipher.

On the contrary, fault analysis disturbs a cryptographic operation by intentionally injecting faults during a cryptographic computation in order to extract the secret information. It can be performed without causing any damage to the device if the fault is injected with an illegal clock signal, a voltage fluctuation, an electromagnetic interference, and so on. That is, noninvasive fault analysis differs from the semi-invasive one on how to inject the faults. The noninvasive attacks leave little evidence of the attack, so that the device user does not know the fact that the device has been attacked.

Both of side-channel analysis and fault analysis can be achieved using off-the-shelf equipments and do not require much expertise for the attackers. Therefore, noninvasive attacks are regarded as a serious security issue in practice. Thus, they are treated as the main topics of this chapter.

### 5.1.2 Passive and Active Noninvasive Physical Attacks

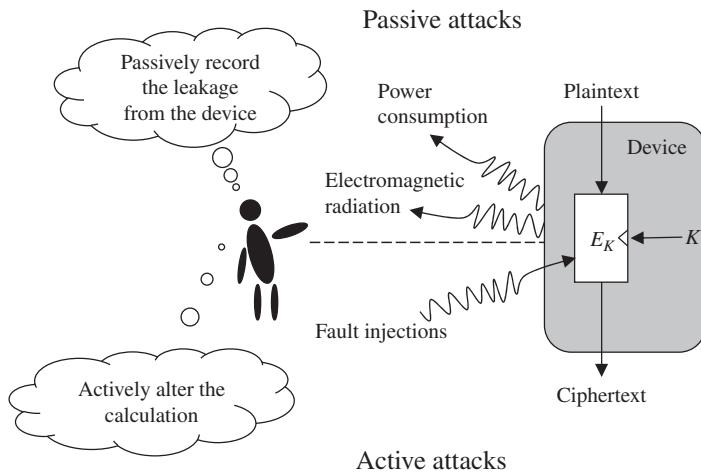
As shown in Figures 5.1 and 5.3, depending on the direction of the physical information that is used for the key recovery, the physical attacks can be divided into two types as the passive attacks and active attacks.

1. **Passive attacks** only observe the physical information leaked from the device to the attackers. The attackers do not affect the calculation of the cryptographic algorithms but only collects the leaked information during those calculations. The noninvasive passive attack is side-channel analysis attack, **side-channel attack**, or SCA for short.
2. **Active attacks** disturb the calculations of the cryptographic algorithms intentionally, which means that the attackers force some known property of the intermediate values to occur in the target device. The assumption of the active attack implies the alternation of the original calculation of the cryptographic algorithms, which is realized by performing the so-called fault injections. Therefore, these attacks are called fault injection attack, fault analysis attack, **fault attack**, or FA for short.

The passive attack and active attack are different but tightly related to each other. For instance, there is an attack in which the attackers use fault injections to obtain the information leakage from the target device and then apply the key recovery algorithms from side-channel analysis to recover the key. The rest of this chapter mainly focuses on the noninvasive attacks on block ciphers including both the passive and active ones.

### 5.1.3 Cryptanalysis Compared to Side-Channel Analysis and Fault Analysis

Cryptanalysis introduced in Chapter 4 does not use the information related to the intermediate values during the cryptographic calculations. In contrast, side-channel analysis and fault



**Figure 5.3** Passive and active attacks

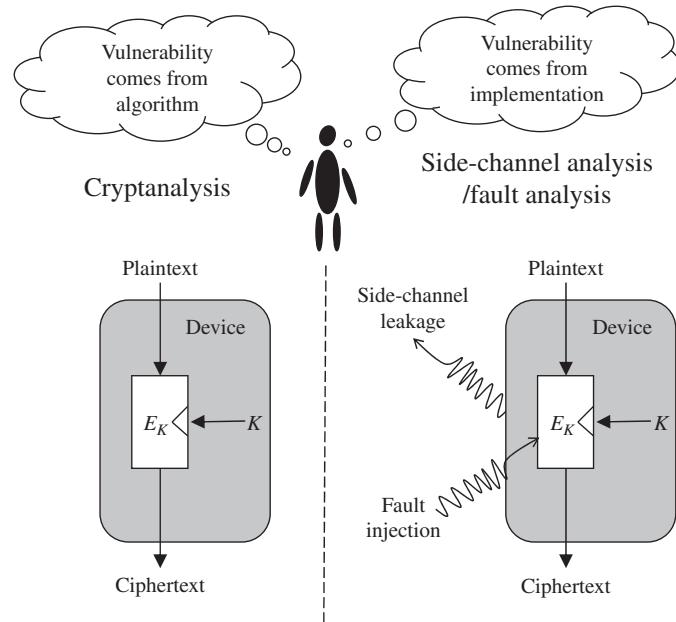
analysis take advantage of the information about intermediate values. Therefore, they become one of the most powerful and practical threats to cryptographic implementations.

Both side-channel analysis and fault analysis require the attackers to be able to contact the target device. However, cryptanalysis only requires the access to the input and output of the target algorithm. As shown in Figure 5.4, the vulnerability of theoretical cryptanalysis comes from the algorithm of the designed cipher, whereas the vulnerabilities of side-channel analysis and fault analysis largely depend on the implementation techniques of the block cipher. In other words, for different implementations of the same algorithm of a cipher, the applications of side-channel analysis and fault analysis could be different.

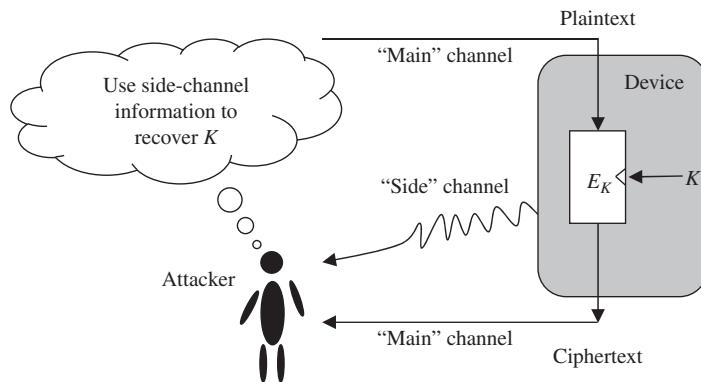
## 5.2 Basics of Side-Channel Analysis

### 5.2.1 Side Channels of Digital Circuits

The modern cryptographic modules are all based on the digital computation that is performed on physical devices. When executing the computation, the cryptographic device consumes power and causes heat, electromagnetic radiation, and so on. That is, physical manipulation and interaction of the cryptographic device open new information channels available to the attackers. As shown in Figure 5.5, the leakage channels of the physical information are considered side channels in contrast to the main channel that transmits input and output data of a block cipher. When the cryptographic devices are available to the attackers, via the side channels, the side-channel information such as power consumption can be measured during the cryptographic calculation. Because the side-channel information depends on the value of the processed data (intermediate value), the sensitive information that is related to the key could be extracted by analyzing the side-channel information. The analog physical leakage such as the power consumption or the electromagnetic radiation leaked from a device is called side-channel information. Note that the side-channel information such as the power consumption always contains noise that is not related to the cryptographic calculations.



**Figure 5.4** Cryptanalysis compared to side-channel analysis and fault analysis



**Figure 5.5** Main channel and side channel for block ciphers

The side channels exist for any physical devices. Thus, the security against side-channel analysis cannot be ignored when a cryptographic algorithm is physically implemented. Even well-designed cryptographic algorithm does not ensure its security when it is implemented in a practical device such as CPU (central processing unit) and smart card. Even for the same cryptographic algorithm, many different implementations can be achieved. Minimizing the information leakage via side channels with the least cost is set as one of the goals of the implementations.

### 5.2.2 Goal of Side-Channel Analysis

The goal of side-channel analysis is twofold. One is for device designers to verify the security of the device and the other is for the attackers to break the cryptosystem. The chip designers attempt to minimize the side-channel information leakage that can be used for the key recovery, in designing their chip. On the other hand, the attackers attempt to find useful side-channel information and the analysis method to maximize the side-channel information leakage. These two perspectives differ slightly depending on whether the secret key is known or not. As mentioned earlier, this chapter mainly stands in the attacker's perspective, so the key recovery in an unknown key setting is placed as a central discussion item.

When the secret key in a cryptographic device is exposed, the security of the system using the device would be broken. The attackers attempt to recover the secret key of a cipher with the least effort. Therefore, the most vulnerable part of the device is of interest to the attackers. Given a cryptographic device, the attackers explore what kinds of attack could be the most powerful side-channel analysis. First of all, if the attackers can read the value of the secret key directly from the device via a side channel, it will be the most powerful side-channel analysis. However, this cannot be easily done especially with a **tamper-proofed device**. The secret key is usually stored in nonvolatile memories inside the device that is protected by a package as shown in Figure 5.2. Therefore, it is almost impossible to read out the secret key physically without breaking the package of the chip. This kind of attack cannot be realized with the noninvasive side-channel analysis.

Although the secret key cannot be accessed outside the device, it is used for performing the cryptographic computation when encrypting the plaintext. Accordingly, the intermediate values during the computation are related to the secret key, and the side-channel information contains information correlated to the value of the secret key. As long as the secret key information can be extracted from the side-channel information using side-channel analysis, the key recovery can be achieved.

In order to achieve a successful key recovery, a requirement for side-channel analysis is the knowledge of the target device for the attackers. On the one hand, the knowledge of the target device helps the attackers to understand the relations between the intermediate values and the side-channel information. On the other hand, the attackers attempt to achieve the side-channel analysis that requires the least knowledge of the target device to recover the key. The knowledge of the device includes the information about the architecture of the target cryptographic device and the characteristics of the power consumption of the device.

In addition to the knowledge of the device, the attackers' computational capability is tightly related to the success of the key recovery. When analyzing the side-channel information, the attackers need to reduce the computational cost for the key recovery by dividing it into multiple small computations until the attackers can compute them within a reasonable analysis time. This is called **divide-and-conquer** technique, which is detailed later in this chapter. The key recovery computation that can be done by personal computers is considered a real threat because every attacker can use a personal computer with a decent computational ability in the current age.

The key recovery algorithm extracts useful information from a set of noisy data. Regarding the required number of traces to successfully recover a key, the smaller the number of traces is, the more advanced the key recovery algorithm is. That is, the number of traces should be minimized for the key recovery of the side-channel analysis, and it is commonly used as an efficiency metric of the key recovery algorithms.

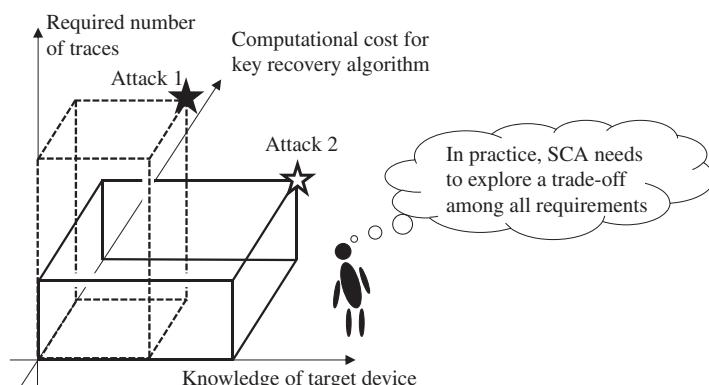
### 5.2.2.1 Trade-Off in Side-Channel Analysis

In practice, the parameters, that is, the knowledge of the target device, the computational cost, and the required number of traces, to make an efficient side-channel analysis are conflicting with each other. For example, less knowledge of the target device implies that more traces are required for the key recovery. In addition, when the number of traces for the key recovery is limited, a more complicated key recovery algorithm is required with more computational cost. Therefore, the attackers always attempt to find a reasonable trade-off among the required number of traces, the computational cost, and the required knowledge of the target device as shown in Figure 5.6. In the figure, Attack 2 requires fewer traces than Attack 1 but requires more computational cost and more knowledge of the target device. In this way, the attackers explore the trade-offs suitable for the corresponding attack scenarios. Note that a similar trade-off also exists for fault analysis.

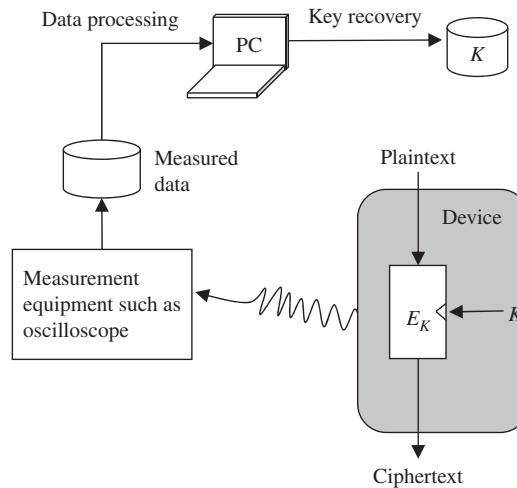
### 5.2.3 General Procedures of Side-Channel Analysis

The general procedures of side-channel analysis can be divided mainly into two steps as shown in Figure 5.7.

- **Manipulation of the device for measuring physical information:** the attackers should have the access to the device. For example, control the plaintext and/or ciphertext of a block cipher. Then, the attackers need to use side channels to obtain the side-channel information when the device is performing the cryptographic computation. For the passive attack, the attackers only observe or measure the physical information leaked from the device.
- **Data analysis of the side-channel information:** the side-channel measurement data goes through the data processing such as data alignment and noise reduction. If the side-channel information includes enough information regarding the secret key, and the attackers have an ability to extract key-related information from the data, the key recovery attack will be successful. In this regard, it is natural that the attackers attempt to obtain more traces and find more powerful analysis tools in order to achieve an efficient key recovery attack.



**Figure 5.6** Trade-offs in side-channel analysis



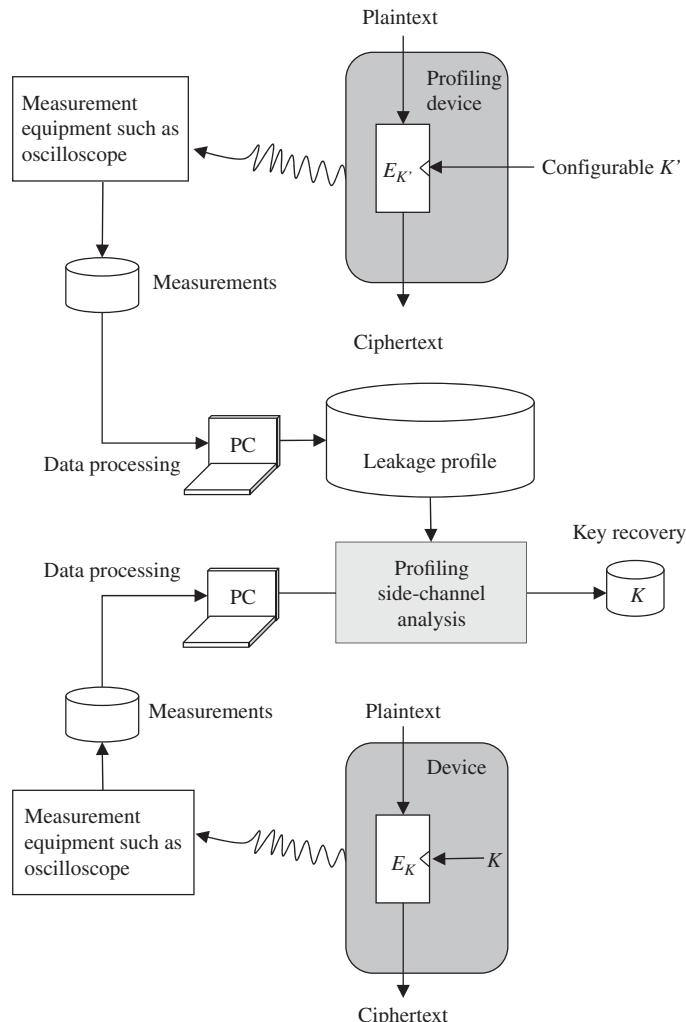
**Figure 5.7** General procedures of side-channel analysis

#### 5.2.4 Profiling versus Non-profiling Side-Channel Analysis

According to the attackers' ability, side-channel analysis can be divided into **profiling analysis** and **non-profiling analysis**. The challenges of side-channel analysis are based on how to link the side-channel leakage to the intermediate values and how to reduce the noise of the side-channel measurement data to identify the correct key. Profiling analysis and nonprofiling analysis differ in the methods for linking the side-channel leakage to intermediate value.

For profiling analysis, the attackers can use a **profiling device**, which has similar characteristics of the side-channel leakage of the target device with a configurable key, to learn the details of the leakage characteristics of the target device. The profiling device could be the same manufacture as the target device, so that the similar leakage characteristics can be expected. Figure 5.8 shows the basic idea of a profiling side-channel analysis. Profiling analysis has two phases; the **profiling phase** and the **attack phase**. In the profiling phase, the attackers study the unique leakage characteristics of the target device using the profiling device with a configured key. Note that as the attackers do not know the key,  $K$ , in the target device, the configured key,  $K'$ , is different from  $K$  as shown in Figure 5.8. Thus, the attackers attempt to consider how to make the leakage profile that does not largely depend on the value of key. The leakage characteristics learned from the profiling device are called a **leakage profile**. In the attack phase, the attackers attempt to recover the unknown key in the target device with the help of the leakage profile. Profiling analysis requires fewer traces in the attack phase compared to the nonprofiling analysis.

For nonprofiling analysis, the attackers attempt to use general knowledge of the target device to construct a **leakage model (LM)** that describes the relations between processed data and the side-channel information in the key recovery based on the knowledge of the target device. With the LM and the side-channel measurement data obtained from the target device, nonprofiling analysis conducts the key recovery in the attack phase. Figure 5.9 shows the basic idea of nonprofiling side-channel analysis. More details of nonprofiling side-channel analysis are explained later with the attack examples.

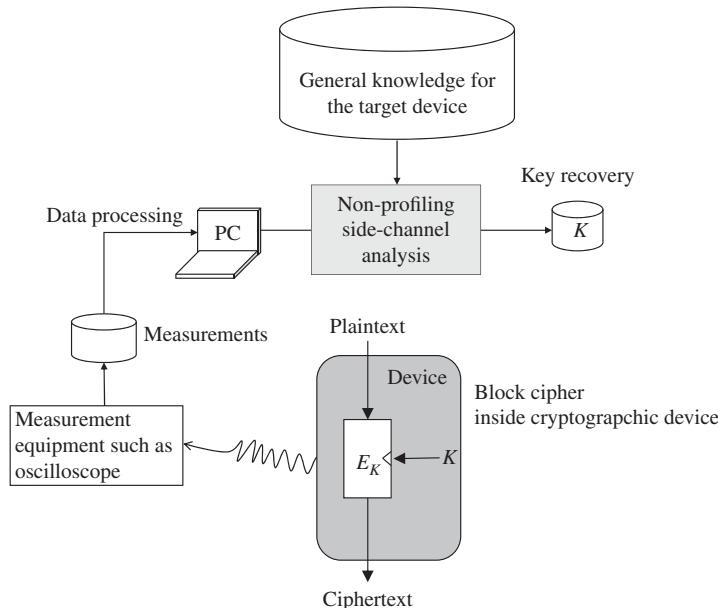


**Figure 5.8** Profiling side-channel analysis

### 5.2.5 Divide-and-Conquer Algorithm

Divide-and-conquer algorithm is the essence to explain the effectiveness of side-channel analysis. The information leakage via the side channels is accompanied by noise in the traces. Therefore, in the sense of the signal quality, the side-channel information is significantly worse than the digital information such as plaintexts and ciphertexts. However, the fact that the side-channel leakage can be linked to the intermediate value makes side-channel analysis very powerful.

Side-channel analysis only focuses on a part of the cryptographic algorithm that is close to the **public data**, that is, plaintexts or ciphertexts. For a part of the cryptographic algorithm, in most cases, it is easy and obvious to divide the computation into small ones. For example, the



**Figure 5.9** Nonprofiling side-channel analysis

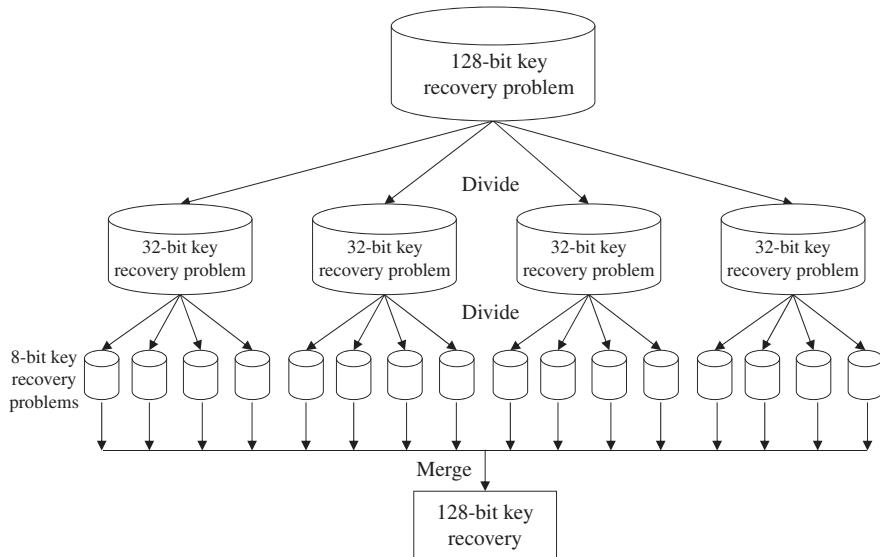
related key bits for a small part of the attacked algorithm can be 32 bits, 8 bits, or even 1 bit. To achieve both the security and the efficiency, modern block ciphers are tend to be composed of many small components. For a part of the algorithm that is focused by side-channel analysis, usually it can be broken down into small computational parts. Then, the key piece used for each part can be exhaustively tested with a reasonable computational cost. The side-channel information can be used in the exhaustively test to recover each small piece of the key.

Figure 5.10 shows the concept of a divide-and-conquer algorithm. The recovery of a 128-bit key is divided into the recovery of sixteen 8-bit key pieces. After recovering and combining all the 8-bit key pieces, the original 128-bit key can be recovered.

In side-channel analysis, the key recovery attack for each key piece can be regarded as a brute-force attack with the help of side-channel information. The basic concept of the brute-force attack has been introduced in Chapter 4. For a block cipher such as AES-128, a pair of plaintext and ciphertext can be used to verify the correctness of a guess of the 128-bit key. The guess of the key is used to encrypt the plaintext to get a corresponding ciphertext. Then, the value of the calculated ciphertext is compared with the real one. As long as the calculated ciphertext is different from the real one, it is sure that the current guess of the key is different from the real key.

When the attackers discard the wrong key candidates, the key space can be reduced. If the key space can be continuously reduced, eventually the key recovery is possible for the attackers. However, the brute-force attack for a modern cryptographic algorithm, whose key size is at least 128 bits, cannot be finished practically because there are too many key candidates to be tested even with the fastest super computer in the world.

Instead of using pairs of plaintext and ciphertext in the original brute-force attack, pairs of side-channel leakage and public data, that is, plaintext or ciphertext, are used in the key



**Figure 5.10** Concept of divide-and-conquer algorithm

recovery of side-channel analysis. With the divide-and-conquer algorithm, the key recovery of the full key becomes the recovery of several small key pieces. For each key piece, the number of all possible key candidates is small enough to be exhaustively verified.

Assume that the side-channel leakage can be converted into the exact intermediate values with probability 1. Then, the brute-force attack in side-channel analysis becomes exactly the same with the one introduced in Chapter 4. In practice, the side-channel key recovery algorithms are developed to convert the side-channel leakage to the information of intermediate values.

**Exercise 5.1** Assume that there are two block ciphers A and B. Both of them have a 128-bit secret key and a 128-bit datapath. Block cipher A uses 8-bit S-boxes, so its secret key can be recovered 8-bit by 8-bit with the divide-and-conquer algorithm. Block cipher B uses 16-bit S-boxes, so its secret key can be recovered 16-bit by 16-bit with the divide-and-conquer algorithm. Discuss the difference of the computational cost between the key recovery for block ciphers A and B.

### 5.3 Side-Channel Analysis on Block Ciphers

This section introduces the details for side-channel analysis on block ciphers. As the most promising method for side-channel analysis, power analysis is explained. Power analysis

receives the most attention in side-channel analysis research area as the power consumption is inevitable for the cryptographic computations and it is easy to be measured.

In the introduction of power analysis, the measurement of the power traces and the general key recovery algorithm will be explained. Several detailed key recovery algorithms for power analysis and their attack results on two hardware AES implantations will be explained.

### 5.3.1 Power Consumption Measurement in Power Analysis

In this chapter, **side-channel measurement data** denotes the sets of digitalized information converted from analog physical leakage at a certain sampling frequency. They contain the digitalized physical leakage against the discrete time determined by the **sample points**. The side-channel measurement data is also called **traces**, for example, power traces in the case for measuring the power consumption.

The side-channel measurement data is digitalized information converted from analog physical leakage that is significantly influenced by a measurement setting, environmental parameters, and so on. Thus, in the side-channel measurement data, **measurement noise** inevitably exists. The noise in the power traces is relatively small compared to other side-channel information. The attackers attempt to improve the measurement setup to reduce the measurement noise in their measurement data.

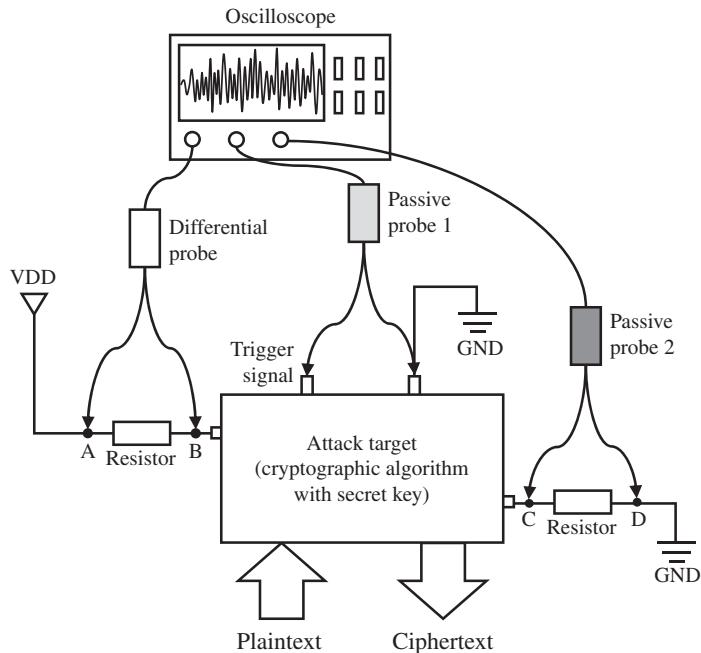
As shown in Figure 5.11, general-purpose oscilloscopes, probes, and resistors can be used in measuring the power consumption. The oscilloscope samples the voltage fluctuation along the time and records them as a power trace. The target device is usually an IC chip, in which a cryptographic algorithm is implemented.

The power supply and the ground are denoted by VDD and GND in Figure 5.11. For the power consumption measurement, the resistor is inserted between the VDD of the device and the VDD pin of the IC chip or between the GND of the device and the GND pin of the IC chip. The voltage drop by the resistor is related to the power consumption of the cryptographic algorithm, so the attackers use it for the SCAs.

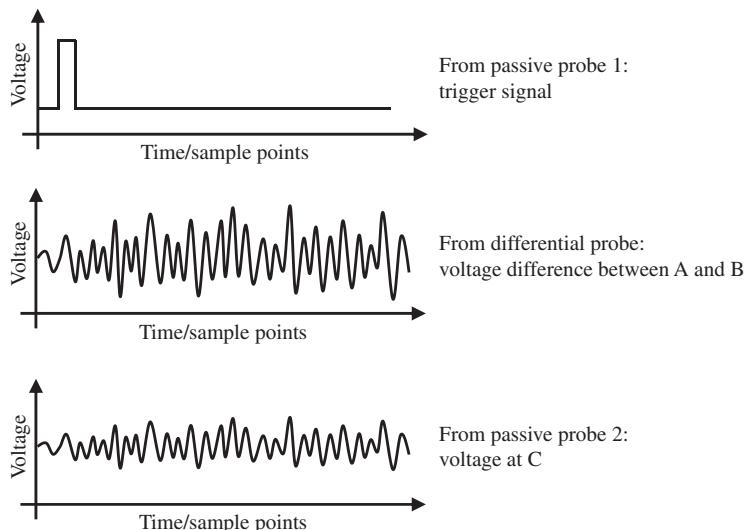
In the measurement of power consumption, a probe is used for transmitting the voltage fluctuation to the oscilloscope. There are mainly two types of probes used for the power consumption measurement; differential probe and passive probe. The differential probe is able to measure the voltage difference between two measurement points. The passive probe can measure the voltage of a measurement point against the ground of the device.

In order to measure the power consumption that is useful for the key recovery, the measurement timing is very important. The power consumption measurement should be performed when the target device is performing a cryptographic algorithm. The appropriate timing for the oscilloscope to sample and record the power consumption can be achieved using a trigger signal. The trigger signal is usually obtained from the target device itself and has a fixed timing relationship with the cryptographic algorithm. Thus, the trigger signal can be used to achieve the alignment of power traces for multiple power measurements. The accuracy of the trigger signal directly relates to the accuracy of the timing alignment for multiple power traces, which could affect the efficiency of power analysis eventually.

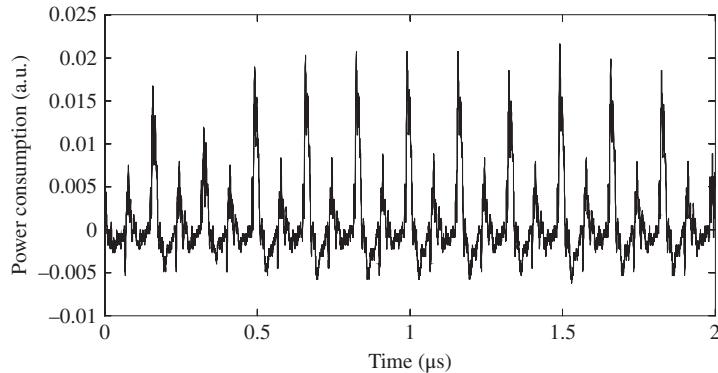
The side-channel measurement data or the traces are normally plotted as a continuous line along the time axis or the sample point axis. In accordance with the measurement setup shown in Figure 5.11, an illustration of the measurement data is shown in Figure 5.12. The trigger



**Figure 5.11** A typical power measurement setup



**Figure 5.12** Illustration of observed data for power measurement setup shown in Figure 5.11



**Figure 5.13** Power consumption trace example for hardware implementation of AES

signal from passive probe 1 detects a start signal of the cryptographic algorithm. The measurement data from differential probe is the sampled voltage difference between point A and point B, which is related to the current of the resistor connected to the VDD pin. The measurement data from passive probe 2 is the sampled voltage at point C, which is related to the current flowed via the GND pin. In Figure 5.11, the traces from differential probe and from passive probe 2 are synchronized by the same trigger signal. Thus, these two traces should contain the almost same information about the power consumption of the target IC chip. In practice, both the VDD part and the GND part can be chosen to perform the power measurement.

An example of the power trace from a real hardware implementation of AES is shown in Figure 5.13. The power trace is a line that connects all the neighboring sampled voltage points. The horizontal axis stands for the time or the sample points. The vertical axis is the voltage of measurement result in an arbitrary unit that is denoted by “a.u.” There are 11 power consumption peaks in Figure 5.13. They correspond to one clock cycle for each of the 10 rounds of AES-128 and one more clock cycle for the data transfer. Note that the public data such as plaintexts and ciphertexts could be recorded by the attackers depending on the attack scenario as well.

Algorithm 5.1 illustrates the procedures of the collection of power traces. For a set of plaintexts  $\{P_1, P_2, \dots, P_N\}$ , the corresponding ciphertexts  $\{C_1, C_2, \dots, C_N\}$  and the power traces  $\{W_1, W_2, \dots, W_N\}$  are collected. The  $j$ th sample point of the  $i$ th power trace is denoted by  $W_{i,j}$ , where  $i = 1, 2, \dots, N$  and  $j = 1, 2, \dots, M$ . Thus, the  $i$ th power trace  $W_i$  is represented as a sequence set of  $M$  sampling points, that is,

$$W_i = \{W_{i,1}, W_{i,2}, \dots, W_{i,M}\}.$$

Figure 5.14 shows the illustration of the data measurement of the plaintexts/ciphertexts and the power traces.

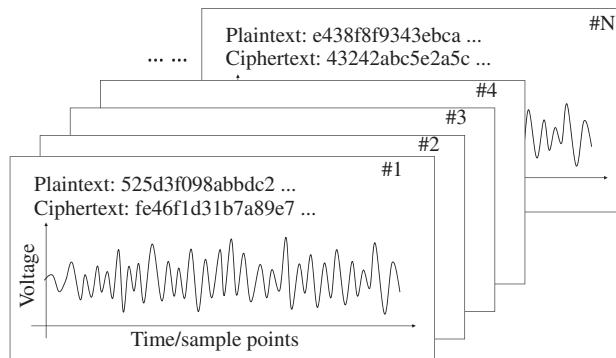
In the measurement data of power traces, the attackers expect that they contain the **side-channel information leakage** or **side-channel leakage** that is the useful information for the key recovery. The power consumption of a cryptographic device could contain the information about the intermediate value. As the intermediate value could be related to the secret key, the attackers may be able to use some key recovery algorithms to extract the side-channel information leakage inside the side-channel measurement data.

**Algorithm 5.1** Collection of Power Consumption Traces

**Input:** A set of plaintexts,  $\{P_1, P_2, \dots, P_N\}$

**Output:** Paris of ciphertexts and power traces  $\{(C_1, W_1), (C_2, W_2), \dots, (C_N, W_N)\}$

- 1: **for**  $i = 1$  to  $N$  **then**
- 2:   Set the plaintext as  $P_i$ ;
- 3:   Record ciphertext as  $C_i$  and power traces for  $E_K(P_i)$  as  $W_i$ ;
- 4: **end for**
- 5: **return**  $C_i$  and  $W_i$  for  $i = 1, 2, \dots, N$ ;



**Figure 5.14** Data measurement of power analysis

### 5.3.2 Simple Power Analysis and Differential Power Analysis

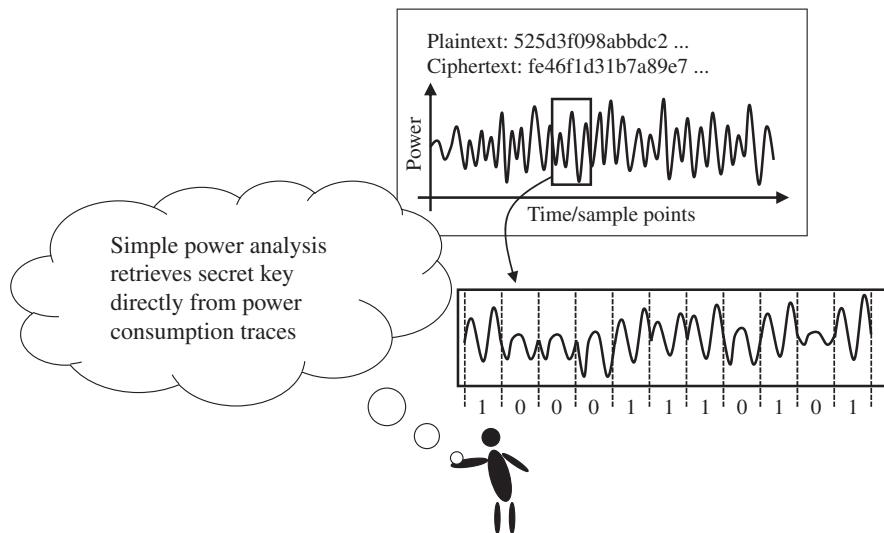
The key recovery algorithm of power analysis can be mainly divided into two types as **simple power analysis (SPA)** and **differential power analysis (DPA)**, which were proposed in Kocher *et al.* (1999). SPA uses only one single power trace or the mean of a few power traces to perform the key recovery. Assume that there are  $N$  power traces, each has  $M$  sample points. The mean of these power traces can be represented by

$$\left\{ \frac{1}{N} \sum_{i=1}^N W_{i,1}, \frac{1}{N} \sum_{i=1}^N W_{i,2}, \dots, \frac{1}{N} \sum_{i=1}^N W_{i,M} \right\}. \quad (5.1)$$

In SPA, the value of the secret key can be understood only by reading the shape of the power trace.

The scenarios of successful applications of SPA are limited. One of successful applications of SPA is the modular exponentiation of the RSA (Rivest–Shamir–Adleman) algorithm that is a widely used public key cryptographic algorithm. Comparing the power trace with the power consumption patterns, the key recovery of SPA can be performed. Figure 5.15 shows an attack illustration of SPA where the power consumption patterns for value 1 or 0 of each bit of the secret key can be easily identified. As SPA on AES implementation is hardly possible, this chapter skips more details of SPA.

In contrast to SPA, DPA refers to power analysis that performs statistical analysis on multiple traces in the key recovery. For DPA, the secret information cannot be directly retrieved with



**Figure 5.15** Attack illustration of simple power analysis

one or a few power traces. A large amount of power traces are normally required to perform statistical analysis that can extract a small difference among the power traces. In practice, the number of the power traces used for DPA ranges from hundreds to a few millions. Hereafter, this chapter mainly focuses on the key recovery algorithm of the DPA attacks.

### 5.3.3 General Key Recovery Algorithm for DPA

This section explains the general key recovery algorithm for the DPA attacks. The input for the key recovery algorithm includes the power traces and the public data such as ciphertexts. The goal of the key recovery algorithm is to obtain the secret key, which is achieved by making sure that the correct secret key value can be distinguished from others. The attackers construct links among the secret key, the public data, and the side-channel measurement data to achieve the successful key recovery.

As the attackers do not know the value of the secret key, DPA starts from guessing the value of the key as a key guess. Then, the attackers use the links among the key, the public data, and the power traces to perform some calculations using the current key guess. For every possible key guess, the above-mentioned calculation is performed.

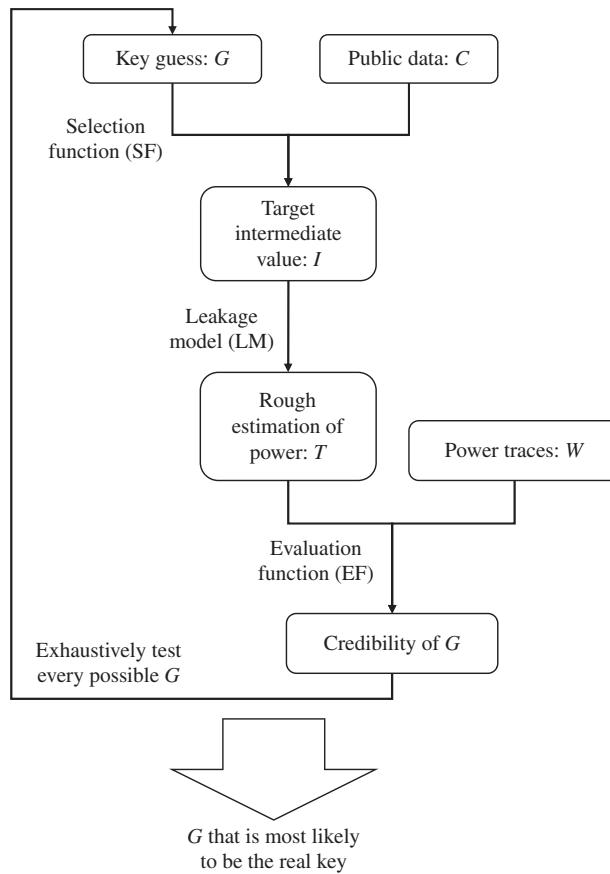
The attackers know that only if the key guess is correct, the links between the key guess and the measurement data can be established and a meaningful result can be obtained. For the wrong key guesses, the above-mentioned calculation behaves with no reasons; some meaningless results will be obtained. Therefore, it is expected that the calculation result for the correct key guess can be distinguished from those for the wrong key guesses. In other words, from the calculation results corresponding to all key candidates, the attackers can identify the correct key.

In a general key recovery algorithm with DPA, the links among the key, the public data, and the power traces can be broken down into **selection function (SF)**, **LM**, and **evaluation function (EF)**. A brief explanation is given as follows.

- **Selection function** is a link from the key guess and the public data to an intermediate value called **target intermediate value**,  $I$ . The SF is normally a part of the cryptographic algorithm that calculates the target intermediate value using the public data and a key guess,  $G$ . For example, when attacking AES using the ciphertexts as the public data, the SF could be a partial decryption using the last round of AES.
- **Leakage model** is a link from the target intermediate value to the rough estimation of the power consumption,  $T$ . The **LM** roughly describes the relations between intermediate values and the side-channel information. In the key recovery algorithms, an LM is a function that maps the calculated target intermediate value to some values that represent rough characteristics of the power consumption. For example, the attackers use the results of the LM with a certain key guess to classify the power traces into different groups and expect that the power traces in different groups have different power consumptions. Note that the LM does not have to describe real power consumption very accurately. As far as the rough leakage characteristics described by the LM match the real power consumption, the key recovery of DPA is possible with enough power traces.
- **Evaluation function** is a link from the rough estimation of power consumption and the real power traces to the credibility of the key guess. Evaluation function uses a statistical tool to extract the relations between the rough estimation of power consumption and the real power traces. The extracted relations are quantitatively expressed as a value that can represent the credibility of the current key guess. In the key recovery algorithm, the EF maps the rough estimation of power consumption with a key guess and the real power traces to the credibility of the current key guess. Finally, the credibilities of all key candidates are compared with each other, and the one that is most likely to be the correct key is picked up as the attack result.

Figure 5.16 shows an illustration of the general key recovery algorithms for DPA. The key guess is linked to the target intermediate value using public data and an SF. The target intermediate value is linked to the rough estimation of power consumption using an LM. The rough estimation of power consumption is linked to the credibility of the current key guess using the real power traces and an EF. Finally, the DPA attack returns the key guess that is most likely to be the correct key.

The key recovery algorithm of DPA is shown in Algorithm 5.2. In Algorithm 5.2, the ciphertexts are used as the public data and only a specific single sample point,  $j$ , of the power traces is used. Step 1 in Algorithm 5.2 is the exhaustive search for all the possible key candidates. Step 3 is the SF and the LM calculation using the ciphertexts and key guess as the inputs. The calculation result of step 3 denoted by  $T$  is expected to have some relations with the real power consumption when the key guess is correct. Step 5 uses the EF to extract the relations between  $T$  and the power traces, and its result denoted by  $E_G$  is obtained for every key candidates as their credibilities as a real key. Finally, step 7 picks up the  $G$  that is most likely to be the expected result for the correct key as the attack result.



**Figure 5.16** General key recovery algorithms for differential power analysis

---

**Algorithm 5.2** General DPA Key Recovery Algorithm with Single Sample Point

**Input:**  $N$  pairs of ciphertext and power trace:  $(C_1, W_1), (C_2, W_2), \dots, (C_N, W_N)$ , sample point index  $j$

**Output:** Recovered key piece:  $K$

- 1: **for** exhaustive guess of  $K$  denoted by  $G$  **do**
  - 2:   **for**  $i = 1$  to  $N$  **do**
  - 3:      $T_{i,G} \leftarrow \text{LM}(\text{SF}(C_i, G))$ ;
  - 4:   **end for**
  - 5:      $E_G \leftarrow \text{EF}(T_{i,G}, W_{i,j})$  for  $i = 1, 2, \dots, N$ ;
  - 6: **end for**
  - 7: **return**  $G$  where  $E_G$  is most likely to be the expected result for the correct key;
-

**Algorithm 5.3** General DPA Key Recovery Algorithm with All Sample Points

**Input:**  $N$  pairs of ciphertext and power trace:  $(C_1, W_1), (C_2, W_2), \dots, (C_N, W_N)$

**Output:** Recovered key piece:  $K$

```

1: for exhaustive guess of  $K$  denoted by  $G$  do
2:   for  $i = 1$  to  $N$  do
3:      $T_{i,G} \leftarrow \text{LM}(\text{SF}(C_i, G));$ 
4:   end for
5:   for each sample point  $j = 1, 2, \dots, M$  of power trace do
6:      $E_{j,G} \leftarrow \text{EF}(T_{i,G}, W_i)$  for  $i = 1, 2, \dots, N;$ 
7:   end for
8:    $\text{ETrace}_G \leftarrow \{E_{1,G}, E_{2,G}, \dots, E_{M,G}\};$ 
9: end for
10: return  $G$  where  $\text{ETrace}_G$  is most likely to be the expected result for the correct key;

```

---

In practice, the attackers are not sure about which of the sample points is the most useful one in the key recovery. The natural solution is to repeat Algorithm 5.2 for each sample point of the power traces. Then based on the results of all the sample points, the key recovery is performed. Note that step 3 in Algorithm 5.2 does not need to be repeated when attacking different sample points. Thus, the key recovery algorithm with all the sample points in the traces examined can be performed as shown in Algorithm 5.3.

Step 5 in Algorithm 5.2 is extended to steps 5 to 7 in Algorithm 5.3. The evaluation result in Algorithm 5.2 for each key guess is in the form of 1 value. The evaluation result in Algorithm 5.3 for each key guess is in the form of a trace that has  $M$  values as shown in step 8 in Algorithm 5.3. The other parts of Algorithms 5.2 and 5.3 are exactly the same.

Hereafter, for the key recovery algorithm of each specific attack method, all the sample points will be used to demonstrate the attack result for the entire AES calculation. In practice, the attackers can pick a few sample points to perform the DPA attacks such that the computational cost can be reduced.

**Exercise 5.2** Compare the computational costs of Algorithms 5.2 and 5.3.  
Explain what kind of trade-off exists between Algorithms 5.2 and 5.3.

### 5.3.3.1 How to Construct SF

The construction of an SF is essentially the same as the selection of the target intermediate value. When the target intermediate value is fixed, the SF is the calculation between public data to the intermediate value following the specification of the block cipher.

To make the key recovery in DPA possible, the target intermediate value must be related to both the power measurement data and the key guesses. If the target intermediate value is not related to the measurement data, the link to the real power traces cannot be established.

Thus, the key recovery will fail. If the target intermediate value is not related to the secret key, for example, using the ciphertext as the target intermediate value, the calculated credibilities will become the same for all the key candidates. Thus, the key recovery will fail as well.

To make the DPA attack efficient, the selection of the target intermediate value should have the following two features.

- A small key piece, for example, a key byte, is involved in the SF, so that the divide-and-conquer algorithm is applied with a reasonable computational cost. To achieve this, the target intermediate value should be close to the public data in the cryptographic algorithm, for example, one round or two rounds before the ciphertext.
- The target intermediate values calculated with a wrong key guess can be treated to be independent of the real ones. When the key guess is correct, all the calculated target intermediate values are the same with the real ones. As long as the leakage function and the EF are reasonable, the link between the rough estimation of power consumption and the real traces can lead to a high credibility for the correct key. For the wrong key guess, if there is no dependency from the wrong target interminable values to the real ones, the calculated credibility for the wrong key guesses can be treated as meaningless values and the key recovery becomes possible. In order to achieve the independence between wrong target intermediate values and the real ones, the SF should involve some nonlinear calculations, such as the SubBytes operation of AES.

When attacking AES, the mostly used target intermediate value is 1 byte of the last round input, that is,  $S_{10}^I[B]$ , where  $B$  denotes the byte position as  $B \in \{0, 1, \dots, 15\}$ .  $S_{10}^I[B]$  is the input to the SubBytes operation that is an important information source of physical leakage. In addition,  $S_{10}^I[B]$  is usually stored and updated in DFFs (delay flip flops), which is another important information source of physical leakage. The public data,  $C$  and  $S_{10}^I$ , has the following relation:  $\text{AK} \circ \text{SR} \circ \text{SB}(S_{10}^I) = C$ . When  $B = 0$ , the first byte of  $S_{10}^I[0]$  can be calculated as  $S^{-1}(C[0] \oplus sk_{10}[0])$  involving 1 byte of the secret key and the nonlinear S-box calculation. Hereafter, for the DPA attacks on AES,  $S_{10}^I[0]$  is used as the target intermediate value and  $sk_{10}[0]$  is the key byte to recover.

### 5.3.3.2 How to Construct LM

The LM roughly describes the relations between intermediate values and the side-channel information. This section explains three LMs that are based on simple assumptions about the power consumption. These three LMs will be applied to the attacks examples as shown later in this chapter.

- **Single-bit model**

The most straightforward assumption to construct an LM is that processing different intermediate values leads to different power consumptions. If the attackers only focus on 1-bit value of the intermediate value, the power consumptions are different when this 1-bit value is 1 or 0. Hereafter, this LM is called the single-bit model. For a CMOS (complementary metal oxide semiconductor) circuit, the motivation of considering the single-bit model is due to the fact that value 1 requires the charge of electrons to a capacitor, whereas the value 0 leads to discharge of electrons.

- **Hamming weight (HW) model**

The single-bit model can be extended to the HW model, where the values of multiple bits are considered. In the single-bit model, it is assumed that processing value 1 and processing value 0 have different power consumptions. In the HW model, it is assumed that all the bits of a multiple-bit intermediate value follow the single-bit model. That is, the power consumption of a multiple-bit intermediate value is proportional to the number of 1s in it. The number of 1s in a value is called **HW**. Therefore, this LM is called HW model. Following the HW model, for an intermediate value 31, one can use  $\text{HW}(31) = 3$  to roughly estimate the power consumption.

- **Hamming distance (HD) model**

The HW model focuses on the intermediate value at a specific timing. For the HD model, the update of intermediate values from one state to another state is the focus. For each bit of an intermediate value, switching the value of this 1-bit intermediate value and keeping the value of this 1-bit intermediate value could have different power consumptions. In a CMOS circuit, the value switch requires either charge or discharge of a capacitor, while keeping the same value does not. On the basis of this assumption, as a very rough LM, the power consumption is expected to be proportional to the number of bit-flips of two states of an intermediate value. The number of bit-flips between two values is called **HD**. Therefore, this LM is called HD model. Following the HD model, for an intermediate value changing from 31 to 83, one can use  $\text{HD}(31, 83) = \text{HW}(31 \oplus 83) = 4$  to roughly estimate the power consumption.

### 5.3.3.3 How to Construct EF

The EF is some statistical tool to extract the relationship between the rough estimation of the power consumption and the real ones. Thus, the selection of the EF is largely related to the description of the LM. For example, when a high correlation is expected between the rough estimation of the power consumption and the real power traces, the correlation coefficient calculation can be used as an EF. The construction of the EF is explained in details for each attack algorithm.

### 5.3.4 Overview of Attack Targets

This section briefly explains the implementations of two hardware AES implementations named **AES-pprm1** and **AES-comp**, which were proposed in Morioka and Satoh (2002) and Satoh *et al.* (2001). Both AES-pprm1 and AES-comp are without any countermeasures against side-channel analysis. Later, the power traces corresponding to these two hardware implementations of AES are used to demonstrate the detailed power analysis algorithms.

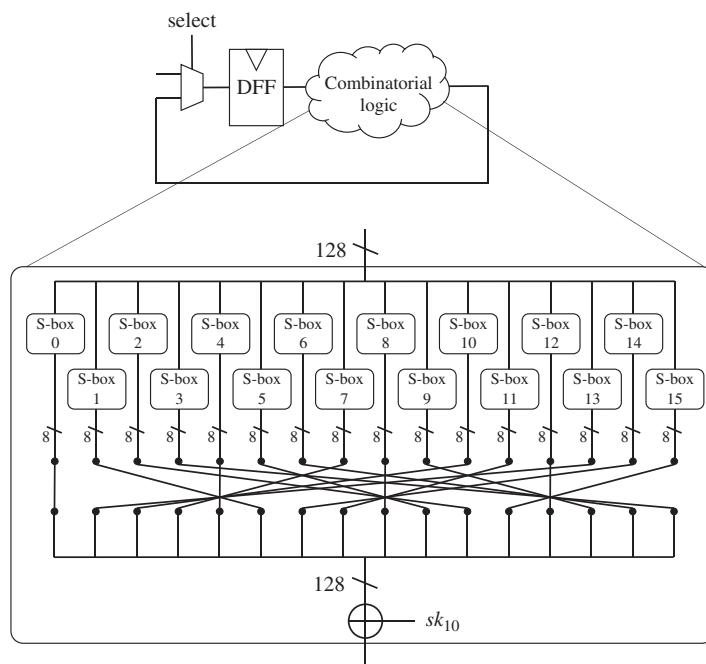
#### 5.3.4.1 Hardware Implementation of AES-pprm1 and AES-comp

The implementations of AES-pprm1 and AES-comp share the same basic architecture, that is, a parallel implementation using the loop architecture. Each AES round is calculated inside 1 clock cycle. In the performed key recovery algorithms later in this chapter, only the last round of AES is analyzed, which has the SubBytes, ShiftRows, and AddRoundKey operations. For the SubBytes operation, there are 16 S-box circuits in the combinatorial logic and they

are calculated in parallel. The ShiftRows operation is implemented by arranging the wires in the circuit. The AddRoundKey operation is the XORs with the subkey  $sk_{10}$ . The illustration of the hardware architecture of the last round for AES-pprm1 and AES-comp is shown in Figure 5.17.

Owing to the fact that 16 S-boxes are calculated in parallel, the measured power consumption can be considered the summation of the power consumption of all the 16 S-boxes. As mentioned earlier, in the key recovery of AES, the key bytes are recovered byte by byte. The target intermediate value is set to 1 byte of value that is related to 1 byte of the key. Thus, it is favorable for the attackers to measure the power consumption for each S-box independently. However, for AES-pprm1 and AES-comp the power consumption is measured as the summation of that for each S-box. When targeting one S-box, the power consumption of the other 15 S-boxes can be treated as noise in the key recovery. To understand this, recall that the rough estimation of power consumption is related to only 1 byte of the last AES round input. However, the power traces correspond to the summation of power consumption for 16 bytes of the last AES round input. The power consumption of the other 15 S-boxes makes the relations between the rough estimation and the power consumed by the target byte more difficult to extract.

Here, different from the measurement noise, another type of noise called the **algorithmic noise** is introduced. The algorithmic noise is caused in the key recovery algorithm of side-channel analysis. For a cryptographic device, the measured side-channel traces could correspond to the information of a lot of signal transitions that occur at the same time. In side-channel analysis, the attackers cannot use all the related signal transitions in the key



**Figure 5.17** Hardware architecture of last round of AES-pprm1 and AES-comp

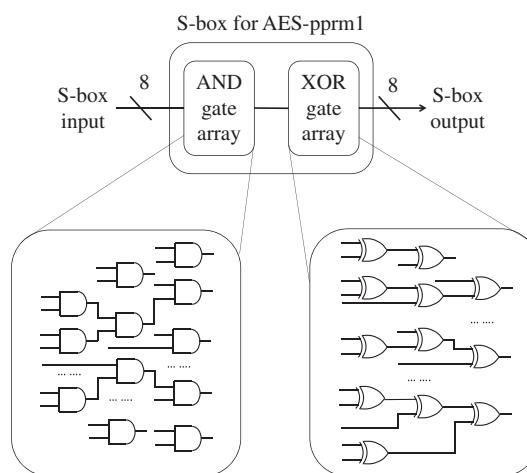
recovery owing to the limitation of the attacker's computational capability and knowledge of the device. Thus, only a part of the signal transitions is related to a specific attack following the divide-and-conquer approach. Simply speaking, the attack algorithm attempts to recover the key piece by piece, but the data measurement cannot measure the side-channel information for each piece independently. Thus, the algorithmic noise appears in the attack. The signal transitions that are not related to the current side-channel analysis are considered the algorithmic noise.

Existence of the algorithmic noise depends on the target hardware architecture and the key recovery algorithm. Namely, the algorithmic noise is not introduced by the measurement but by the key recovery algorithm.

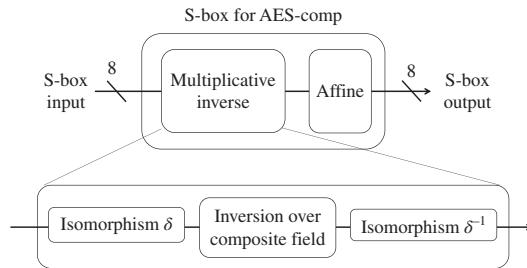
#### 5.3.4.2 S-Box Implementation of AES-pprm1 and AES-comp

The difference between AES-pprm1 and AES-comp is the implementation methods of their S-boxes. The S-box for AES-pprm1 is implemented using the 1 stage Positive Polarity Reed-Muller. As shown in Figure 5.18, the S-box of AES-pprm1 has a special structure that is an AND gate array followed by an XOR gate array. In the AND gate array and the XOR gate array, only the AND gates or the XOR gates are used. AES-pprm1 is used as an experimental implementation that is not likely to be used in a practical device. Owing to the structure of AES-pprm1 S-box, the side-channel leakage of AES-pprm1 can be described clearly with the side-channel LMs. This chapter uses AES-pprm1 to demonstrate the effectiveness of the introduced key recovery algorithms.

The S-box for AES-comp is implemented in composite-field arithmetic as shown in Figure 5.19. The AES S-box is constructed by two operations mathematically. The first one is the multiplicative inverse for the S-box input, and the second operation is an affine transformation. As introduced in Chapter 1, the multiplicative inverse in S-box is based on modular multiplicative inversion in  $GF(2)[x]/x^8 + x^4 + x^3 + x + 1$ . In AES-comp, the multiplicative inverse is separated into three steps. In the first step, the S-box input is



**Figure 5.18** Hardware architecture of AES-pprm1 S-box



**Figure 5.19** Hardware architecture of AES-comp S-box

converted from its original field into the composite field by an isomorphism function  $\delta$ . In the second step, the multiplication inverse in the composite-field is calculated. In the third step, the multiplication inverse is converted back to the original field by an inverse isomorphism function  $\delta^{-1}$ . The purpose of performing the multiplication inverse in the composite field is to achieve an S-box implementation with fewer gates and lower power consumption. AES-comp implementation is widely used in practical devices. Compared to AES-pprm1, it is more difficult to use an LM to match the power consumption characteristics for AES-comp.

In the following section, some tests are performed to see how the introduced three LMs match the real power consumption of AES-pprm1 and AES-comp. That is to say, with the real power traces and the value of the key and ciphertexts, the power traces are divided into different groups according to the intermediate values and the LM. Then it is checked whether the difference exists statistically for different groups of the power traces. In the following test, 65,536 power traces for each implementation are used. Each power trace has 10,000 sample points. The last round AES input,  $S_{10}^I$ , is the focused intermediate value.

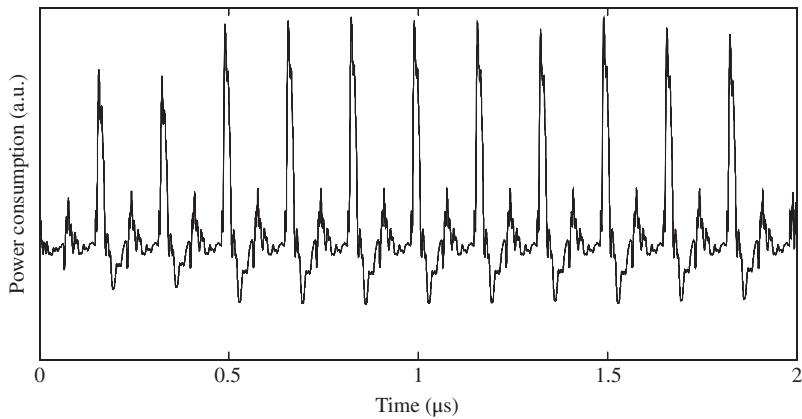
#### 5.3.4.3 How LMs Matches Power Traces of AES-pprm1

This section shows how much the single-bit model, the HW model, and the HD model match the power traces of AES-pprm1.

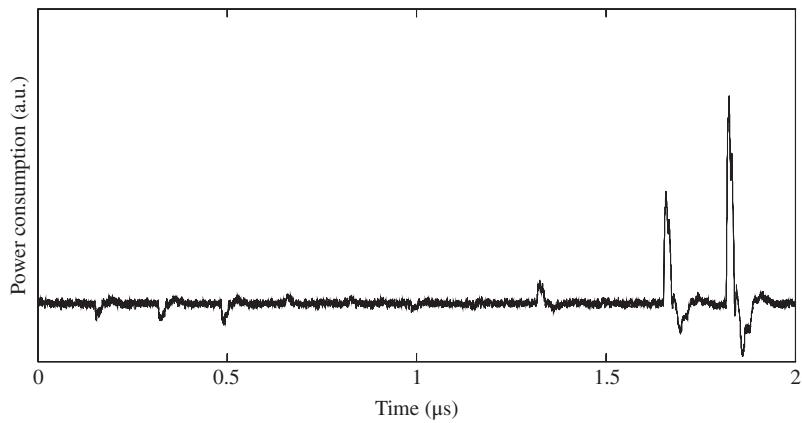
- **Single-bit model and AES-pprm1**

For the single-bit model, the most significant bit (MSB) of  $S_{10}^I$  is used for the group separation. The power traces are separated into two groups as the group with MSB of  $S_{10}^I = 1$  and the group with MSB of  $S_{10}^I = 0$ . Then, the mean traces for two groups of the power traces are calculated and plotted in the same figure as shown in Figure 5.20. If the single-bit model matches the real power traces, it is expected that one can see some difference between the mean traces of two groups of data.

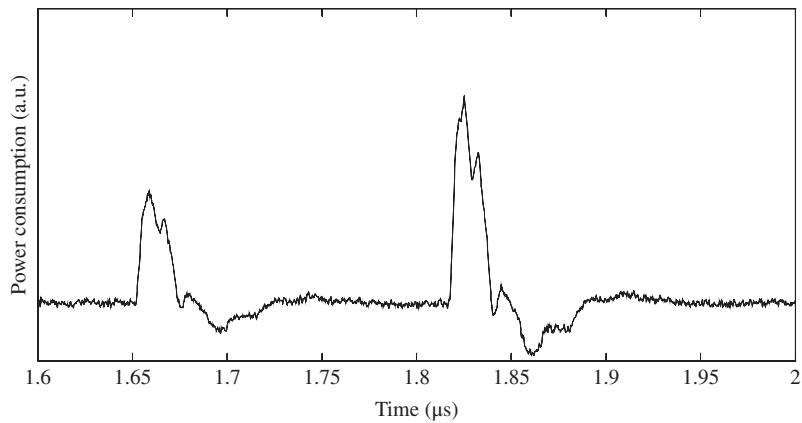
Figure 5.20 shows the mean traces of AES-pprm1 for two groups of power traces either MSB of  $S_{10}^I = 1$  or MSB of  $S_{10}^I = 0$ . From Figure 5.20, the two mean traces are overlapped with each other as the difference between them is too small. To confirm the details, the difference between two mean traces for AES-pprm1 is calculated and plotted as shown in Figure 5.21. There are two peaks around the last two clock cycles of the **difference of mean** trace, which is zoomed and shown in Figure 5.22. The above-mentioned result confirms that the power traces separated by the single-bit model do have a small difference in the power consumptions. Only when the difference of means (DoM) is calculated, the difference can be observed.



**Figure 5.20** Two mean traces of AES-pprm1 after group separation using single-bit model



**Figure 5.21** Difference between two mean traces of AES-pprm1 using single-bit model

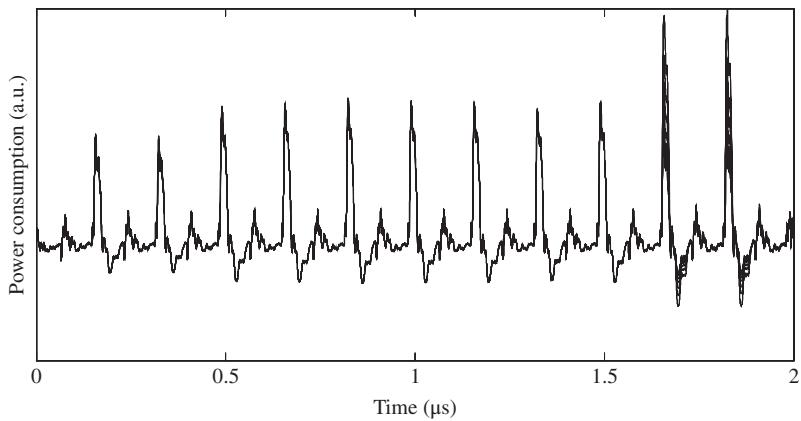


**Figure 5.22** Zoomed Figure 5.23 in last 2 clock cycles

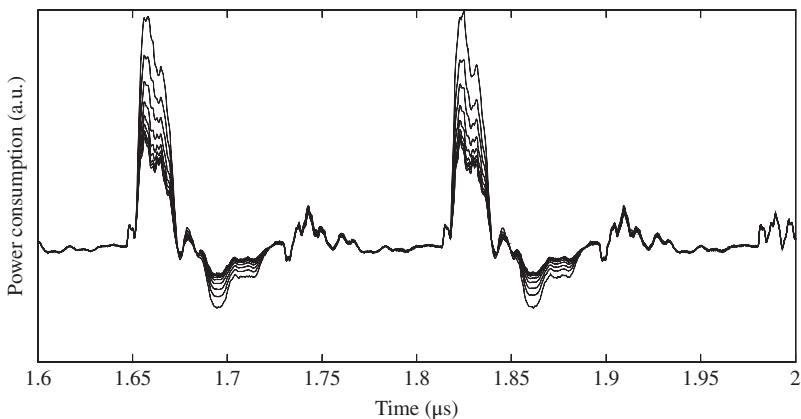
- **HW model and AES-pprm1**

For testing the HW model, a method that is based on solving the system of linear functions is used to get more accurate mean traces for nine groups of data corresponding to nine possible values of the HW. The details of the constructing and solving the linear functions are omitted in this chapter.

Figure 5.23 shows the nine mean traces of AES-pprm1 after the group separation using the HW model. Figures 5.24 and 5.25 show the Figure 5.23 zoomed in the last two clock cycles and Figure 5.23 zoomed in the peak around  $1.83\text{ }\mu\text{s}$  (microsecond), respectively. It can be seen that the nine mean traces for the sample points before  $1.5\text{ }\mu\text{s}$  show no difference at all as they correspond to the first nine AES rounds that are independent of the last AES round input  $S_{10}^I$ . For the last two clock cycles of AES calculation, the nine mean traces correspond



**Figure 5.23** Nine mean traces of AES-pprm1 after group separation using HW model



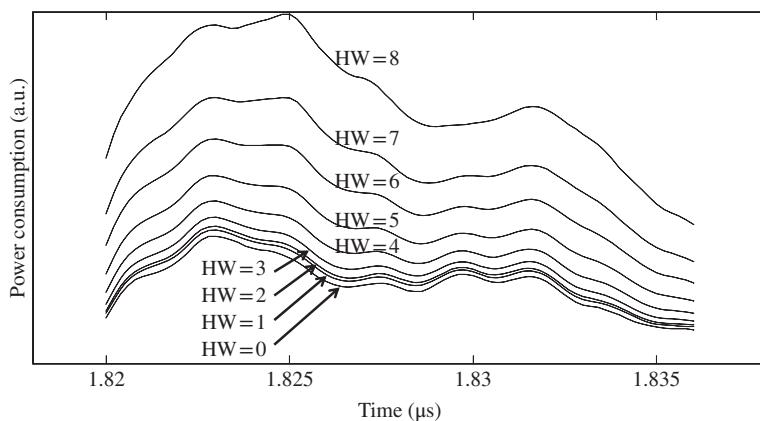
**Figure 5.24** Zoomed Figure 5.23 in last two clock cycles

to nine HW values show clear differences. In Figure 5.25, it is labeled and confirmed that a higher HW value always shows a higher result of the power consumption. This result implies that the HW model is generally a good LM when attacking the AES-pprm1 implementation.

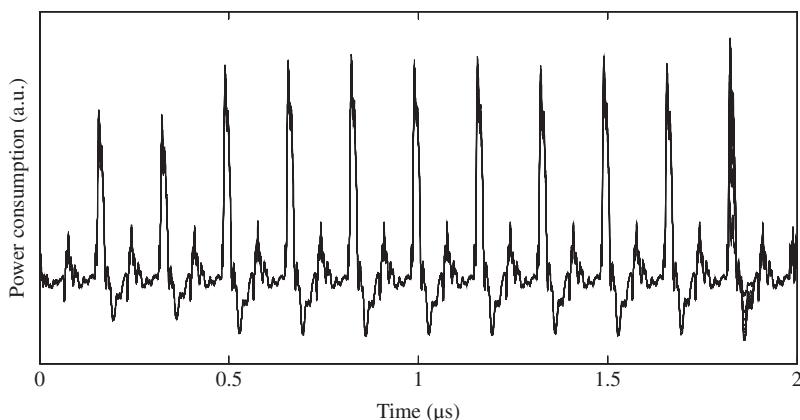
Interestingly, it is observed that the power consumptions for  $\text{HW} = 0$  to  $\text{HW} = 8$  have an increasing sequence of the difference between two consecutive HWs. For example, the difference between the power consumptions of  $\text{HW} = 8$  and  $\text{HW} = 7$  is much larger than that between the power consumption of  $\text{HW} = 1$  and  $\text{HW} = 0$ .

- **HD model and AES-pprm1**

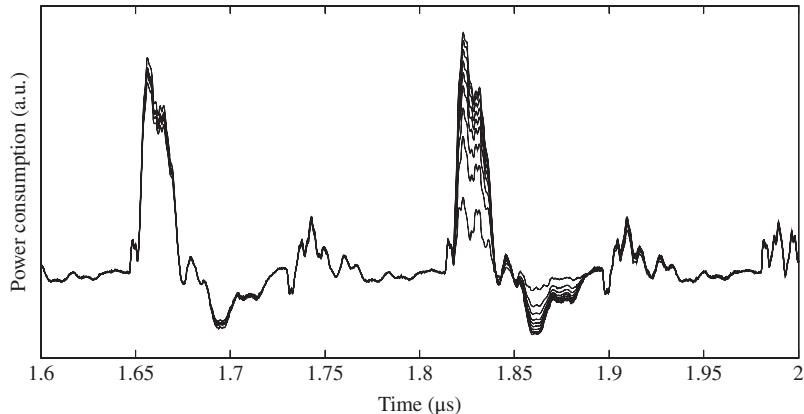
Following the same approach, the nine mean traces of AES-pprm1 after the group separation using HD model are shown in Figure 5.26. Figures 5.27 and 5.28 show that Figure 5.26 zoomed in the last two clock cycles and Figure 5.26 zoomed in the peak around  $1.83 \mu\text{s}$ , respectively.



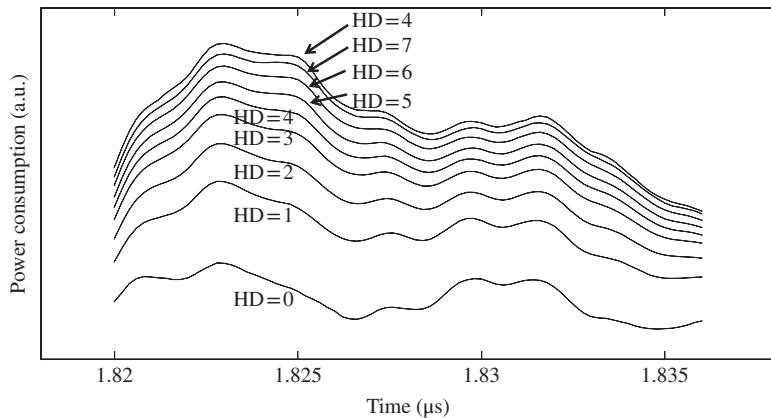
**Figure 5.25** Zoomed Figure 5.23 around  $1.83 \mu\text{s}$



**Figure 5.26** Nine mean traces of AES-pprm1 after group separation using HD model



**Figure 5.27** Zoomed Figure 5.26 in last two clock cycles



**Figure 5.28** Zoomed Figure 5.26 around  $1.83 \mu\text{s}$

As shown in Figures 5.26 and 5.27, the clear difference between the nine mean traces appears at the last clock cycle. In the last clock cycle, the last AES round input is updated to the ciphertext in DFFs. As labeled and confirmed in Figure 5.28, a higher HD always has a higher power consumption in the mean traces of various HD values for AES-pprm1. This result implies that the HD model is also a good LM when attacking the AES-pprm1 implementation. Interestingly, it is observed that the power consumptions for HD = 0 to HD = 8 have a decreasing sequence of the difference between two consecutive HWs.

#### 5.3.4.4 How LMs Matches Power Traces of AES-comp

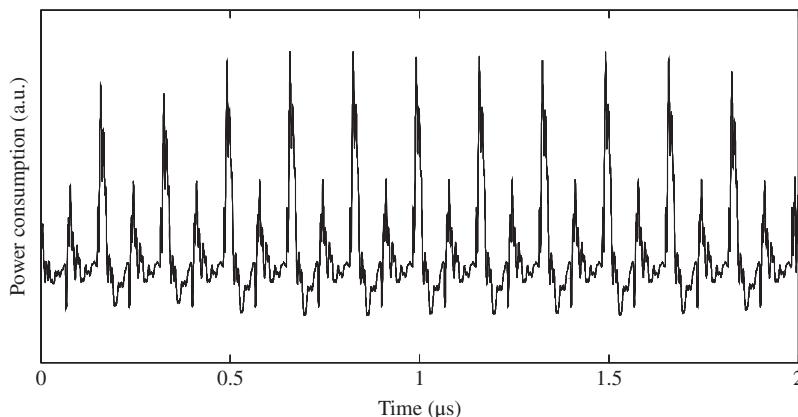
This section shows how much the single-bit model, the HW model, and the HD model match the power traces of AES-pprm1.

- **Single-bit model and AES-comp**

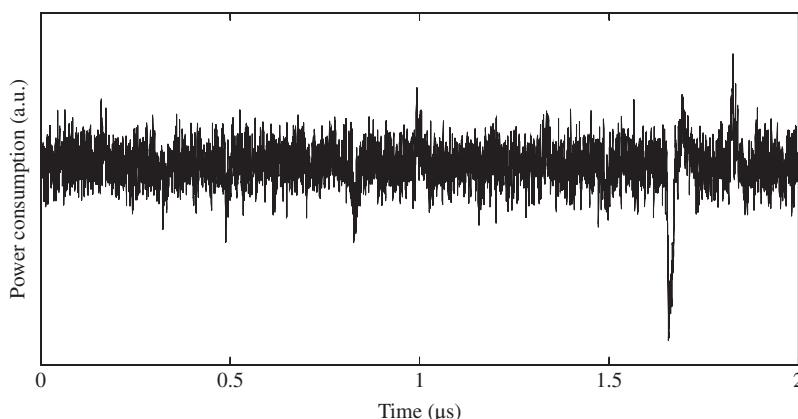
Following the same approach, two mean traces of AES-comp after the group separation using the value of the MSB of  $S'_{10}$  are shown in Figure 5.29. Similarly to the result of AES-pprm1, these two mean traces show almost no difference in Figure 5.29. To confirm the details, the difference between these two mean traces is calculated and plotted in Figure 5.30. The DoM zoomed in the last two clock cycles is shown in Figure 5.31, in which the difference peak in the second to the last clock cycle appears.

- **HW model and AES-comp**

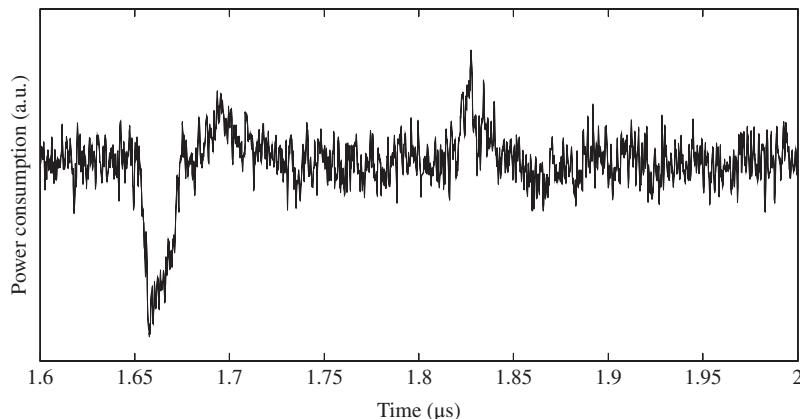
Similar to the AES-pprm1 implementation, the nine mean traces for the AES-comp after the group separation using HW model are shown in Figure 5.32. In the zoomed results as shown in Figures 5.33 and 5.34, it is confirmed that one of the HW values can be distinguished from others. As labeled in Figure 5.34, this distinguishable HW value is corresponding to



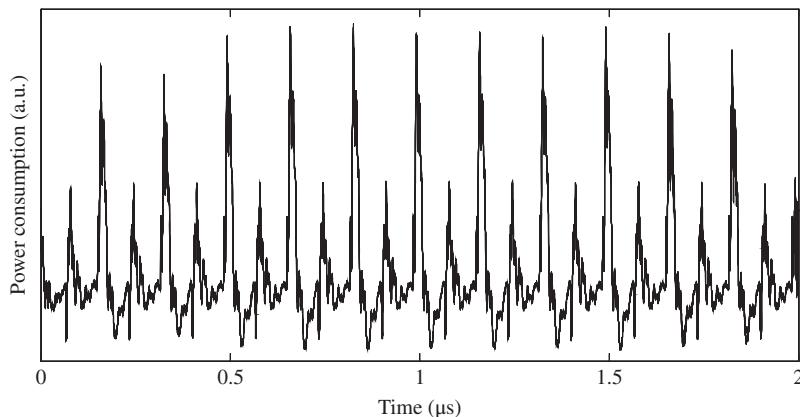
**Figure 5.29** Two mean traces of AES-comp after group separation using single-bit model



**Figure 5.30** Difference between two mean traces of AES-comp using single-bit model



**Figure 5.31** Zoomed Figure 5.23 in last two clock cycles

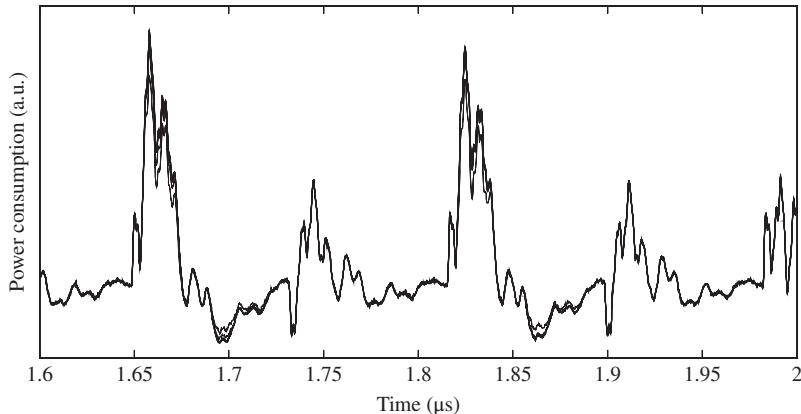


**Figure 5.32** Nine mean traces of AES-comp after group separation using HW model

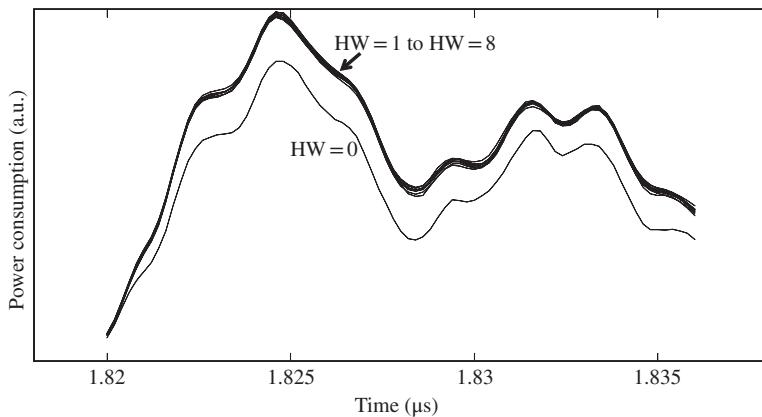
$HW = 0$ . Thus, the power consumption of AES-comp does not show a clear difference for different HWS of the S-box input except for the case when  $HW = 0$ . When the mean traces do not show the clear difference, it implies that the HW model is not a very good LM for AES-comp. However, as  $HW = 0$  shows a clear difference from others, power analysis that focuses on the zero value of the HW could be considered, which will be explained in detail later in this chapter. Note that the same pattern of the mean traces can be observed for the second to last clock cycle for AES-comp.

- **HD model and AES-comp**

Figures 5.35–5.37 show the nine mean traces of AES-comp after the group separation using the HD model. In Figure 5.35, the mean traces for nine HD values show some difference only in the last clock cycle. As labeled and shown in Figure 5.37, in the last clock cycle, one of the nine HD values is largely different from the others. It is confirmed that the



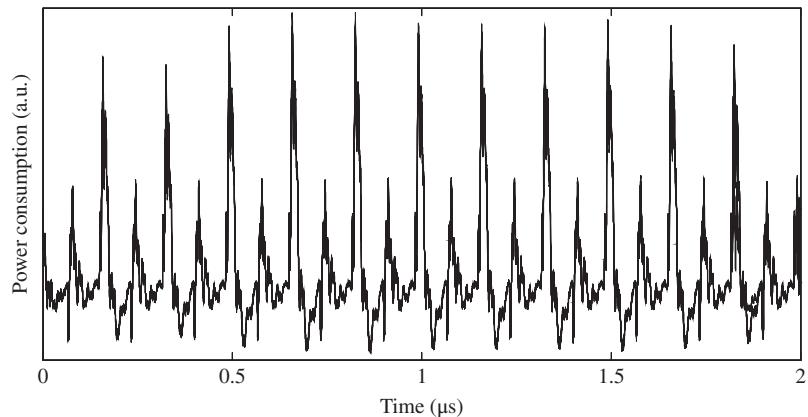
**Figure 5.33** Zoomed Figure 5.32 in last two clock cycles



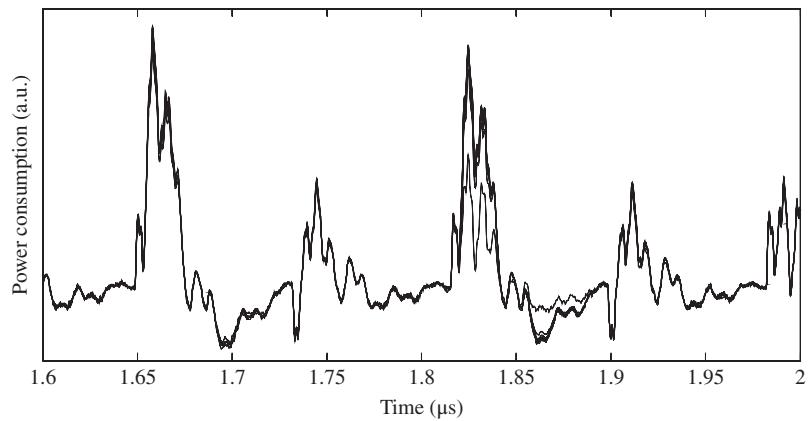
**Figure 5.34** Zoomed Figure 5.32 around 1.83 μs

most distinguishable mean trace is corresponding to  $\text{HD} = 0$ . It is also confirmed that in Figure 5.36, the mean traces for  $\text{HD} = 1$  to  $\text{HD} = 8$  show a proportional relationship with the  $\text{HD}$  value with very small differences. Thus for AES-comp implementation, the  $\text{HD}$  model could be a good model and the zero-value in  $\text{HD}$  also leads to a special leakage.

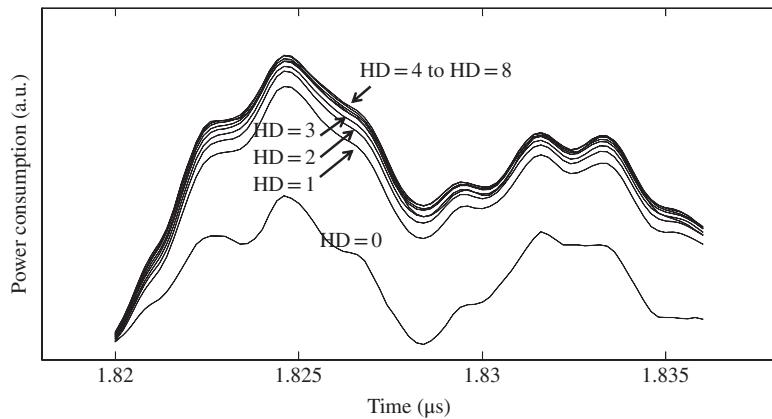
The above-mentioned observations show that in the known key setting, after the group separation according to the LM, the power consumption for different groups does appear some difference. These results imply that the effectiveness of the LM, thus the links among public data, correct key guess, and the power traces can be established. In the attack setting where the key is unknown, the remaining challenge is to see whether or not the credibility for the correct key can be distinguished from the wrong keys. Hereafter, the attack algorithms based on these LMs are explained in detail.



**Figure 5.35** Nine mean traces of AES-comp after group separation using HD model



**Figure 5.36** Zoomed Figure 5.35 in last two clock cycles



**Figure 5.37** Zoomed Figure 5.35 around 1.83 μs

### 5.3.5 Single-Bit DPA Attack on AES-128 Hardware Implementations

This section explains the DPA attack that is based on the single-bit model, which is called the **single-bit DPA attack**. The single-bit DPA attack is one of the first introduced SCAs, which can be applied to many devices owing to its very general LM. This section first explains the attack algorithm of the single-bit DPA attack. Then, the AES-pprm1 and AES-comp are used as the attack targets to demonstrate the single-bit DPA attack.

#### 5.3.5.1 SF, LM, and EF for Single-Bit DPA

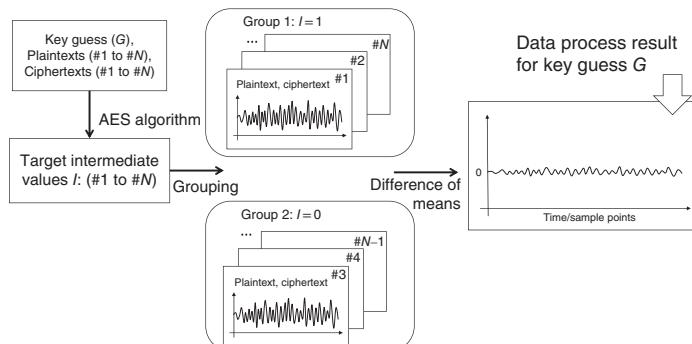
In single-bit DPA, the attackers focus on only 1 bit of the intermediate value as the target intermediate value. This target intermediate value can be calculated using public data and a guess of a key piece using the SF. The LM in single-bit DPA uses the value of the target intermediate value as the rough estimation of the power consumption, that is, the power consumption is 1 when the target intermediate value is 1 and the power consumption is 0 when the target intermediate value is 0.

The attackers confirm the relations between the rough estimation of power consumptions and the real power traces for each key guess using an EF. Recall that in Figures 5.21 and 5.30, the power consumption difference according to the single-bit model becomes clear when the DoM is calculated for both AES-pprm1 and AES-comp. Similarly, in the single-bit DPA attack, the calculation of **DoM** is used as the EF.

Specifically, the attackers separate the power traces into two groups according to the value of target intermediate value. Then, the attackers calculate the mean trace for each group of power traces and calculate the difference between two mean traces. This procedure is illustrated in Figure 5.38. For each possible candidate of the small key piece, the group separation and the calculation of DoM are performed.

#### 5.3.5.2 Why Key Recovery Could Succeed?

Let us consider the mechanism of the key recovery for single-bit DPA by considering the distributions of intermediate values after the group separation and the calculation of DoM.



**Figure 5.38** Data processing for each key guess in single-bit DPA

- For any intermediate value that is not the target intermediate value, the power consumption that is related to it belongs to the algorithmic noise. As these intermediate values are not used in the group separation, the same distribution of them is expected for each group of power traces. Thus, it is expected that they contribute to an almost zero difference in the trace of DoM.
- For the target intermediate value  $I$ , the result of group separation depends on the correctness of the key guess  $G$ .

When  $G$  is wrong, the attackers expect that the calculated target intermediate values are independent of the real values.<sup>1</sup> After the group separation based on the calculated target intermediate values, each group of power traces is expected to have the same distribution of the real values of  $I$ . Thus, it is expected that when  $G$  is wrong, the target intermediate value contributes to an almost zero difference in the trace of DoM.

When  $G$  is correct, all the calculated target intermediate values of  $I$  are correct. After the group separation based on the calculated target intermediate values, the power traces in a group either have all the  $I = 1$  traces or all the  $I = 0$  traces. For the target intermediate value, the difference of power consumption for processing  $I = 1$  and processing  $I = 0$  should appear in the trace of DoM. This difference has been confirmed for AES-pprm1 and AES-comp in Section 5.3.4.

As a summary, only when  $G$  is correct, the trace of DoM is expected to have some nonzero peaks. Thus, the attackers can identify the trace of DoM that has special nonzero peaks and identify the correct  $G$ . With the traces of DoM for all the key guesses, the  $G$  that corresponds to the largest absolute value of the DoM, that is,  $\arg \max_G |\text{DoM}_G|$ , should be considered the correct key.

Figure 5.39 shows an illustration of the key identification of single-bit DPA. For the correct  $G$ , some nonzero peaks are expected in DoM and for wrong key guesses an almost zero difference is expected in DoM. In practice, DoM also shows some nonzero peaks for the wrong key guesses due to two reasons. The first reason is the noise, both the measurement noise and algorithmic noise lead to nonzero values even for wrong key guesses. Another reason is that the target intermediate values for correct key and wrong keys are not perfectly independent. The point of DPA is that when enough power traces are used, the effect of the noise is going to be reduced to close to zero and the wrong key guesses will not have a bigger peak than the correct key. Only for the correct key, the group separation is perfect with regard to the value of the target intermediate value.

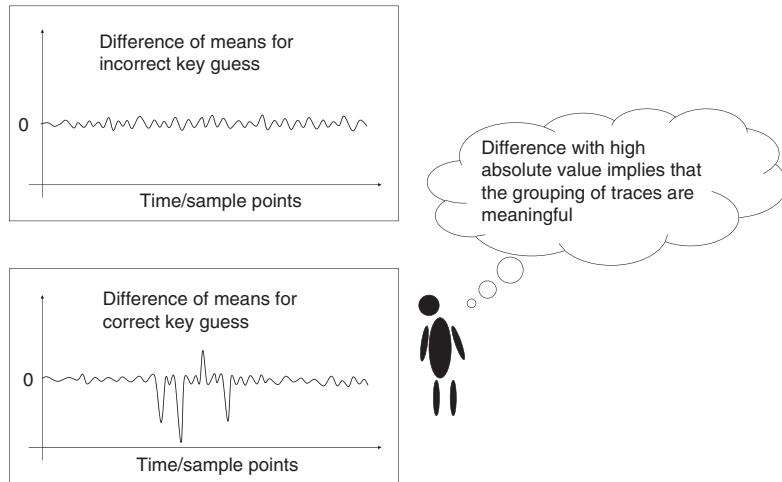
### 5.3.5.3 Single-Bit DPA Applied to AES-pprm1 and AES-comp

The single-bit DPA attack is applied to AES-pprm1 and AES-comp targeting  $sk_{10}[0]$ , in which the MSB of  $S_{10}^I[0]$  is used as the target intermediate value. The SF includes the AddRoundKey operation, inverse of the ShiftRows operation, and inverse of the SubBytes operation. The ShiftRows operation can be ignored here as the byte position is not changed for  $S_{10}^I[0]$ .

The key recovery algorithm of the single-bit DPA attack targeting  $sk_{10}[0]$  is shown in Algorithm 5.4. In step 3, the MSB of  $S_{10}^I[0]$  is calculated for the group separation of power

---

<sup>1</sup> In practice, the relationship of the calculated target intermediate values and the real ones is more complicated. Generally speaking, the independence assumption is true when the calculation from public data to the target intermediate value has a high nonlinearity.



**Figure 5.39** Key identification in single-bit DPA

---

#### Algorithm 5.4 Single-Bit DPA Targeting $sk_{10}[0]$

---

**Input:**  $N$  pairs of ciphertext and power trace:  $(C_1, W_1), (C_2, W_2), \dots, (C_N, W_N)$

**Output:** Recovered key piece:  $sk_{10}[0]$

```

1: for  $G = 0$  to  $255$  do
2:   for  $i = 1$  to  $N$  do
3:      $T_{i,G} \leftarrow$  MSB of( $S^{-1}(C_i[0] \oplus G)$ );
4:   end for
5:   for each sample point  $j = 1, 2, \dots, M$  of power trace do
6:      $DoM_{j,G} \leftarrow$  (Mean of  $W_{j,i}$  such that  $T_{i,G} = 1$ ) - (Mean of  $W_{j,i}$  such that  $T_{i,G} = 0$ );
7:   end for
8:    $DoMTrace_G \leftarrow \{DoM_{1,G}, DoM_{2,G}, \dots, DoM_{M,G}\}$ ;
9: end for
10:  $sk_{10}[0] \leftarrow \arg \max_G |DoMTrace_G|$ ;
11: return  $sk_{10}[0]$ ;

```

---

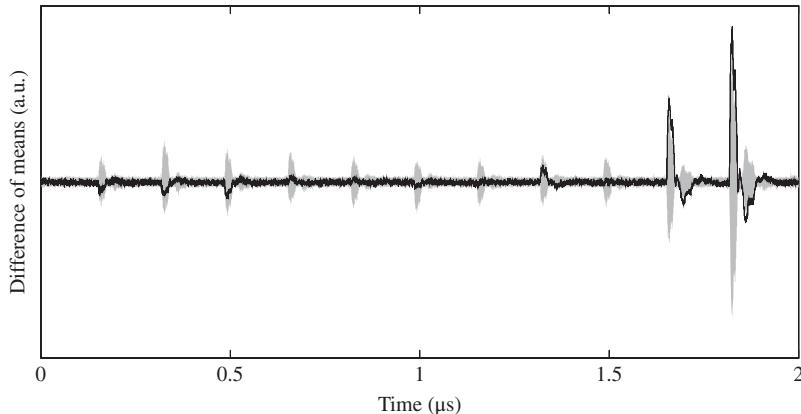
traces. In steps 5 to 8, the trace of DoM is calculated. The loop in step 1 ensures that the trace of DoM is obtained for each possible key candidate. Step 10 identifies the correct key by finding DoM trace that has the largest nonzero absolute value.

The attack result of the single-bit DPA attack on AES-pprm1 targeting  $sk_{10}[0]$  is shown in Figures 5.40 and 5.41.<sup>2</sup> The traces of DoM for all the 256 key candidates are plotted in Figure 5.40, where “a.u.” stands for arbitrary unit. The black line corresponds to the correct key guess. The area in gray corresponds to the traces of DoM for all the wrong key values.

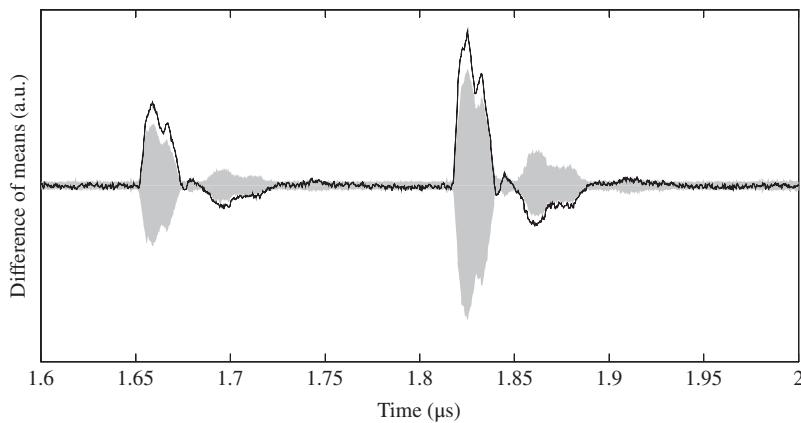
In Figure 5.40, the traces of DoM include the entire AES calculation of 11 clock cycles. As shown in Figure 5.40, the first nine cycles do not help recovering the correct key as the

---

<sup>2</sup> For all the attacks shown in this chapter, all the available 65,536 traces are used.



**Figure 5.40** Single-bit DPA result targeting  $sk_{10}[0]$  for AES-pprm1

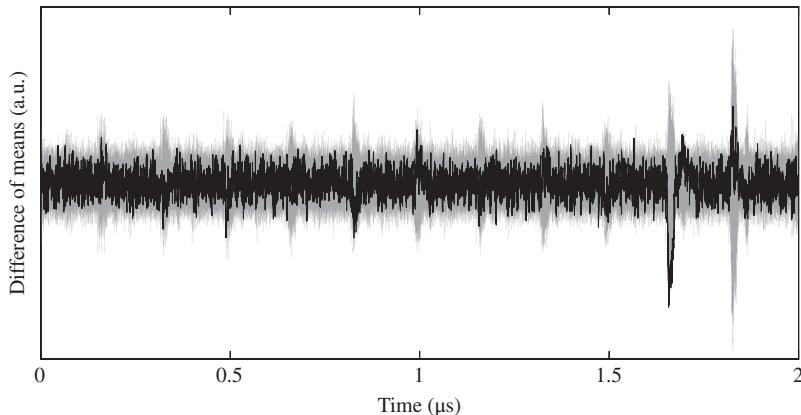


**Figure 5.41** Zoomed Figure 5.40 in last two clock cycles

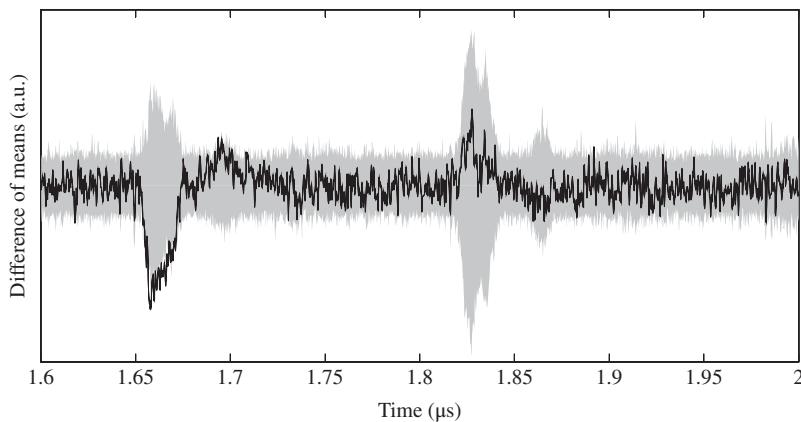
target intermediate value has no relations with the first nine round functions. To make the attack results of the related timing clear, Figure 5.40 zoomed at the last two clock cycles is shown in Figure 5.41. The correct key leads to the highest nonzero value of DoM among all the key candidates. Thus, the key recovery of  $sk_{10}[0]$  is successful for AES-pprm1 using single-bit DPA. There are also some nonzero peaks exist for the wrong key guesses; however, their peaks are smaller than the peak of the correct key guess.

Similarly, the attack result of the single-bit DPA attack on AES-comp is shown in Figures 5.42 and 5.43. For AES-comp, the black line of the correct key shows the highest absolute value in the traces of DoM at the second to the last clock cycle. However, it is not clear to identify the trace of DoM for the correct key from those results of the wrong key guesses. In this case, the attackers have to either use more power traces to reduce the noise or select a few key candidates with the highest DoM as the attack results.

The single-bit DPA attack already confirmed the possibility of the key recovery on practical implementations of AES. The same procedure of Algorithm 5.4 can be repeated to recover



**Figure 5.42** Single-bit DPA result targeting  $sk_{10}[0]$  for AES-comp



**Figure 5.43** Zoomed Figure 5.42 in last two clock cycles

other key bytes of  $sk_{10}$ . One can also try to find out the least number of power traces to recover all the key bytes. These analyses are related to the attack efficiency, while this chapter mainly focuses on the attack effectiveness. Thus, the details of the full-key recovery are omitted. Hereafter, for each introduced power analysis algorithm, only the key recovery result of  $sk_{10}[0]$  is demonstrated.

**Exercise 5.3** Discuss the reasons why the absolute value calculation is necessary in step 10 in Algorithm 5.4. Consider the case where step 6 in Algorithm 5.4 becomes  $DoM_{j,G} \leftarrow (\text{Mean of } W_{j,i} \text{ such that } T_{i,G} = 0) - (\text{Mean of } W_{j,i} \text{ such that } T_{i,G} = 1)$ .

**Table 5.1** A cipher example for attack simulation

Input $I$	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
S-box output	c	0	f	a	2	b	9	5	8	3	d	7	1	e	6	4
Ciphertext $C$	0	c	3	6	e	7	5	9	4	f	1	b	d	2	a	8
MSB of $I$	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
HW of $I$	0	1	1	2	1	2	2	3	1	2	2	3	2	3	3	4

**Exercise 5.4** Assume that there is a cipher, whose last round simply consists of a 4-bit S-box shown in Table 1.5 and an XOR with 4-bit of key. Assume that the subkey in the last round is  $c$ , and the power consumption of this cipher is equal to the MSB of the S-box input.

Run a single-bit DPA attack using the MSB of S-box input  $I$  as the target intermediate value. The relation between the last round input  $I$  and power consumption (MSB of  $I$ ) is shown in Table 5.1. Confirm that the correct key value  $c$  is recovered.

**Exercise 5.5** Assume that there is a cipher, whose last round simply consists of a 4-bit S-box shown in Table 1.5 and an XOR with 4-bit of key. Assume that the subkey in the last round is  $c$ , and the power consumption of this cipher is equal to the HW of the S-box input.

Run a single-bit DPA attack using the MSB of S-box input  $I$  as the target intermediate value. The relation between the last round input  $I$  and power consumption (HW of  $I$ ) is shown in Table 5.1. Confirm that the correct key value  $c$  is recovered.

**Exercise 5.6** Consider how to extend the single-bit DPA attack in Exercise 5.5 to a two-bit DPA attack? For example, the first two MSBs are used as the target intermediate value.

With the 2-bit target intermediate value, how to separate the power traces into groups and how to calculate Dom? Or other methods could be considered?

### 5.3.6 Attacks Using HW Model on AES-128 Hardware Implementations

This section explains the DPA attacks using the HW model, which is called the **HW-model-based DPA attack** in this book. Recall that in single-bit DPA, the SF calculates

1 byte of the last AES round input and uses its MSB as the target intermediate value. The rest 7 bits of the calculated last AES round input are not used in the LM. By effectively extending the single-bit model to a multiple-bit model such as the HW model, the HW-model-based DPA attack can be considered.

For the HW-model-based DPA attack, the target intermediate value is an 8-bit value and the LM is the HW model. The HW model assumes that the power consumption is proportional to the HW of the intermediate value. In other words, the HW model assumes that there is a linear dependency between the HW and the power consumption, which is a correlation between the HW and the power consumption. Following the HW model, the HW of target intermediate values is used as the rough estimation of power consumption. To extract the correlation between the rough estimation of power consumption and the real power traces, the correlation calculation is suitable to be the EF.

For the correlation calculations, the Pearson's correlation coefficient is the most used one. For two data sequences  $X_i$  and  $Y_i$ , where  $i = 1, 2, \dots, n$ , the correlation coefficient can be calculated as

$$\rho(X, Y) = \frac{\sum_{i=1}^n (X_i - \bar{X})(Y_i - \bar{Y})}{\sqrt{\sum_{i=1}^n (X_i - \bar{X})^2 \cdot \sum_{i=1}^n (Y_i - \bar{Y})^2}}. \quad (5.2)$$

Assume that the HW model matches the power consumed by the target device. Let us consider the reason that the HW-model-based DPA attack can recover the correct key. When the key guess is correct, the calculated target intermediate values are the real ones. Thus, the calculated HW and the real power traces should lead to a high correlation. When the key guess is wrong, the calculated target intermediate values are expected to be independent of the real ones; thus, the calculated HW has almost zero correlation with the power traces. As a summary, it is expected that the correct key will lead to the peak in the correlations results.

### 5.3.6.1 HW-Model-Based DPA Attack Applied to AES-pprm1 and AES-comp

For the HW-model-based DPA attack on AES-pprm1 and AES-comp targeting  $sk_{10}[0]$ , the target intermediate value is an 8-bit value as  $S_{10}^I[0]$ . For the HW-model-based DPA targeting  $sk_{10}[0]$ , the key recovery algorithm is shown in Algorithm 5.5. Step 3 is the SF and the LM that uses the HW of  $S_{10}^I[0] = S^{-1}(C_i[0] \oplus G)$  as the rough estimation of the power consumption. Steps 5 to 9 calculate the correlation coefficients between the rough estimation of the power consumption and the real power traces at each sample point. For a key guess  $G$ , the correlation coefficients corresponding to all the sample points become a trace of correlation coefficients denoted by  $\text{CorTrace}_G$  in step 10. Finally,  $G$  that leads to the largest value shown in the traces of correlation coefficients is selected as the attack result.

The same power traces of AES-pprm1 and AES-comp are used to demonstrate the attack. The attack result on AES-pprm1 is shown in Figures 5.44 and 5.45. As shown in Figure 5.45, for AES-pprm1, the correct key can be clearly distinguished from others in the last two clock cycles. Compared to the attack result of single-bit DPA, the HW-model-based DPA attack shows an improvement. Recall that the group separation result for AES-pprm1 using HW model as shown in Figure 5.25, the mean traces of the power consumptions have a clear proportional relation against the HW. Thus, the HW-model-based DPA attack has a good attack result against the AES-pprm1.

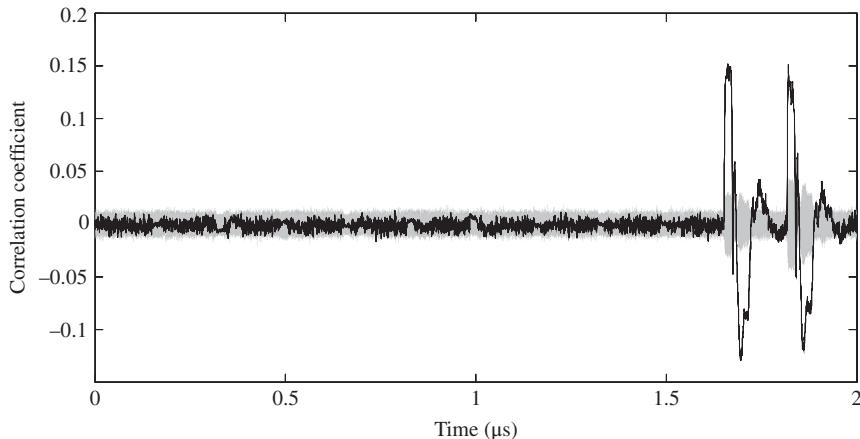
**Algorithm 5.5** HW Model Based DPA Targeting  $sk_{10}[0]$ **Input:**  $N$  pairs of ciphertext and power trace:  $(C_1, W_1), (C_2, W_2), \dots, (C_N, W_N)$ **Output:** Recovered key byte:  $sk_{10}[0]$ 

```

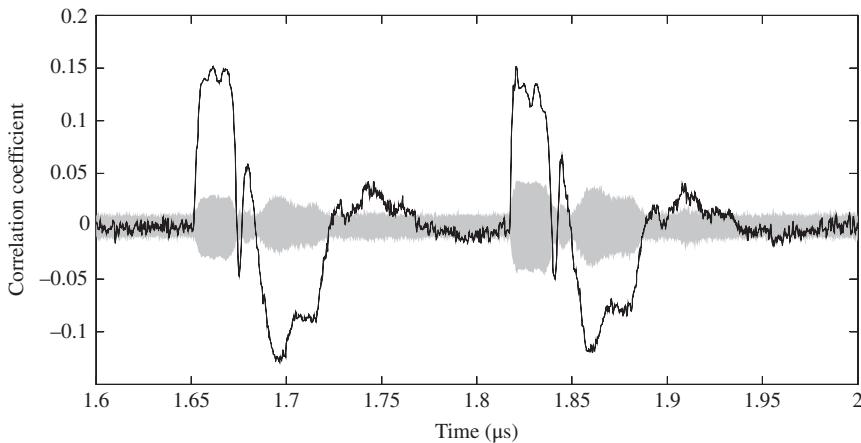
1: for  $G = 0$  to 255 do
2:   for  $i = 1$  to  $N$  do
3:      $T_i[0] \leftarrow \text{HW}(S^{-1}(C_i[0] \oplus G));$ 
4:   end for
5:   for each sample point  $j = 1, 2, \dots, M$  of power trace do
6:     Prepare a sequence  $X$ , where  $X_i = W_{i,j}$  for  $i = 0, 1, \dots, N;$ 
7:     Prepare a sequence  $Y$ , where  $Y_i = T_{i,G}$  for  $i = 0, 1, \dots, N;$ 
8:      $\text{Cor}_{j,G} \leftarrow \rho(X, Y);$ 
9:   end for
10:   $\text{CorTrace}_G \leftarrow \{\text{Cor}_{1,G}, \text{Cor}_{2,G}, \dots, \text{Cor}_{M,G}\};$ 
11: end for
12:  $sk_{10}[0] \leftarrow \arg \max_G \text{CorTrace}_G;$ 
13: return  $sk_{10}[0];$ 

```

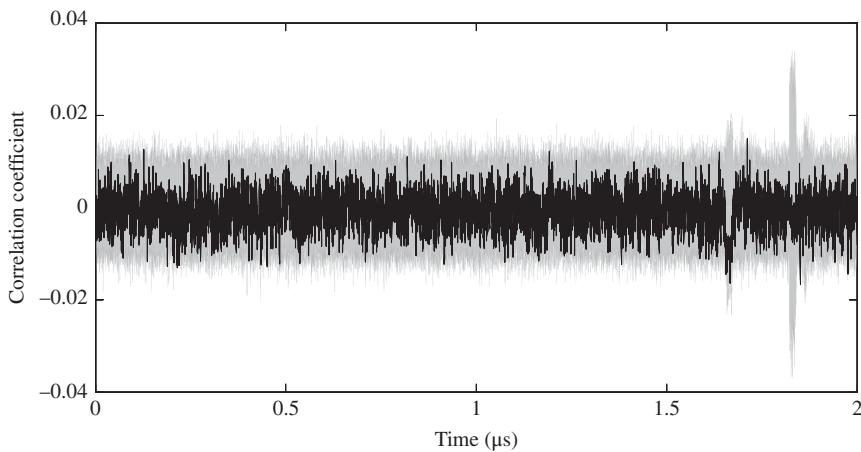
---

**Figure 5.44** HW-model-based DPA result targeting  $sk_{10}[0]$  for AES-pprm1

The attack result targeting  $sk_{10}[0]$  of AES-comp is shown in Figures 5.46 and 5.47. For AES-comp, the correct key cannot be distinguished from others, which implies a failure in the key recovery. Compared to the attack result of single-bit DPA, the HW-model-based DPA attack shows a worse result. Recall that the group separation result for AES-comp using HW model as shown in Figure 5.34, expect the mean trace for  $\text{HW} = 0$ , the mean traces for  $\text{HW} = 1$  to 8 show almost no difference. As the correlation between the rough estimation and the real power traces is too low for the correct key guess, the attack on AES-comp fails.

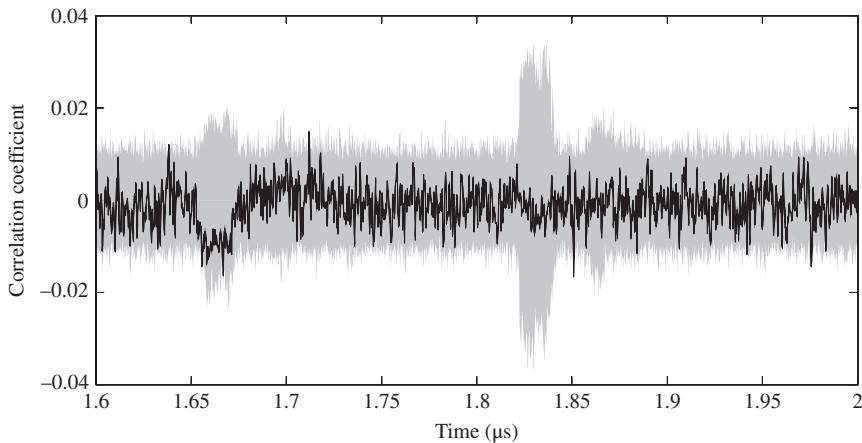


**Figure 5.45** Zoomed Figure 5.44 in last two clock cycles



**Figure 5.46** HW-model-based DPA result targeting  $sk_{10}[0]$  for AES-comp

**Exercise 5.7** Does the key recovery failure of the HW-model-based DPA on AES-comp imply that there is no side-channel leakage related to the last AES round input for AES-comp? Explain the reasons of your answer. Note that the answer to this question will become clear after reading this chapter.



**Figure 5.47** Zoomed Figure 5.46 in last two clock cycles

### 5.3.6.2 Attack Using Zero-Value Model

Recall that as shown in Figure 5.34, the mean trace for  $\text{HW} = 0$  actually shows a difference from the mean traces for  $\text{HW} = 1$  to 8. Thus, it could be useful to consider an attack that focuses on the zero-value, which is called **zero-value analysis**. Zero-value analysis takes zero of an 8-bit target intermediate value as a special value for a special side-channel leakage. In the zero-value analysis, the power traces are separated into two groups as the group with zero value and the group with nonzero values. The rest part of the zero-value analysis is the same with the HW-model-based DPA attack.

The reason that the zero value is special for the AES S-box is briefly explained as follows. For AES implementation, most of the side-channel leakage comes from the SubBytes operation, which is the only nonlinear operation. AES S-box is constructed by a multiplicative inverse calculation and an affine transformation. However, there is no multiplicative inverse of zero mathematically as any value multiples zero results zero. Thus, AES S-box with its input zero could show a special behavior. For AES S-box, the multiplicative inverse of zero is set to zero instead of being calculated to zero. Thus, for certain implementations such as the AES-comp S-box, the zero input to S-box could lead to a small power consumption as shown in Figure 5.34.

When targeting  $sk_{10}$  of AES, the target intermediate value is  $S_{10}^I[0]$ , which can be calculated as  $S_{10}^I[0] = S^{-1}(C[0] \oplus G)$ . For LM focused on the zero value, the rough estimation of the power consumption is whether the target intermediate value is a zero value. The correlation coefficient is used as the EF to extract the relations between rough estimation of the power consumption and the real power traces. The key recovery algorithm of zero-value analysis is shown in Algorithm 5.6. Note that DoM can also be used to replace the correlation coefficient as there are only two groups for the power separation. The key recovery is expected to be successful if the zero value in the target intermediate value shows a very special leakage.

The power data of AES-pprm1 and AES-comp are used to demonstrate the attack result of the zero-value analysis. The attack result on AES-pprm1 is shown in Figures 5.48 and 5.49, in which the correct key byte cannot be distinguished from wrong key values.

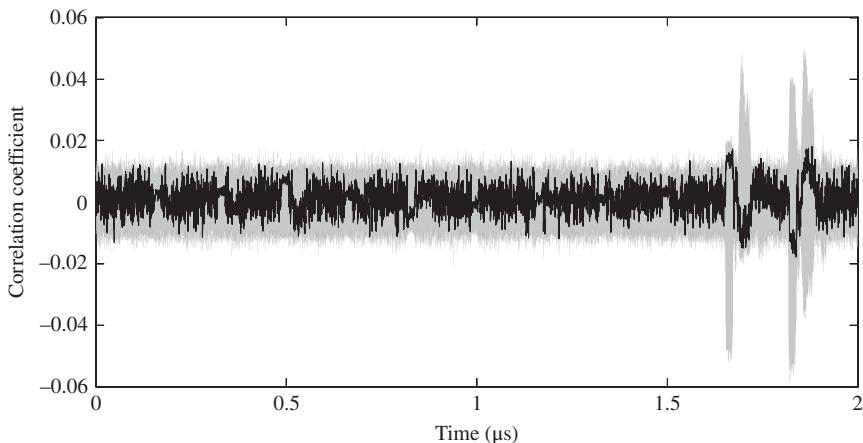
**Algorithm 5.6** Zero-Value Analysis Targeting  $sk_{10}[0]$ **Input:**  $N$  pairs of ciphertext and power trace:  $(C_1, W_1), (C_2, W_2), \dots, (C_N, W_N)$ **Output:** Recovered key byte:  $sk_{10}[0]$ 

```

1: for  $G = 0$  to  $255$  do
2:   for  $i = 1$  to  $N$  do
3:     if  $S^{-1}(C_i[0] \oplus G) = 0$  then
4:        $T_{i,G} \leftarrow 1$ ;
5:     else
6:        $T_{i,G} \leftarrow 0$ ;
7:     end if
8:   end for
9:   for each sample point  $j = 1, 2, \dots, M$  of power trace do
10:    Prepare a sequence  $X$ , where  $X_i = W_{i,j}$  for  $i = 0, 1, \dots, N$ ;
11:    Prepare a sequence  $Y$ , where  $Y_i = T_{i,G}$  for  $i = 0, 1, \dots, N$ ;
12:     $\text{Cor}_{j,G} \leftarrow \rho(X, Y)$ ;
13:   end for
14:    $\text{CorTrace}_G \leftarrow \{\text{Cor}_{1,G}, \text{Cor}_{2,G}, \dots, \text{Cor}_{M,G}\}$ ;
15: end for
16:  $sk_{10}[0] \leftarrow \arg \max_G \text{CorTrace}_G$ ;
17: return  $sk_{10}[0]$ ;

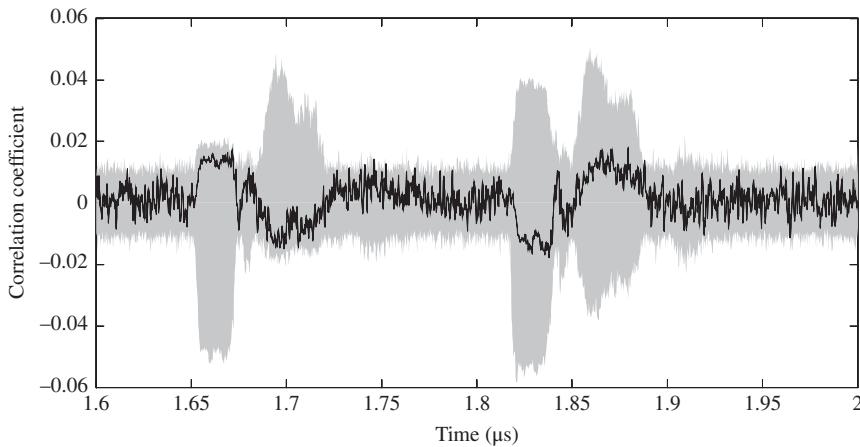
```

---

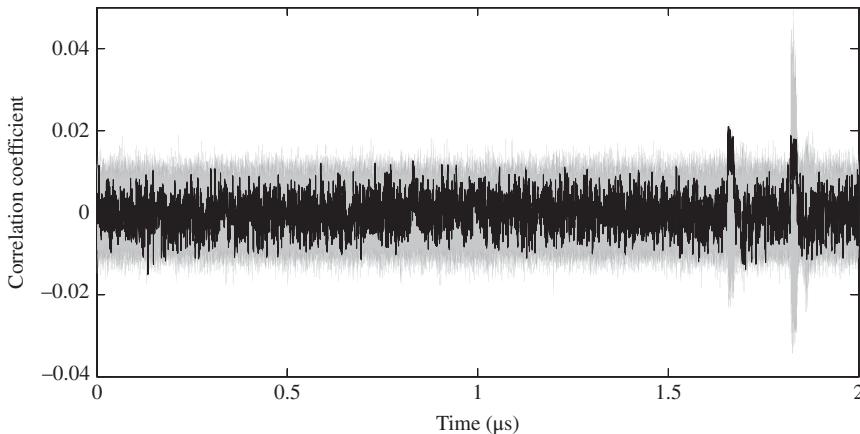
**Figure 5.48** Zero-value analysis result targeting  $sk_{10}[0]$  for AES-pprm1

The attack result on AES-comp is shown in Figures 5.50 and 5.51. Similarly, the correct key byte cannot be identified. The application of the zero-value attack does not show its advantage over previously introduced attacks.

Here, more detailed explanations of the attack result are as follows. First of all, for an 8-bit intermediate value that follows the uniform distribution, the probability of zero occurs with a probability of  $1/2^8$  or  $1/256$ . Therefore, only a small portion of the power traces,  $1/256$ ,



**Figure 5.49** Zoomed Figure 5.48 in last two clock cycles

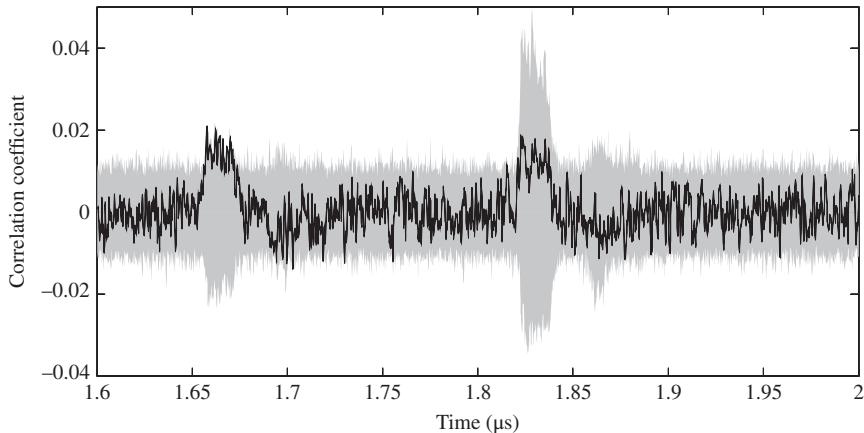


**Figure 5.50** Zero-value analysis result targeting  $sk_{10}[0]$  for on AES-comp

corresponds to the zero value. This fact increases the difficulty of a zero-value analysis. On the other hand, for AES-prrm1, the zero value of S-box input does not lead to a very special leakage as shown in Figure 5.25. Thus, the key recovery fails. For AES-comp, although  $HW = 0$  shows a low-power consumption, it cannot overcome the effects of the noise in the performed attack. Actually, by focusing on the second to the last clock cycle in Figure 5.51, the correct key shows a relatively high correlation in the result. It is expected that the correct key can show a distinguishable peak if more traces are used in the attack for AES-comp.

### 5.3.7 Attacks Using HD Model on AES-128 Hardware Implementations

This section explains the attacks using the HD model on AES-128 hardware implementations called **correlation power analysis (CPA)**. CPA was proposed in Brier *et al.* (2004).



**Figure 5.51** Zoomed Figure 5.50 in last two clock cycles

The naming of the attack comes as it is the first proposal for using the correlation coefficient in the EF. Compared to the HW-model-based DPA attack, the only difference of CPA is replacing the HW model by the HD model in the key recovery.

The HW model focuses on the value of an intermediate value at a specific timing, whereas the HD model focuses on the value transitions of an intermediate value. For hardware implementation, the HD model describes the power consumption for the signal transitions in combinatorial logics and the value update for DFFs.

### 5.3.7.1 CPA Applied to AES-pprm1 and AES-comp

For the CPA attack on AES-128 targeting  $sk_{10}[0]$ , the target intermediate value is still  $S_{10}^I[0]$ . Then, the HD between the last AES round input and the ciphertext is used as the rough estimation of the power consumption. In the hardware implementation of AES, the last AES round input will be updated to the ciphertext at a specific timing, which is expected to show a correlation with the HD. With a key guess, the target intermediate values and the HD can be calculated as the rough estimation of the power consumption. Then, the correlation calculation, which measures the linear dependence between two variables, is used to extract the relations between the rough estimations and the real power traces. When the key guess is wrong, the calculated target intermediate values are expected to be independent of the real ones. Thus, the calculated HDs are expected to have almost zero correlation with the power measurement data. When the key guess is correct, the calculated target intermediate values are the real ones. As confirmed in Figures 5.25 and 5.34, the calculated HDs should show a correlation with the real power traces. Therefore, it is expected that the correct key can be identified as it leads to the peaks in the correlations results.

The key recovery algorithm for the CPA attack targeting  $sk_{10}[0]$  is shown in Algorithm 5.7. The only difference between Algorithm 5.7 and Algorithm 5.5 is step 3 of them, specifically the LM. With the calculated  $S_{10}^I[0] = S^{-1}(C[0] \oplus G)$ , the rough estimation of power consumption is calculated as  $HW(S_{10}^I[0])$  in Algorithm 5.5 and as  $HD(S_{10}^I[0], C[0])$  in Algorithm 5.7. In steps 5 to 9, the correlation coefficient between the rough estimation and real power traces is

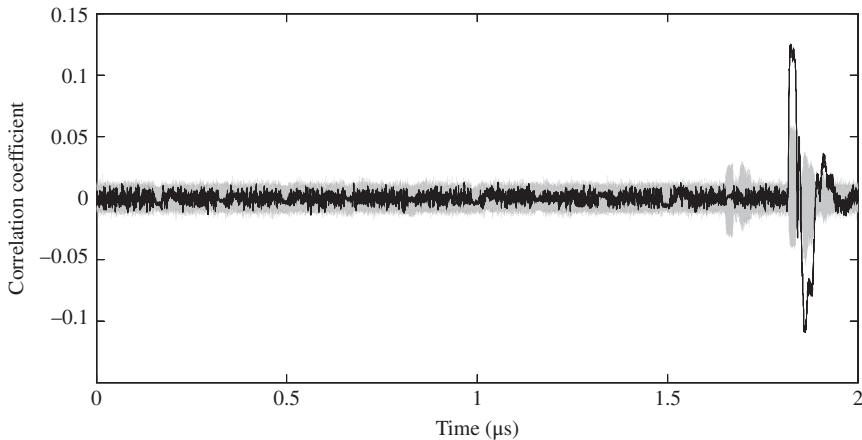
**Algorithm 5.7** Correlation Power Analysis Targeting  $sk_{10}[0]$ **Input:**  $N$  pairs of ciphertext and power trace:  $(C_1, W_1), (C_2, W_2), \dots, (C_N, W_N)$ **Output:** Recovered key byte:  $sk_{10}[0]$ 

```

1: for  $G = 0$  to 255 do
2:   for  $i = 1$  to  $N$  do
3:      $T_i \leftarrow HD(S^{-1}(C_i[0] \oplus G), C_i[0]);$ 
4:   end for
5:   for each sample point  $j = 1, 2, \dots, M$  of power trace do
6:     Prepare a sequence  $X$ , where  $X_i = W_{i,j}$  for  $i = 0, 1, \dots, N;$ 
7:     Prepare a sequence  $Y$ , where  $Y_i = T_{i,G}$  for  $i = 0, 1, \dots, N;$ 
8:      $\text{Cor}_{j,G} \leftarrow \rho(X, Y);$ 
9:   end for
10:   $\text{CorTrace}_G \leftarrow \{\text{Cor}_{1,G}, \text{Cor}_{2,G}, \dots, \text{Cor}_{M,G}\};$ 
11: end for
12:  $sk_{10}[0] \leftarrow \arg \max_G \text{CorTrace}_G;$ 
13: return  $sk_{10}[0];$ 

```

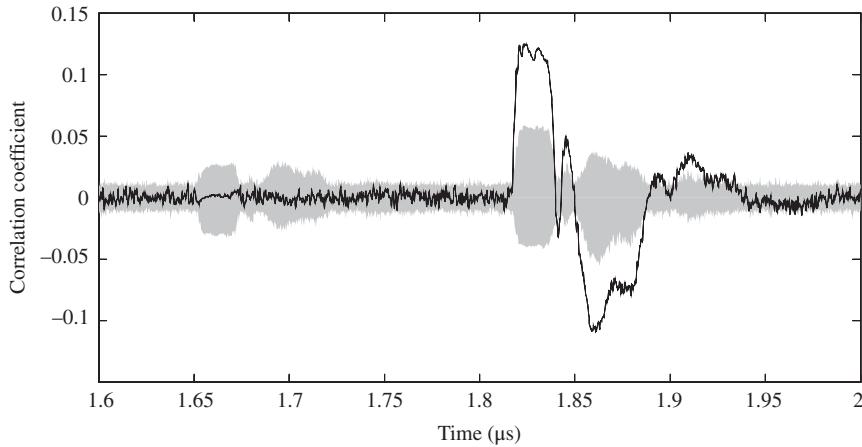
---

**Figure 5.52** CPA result targeting first key byte for AES-pprm1

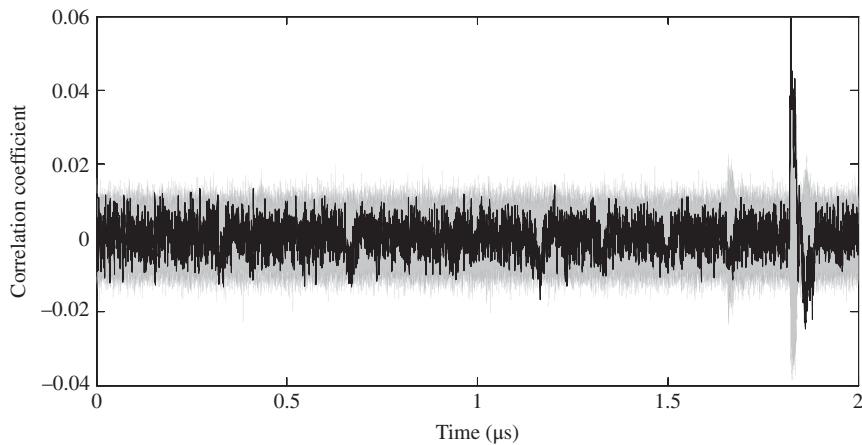
calculated at each sample point. For each key guess  $G$ , the calculated results become a trace of the correlation coefficients denoted by  $\text{CorTrace}_G$ . Then in step 12, the attackers observe all the traces of the correlations and identify  $sk_{10}[0]$  as  $G$  that shows the largest peak in  $\text{CorTrace}_G$ .

The CPA attack result targeting  $sk_{10}[0]$  of AES-pprm1 is shown in Figures 5.52 and 5.53. As shown in Figure 5.53, the CPA attack can successfully recover  $sk_{10}[0]$  for AES-pprm1 at the last clock cycle. This result fits the expectation as shown in Figure 5.28; the nine mean traces for different HD distance show clear difference among them. Thus, the HD model is a good attack model to describe the power consumption for AES-pprm1.

The CPA attack result targeting  $sk_{10}[0]$  of AES-comp is shown in Figures 5.54 and 5.55. As shown in Figure 5.55, the CPA attack can successfully recover  $sk_{10}[0]$  for AES-comp as well.



**Figure 5.53** Zoomed Figure 5.52 in last two clock cycles

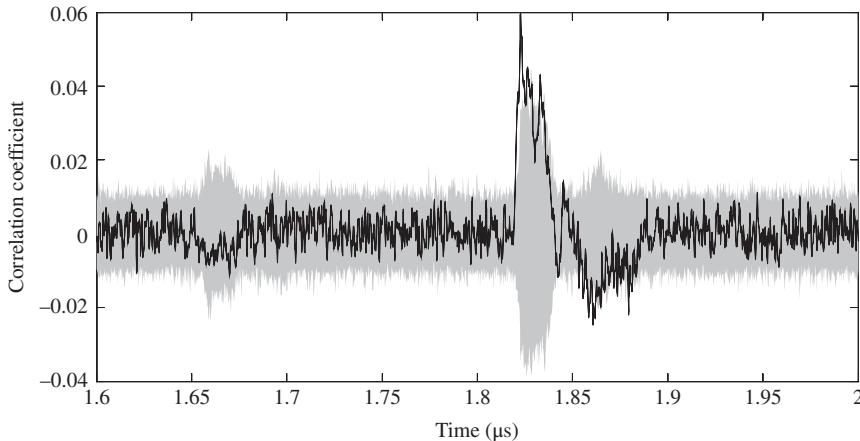


**Figure 5.54** CPA result targeting first key byte for AES-comp

For AES-comp, the correlation trace for the correct key is not very clear to be distinguished from others. Recall that as shown in Figure 5.37, the mean trace for HD = 0 shows a very low power consumption and the mean traces for HD = 1 to 8 shows a proportional relations with the HD with very small differences. Thus, the leakage related to the HD model for AES-comp is not as much as the AES-pprm1. The attack results confirmed this expectation.

### 5.3.7.2 Attack Using Clockwise Collision Model

As shown in Figure 5.37, the mean trace for HD = 0 shows a very low power consumption compared to others for AES-comp. Similarly, for AES-pprm1, the mean trace for HD = 0 also



**Figure 5.55** Zoomed Figure 5.54 in last two clock cycles

shows a relatively low-power consumption as shown in Figure 5.28. Thus, it is reasonable to consider the attack that takes zero value of the HD as a special value, which is called **clockwise collision analysis**. **Clockwise collision** refers to the case when an intermediate value has the same value at two consecutive clock cycles.

Owing to the nature of loop architecture, when a byte of the intermediate value has the same value in two clock cycles, there is no toggles in both DFFs and the combinatorial logics. As mentioned in Chapter 2, the stored values at DFFs are the inputs of the combinatorial logics at the beginning of a clock cycle. At the end of a clock cycle, the outputs of the combinatorial logics are stored at DFFs as the evaluation result of combinatorial logics. Consider the case that the DFFs have the same value in two consecutive clock cycles, the inputs of combinatorial logics will have the same value to evaluate in two consecutive clock cycles. In the second clock cycle, the combinatorial logics just keep the state at the end of the first clock cycle and there is no signal transition in the combinatorial logics. Thus, low-power consumption can be expected.

Clockwise collision analysis assumes that the zero value in HD leads to a dominantly special side-channel leakage. Thus, in clockwise collision analysis, the power consumption is separated into two groups according to whether or not the HD is a zero value. The rest part of the attack is the same with zero-value analysis.

For clockwise collision analysis targeting  $sk_{10}[0]$  on AES-pprm1 and AES-comp, the target intermediate value is  $S_{10}^I[0]$ . Whether or not the HD between  $S_{10}^I[0]$  and  $C[0]$  is a zero value is used as the rough estimation of power consumption. The correlation coefficient is used as the EF. The key recovery algorithm for clockwise collision analysis targeting  $sk_{10}[0]$  is shown in Algorithm 5.8. Step 3 in Algorithm 5.8 is the only different part from the attack algorithm for zero-value analysis as Algorithm 5.6.

Algorithm 5.8 is applied to the power traces of AES-pprm1 and AES-comp to show the attack efficiency for the clockwise collision analysis. The attack result on AES-pprm1 is shown in Figures 5.56 and 5.57. For AES-pprm1, the key recovery is successful at the last clock cycle of the AES. Compared to the CPA attack result, the correlation coefficient for key correct key becomes smaller for the clockwise collision analysis. This is because useful information

**Algorithm 5.8** Clockwise Collision Analysis Targeting  $sk_{10}[0]$ 

**Input:**  $N$  pairs of ciphertext and power trace:  $(C_1, W_1), (C_2, W_2), \dots, (C_N, W_N)$

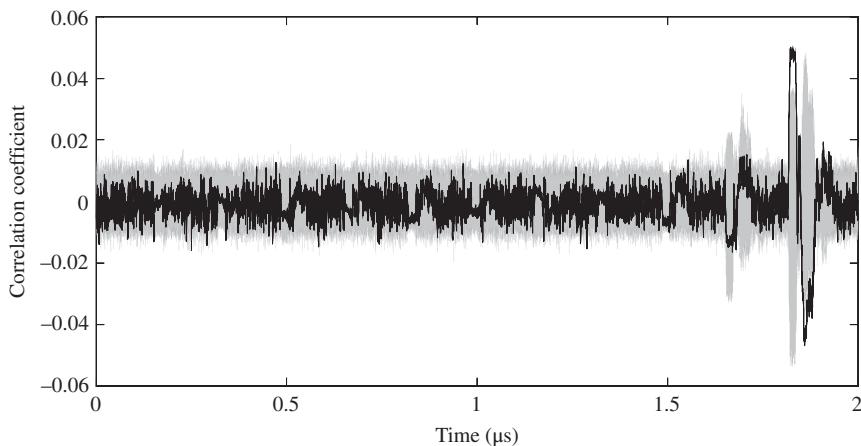
**Output:** Recovered key byte:  $sk_{10}[0]$

```

1: for  $G = 0$  to  $255$  do
2:   for  $i = 1$  to  $N$  do
3:     if  $\text{HD}(S^{-1}(C_i[0] \oplus G), C_i[0]) = 0$  then
4:        $T_{i,G} \leftarrow 1$ ;
5:     else
6:        $T_{i,G} \leftarrow 0$ ;
7:     end if
8:   end for
9:   for each sample point  $j = 1, 2, \dots, M$  of power trace do
10:    Prepare a sequence  $X$ , where  $X_i = W_{i,j}$  for  $i = 0, 1, \dots, N$ ;
11:    Prepare a sequence  $Y$ , where  $Y_i = T_{i,G}$  for  $i = 0, 1, \dots, N$ ;
12:     $\text{Cor}_{j,G} \leftarrow \rho(X, Y)$ ;
13:   end for
14:    $\text{CorTrace}_G \leftarrow \{\text{Cor}_{1,G}, \text{Cor}_{2,G}, \dots, \text{Cor}_{M,G}\}$ ;
15: end for
16:  $sk_{10}[0] \leftarrow \arg \max_G \text{CorTrace}_G$ ;
17: return  $sk_{10}[0]$ ;

```

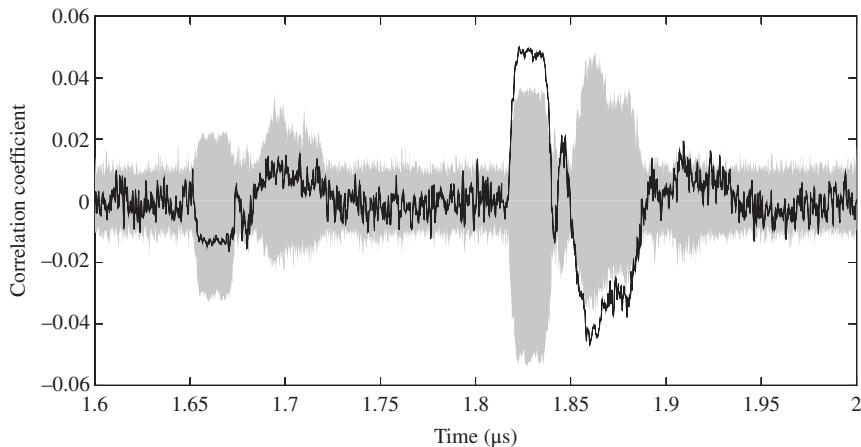
---



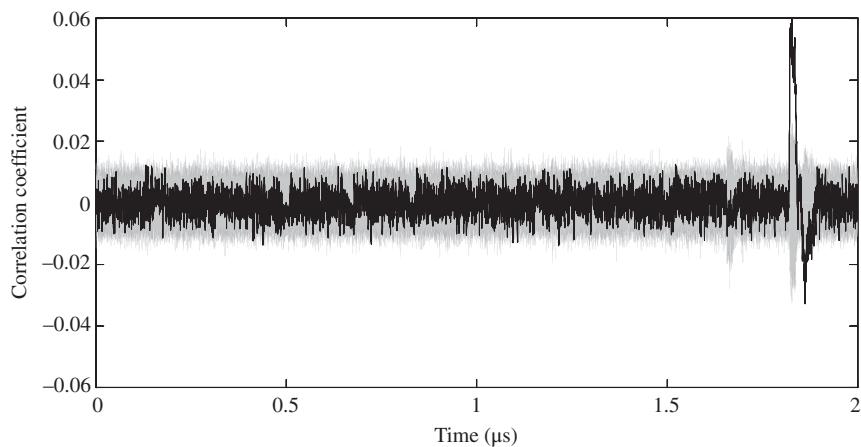
**Figure 5.56** Clockwise collision analysis result targeting  $sk_{10}[0]$  for AES-pprm1

related to  $\text{HD} = 1$  to  $\text{HD} = 8$  is ignored in clockwise collision analysis. From the perspective of attackers, the successful key recovery is achieved with a simplified LM.

The attack result on AES-comp is shown in Figures 5.58 and 5.59. The successful key recovery has been achieved, and interestingly, the attack result is better than that of the CPA attack. To understand the reason, recall the nine mean traces after the group separation as shown



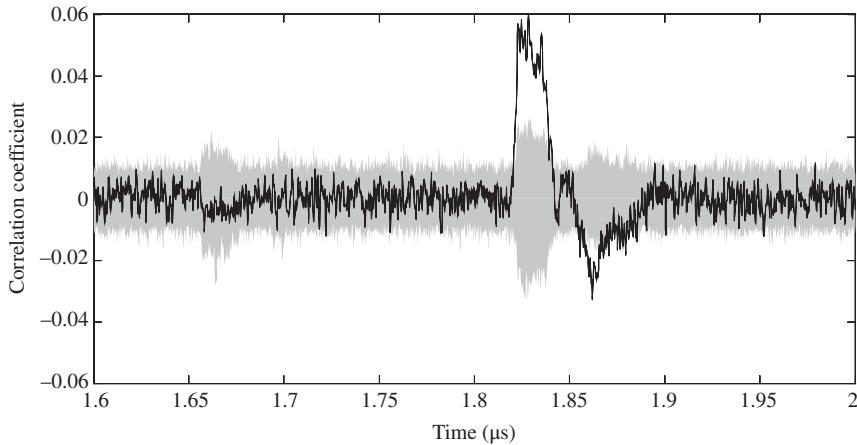
**Figure 5.57** Zoomed Figure 5.56 in last two clock cycles



**Figure 5.58** Clockwise collision analysis result targeting  $sk_{10}[0]$  for AES-comp

in Figure 5.37. Both CPA and clockwise collision analysis can succeed as they both fit the power consumption of AES-comp. However, as  $HD = 0$  has a dominate low-power consumption for AES-comp, the simplified LM has a better attack result for AES-comp.

**Exercise 5.8** Discuss and show how to make a customized HD model for AES-comp to achieve a better attack result? For example, for this customized HD model, with other parts unchanged, what kind of value should be used as the rough estimation of power consumption when  $HD = 0$ ?



**Figure 5.59** Zoomed Figure 5.58 in last two clock cycles

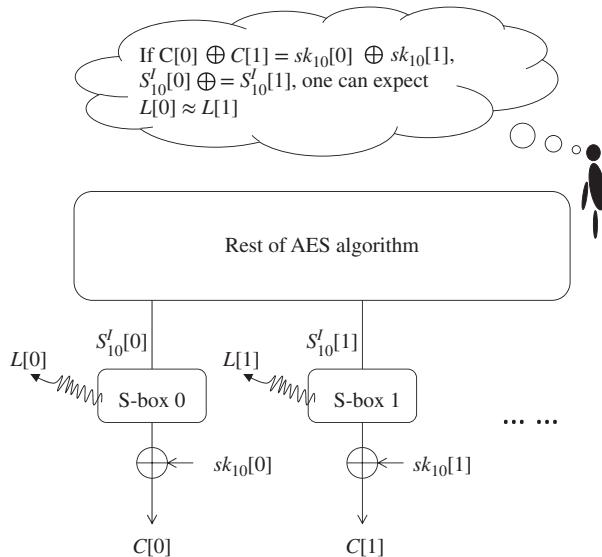
### 5.3.8 Attacks with Collision Model †

This section explains a power analysis that is based on the collision model. In the clockwise collision, the data collision of intermediate values at two clock cycles is considered, for example, the data collision between  $S_{10}^I[0]$  and  $C[0]$ . In the collision model explained in this section, the data collision between two pieces of an intermediate value at the same clock cycle is considered, for example  $S_{10}^I[0]$  and  $S_{10}^I[1]$ , which was for example discussed in Moradi *et al.* (2010). The biggest advantage of the attacks with the collision model is that the attackers do not need to assume the LM of target implementation. Recall that the LM describes rough relations between the intermediate value and the side-channel leakage. Instead of LM, the attackers make an assumption that when two pieces of intermediate value have the same value, the corresponding side-channel leakage for them is similar to each other.

Figure 5.60 shows the basic principle of the attacks using the collision model for a parallel implementation of AES. The basic assumption is that when two S-box inputs have the same value, that is,  $S_{10}^I[0] = S_{10}^I[1]$ , the side-channel leakage from those two S-boxes are similar to each other, that is,  $L[0] \approx L[1]$ . If  $S_{10}^I[0] = S_{10}^I[1]$ ,  $C[0] \oplus sk_{10}[0] = C[1] \oplus sk_{10}[1]$  and  $C[0] \oplus C[1] = sk_{10}[0] \oplus sk_{10}[1]$ . Thus, when the difference between two ciphertext bytes is equal to the difference between the key bytes, the similar leakage of S-boxes can be expected. Therefore, for AES, the attackers can construct a link among the difference of key bytes, the public data, and the case when two S-boxes have similar side-channel leakage. If the attackers can extract the information that two S-boxes have similar side-channel leakage from the power traces, the recovery of the difference between key bytes becomes possible.

There are a few variations of power analysis that use the collision model in the key recovery algorithms. This section only introduces the one that is called **correlation-enhanced power analysis collision attack**. Hereafter, the attack concept of the correlation-enhanced power analysis collision attack is explained.

In the attack with collision model on AES, with a key guess of  $sk_{10}[0]$ , the target intermediate value of  $S_{10}^I[0]$  can be calculated. As no LM is used in the attacks with collision model, there is no rough estimation of the power consumption mapped from the target intermediate value.



**Figure 5.60** Principle of data collision inside an intermediate value at last round of AES

However, the power traces can still be separated into groups directly according to the value of the target intermediate value. For an 8-bit target intermediate value, a group separation of 256 groups can be considered. After the group separation, the mean trace of each group can be calculated.

Note here, for any key guess, the result of the above-mentioned 256 mean traces is exactly the same. Thus, the above-mentioned mean traces after the group separation cannot be directly used to identify the key value, as they are irrelevant to the key value. Actually, the same group separation can be performed without key guess and using the ciphertext directly, as the mapping from  $C[0]$  to  $S'_{10}[0]$  is bijective. Thus, using  $C[0]$  directly, the attackers can separate the power traces into 256 groups and calculate the mean trace for each group. Each mean trace is going to correspond to a value of  $C[0]$ .

In the attack with collision model, the goal is set to identify the key byte difference by focusing on two target intermediate values. For example, the attackers focus on recovering  $sk_{10}[0] \oplus sk_{10}[1]$  by focusing on  $S'_{10}[0]$  and  $S'_{10}[1]$ . For  $C[1]$ , the group separation and the calculation of the mean traces can be performed as well. In the group separation for  $C[1]$ , each mean trace is going to correspond to a value of  $C[1]$ . Recall that when  $C[0] \oplus C[1] = sk_{10}[0] \oplus sk_{10}[1]$ ,  $L[0] \approx L[1]$ , which means the corresponding mean traces are similar.

If the attackers know the value of  $sk_{10}[0] \oplus sk_{10}[1]$ , for a mean trace for a value of  $C[0]$ , the attackers can find a corresponding value of  $C[1]$  such that  $C[0] \oplus C[1] = sk_{10}[0] \oplus sk_{10}[1]$  and the corresponding mean traces are similar to each other. There are 256 possible values for  $C[0]$ , so the attackers can find 256 pairs of corresponding mean traces that should be similar to each other. The similarity of 256 pairs of corresponding mean traces is essentially the linear dependency between two sequences of data, which can be verified by calculating the correlations between them.

Of course before the attack, the attackers do not know the value of  $sk_{10}[0] \oplus sk_{10}[1]$ . Then the solution is the same to other side-channel analysis, which is to exhaustively test all the

---

**Algorithm 5.9** Correlation-Enhanced Power Analysis Collision Attack Targeting  $sk_{10}[0] \oplus sk_{10}[1]$ 


---

**Input:**  $N$  pairs of ciphertext and power trace:  $(C_1, W_1), (C_2, W_2), \dots, (C_N, W_N)$

**Output:** Recovered key byte difference:  $sk_{10}[0] \oplus sk_{10}[1]$

```

1: for  $i = 1$  to  $N$  do
2:   In group separation for  $C[0]$ , move the power trace  $W_i$  to the group  $C_i[0]$ ;
3:   In group separation for  $C[1]$ , move the power trace  $W_i$  to the group  $C_i[1]$ ;
4: end for
5: for  $i = 0$  to 255 do
6:   MeanTrace $1_i \leftarrow$  Mean trace of group  $i$  in group separation for  $C[1]$ ;
7:   MeanTrace $2_i \leftarrow$  Mean trace of group  $i$  in group separation for  $C[2]$ ;
8: end for
9: for  $\Delta G = 0$  to 255 do
10:   for each sample point  $j = 1, 2, \dots, M$  of power trace do
11:     Prepare a sequence  $X$ , where  $X_i = \text{MeanTrace}1_i$  for  $i = 0, 1, \dots, 255$ ;
12:     Prepare a sequence  $Y$ , where  $Y_i = \text{MeanTrace}2_{i+\Delta G}$  for  $i = 0, 1, \dots, 255$ ;
13:      $\text{Cor}_{j,\Delta G} \leftarrow \rho(X, Y)$ ;
14:   end for
15:    $\text{CorTrace}_{\Delta G} \leftarrow \{\text{Cor}_{1,\Delta G}, \text{Cor}_{2,\Delta G}, \dots, \text{Cor}_{M,\Delta G}\}$ ;
16: end for
17:  $sk_{10}[0] \oplus sk_{10}[1] \leftarrow \arg \max_G \text{CorTrace}_G$ ;
18: return  $sk_{10}[0] \oplus sk_{10}[1]$ ;

```

---

possible values of  $sk_{10}[0] \oplus sk_{10}[1]$ . So in correlation-enhanced power analysis collision attack, the attackers first guess the value of  $sk_{10}[0] \oplus sk_{10}[1]$  as  $\Delta G$ . Under a key guess  $\Delta G$ , for a value of  $C[0]$ , the corresponding  $C[1]$  such that  $C[1] = C[0] \oplus \Delta G$  is found to become a pair. At the same time, the corresponding mean traces become a pair. For 256 possible values of  $C[0]$ , 256 pairs of  $C[0]$  and  $C[1]$  can be found, so do the mean traces. Then the correlation is calculated to verify the similarity between pairs of the mean traces as the credibility of the current key guess  $\Delta G$ . Finally, the  $\Delta G$  that corresponds to the highest credibility is selected as the attack result.

The attack algorithm of correlation-enhanced power analysis collision attack on AES targeting  $sk_{10}[0] \oplus sk_{10}[1]$  is shown in Algorithm 5.9. The attackers need to perform the group separation of the power consumptions two times, that is, group separation for  $C[0]$  and group separation for  $C[1]$  in steps 2 and 3 in Algorithm 5.9. For group separation for  $C[0]$ , the power traces are separated into 256 groups according to the value of  $C[0]$ . For group separation for  $C[1]$ , the power traces are separated into 256 groups according to the value of  $C[1]$ . Then for each group separation, the mean trace for each group of power traces is calculated as shown in steps 6 and 7 in Algorithm 5.9. For example,  $\text{MeanTrace}2_{3a}$  is the mean traces of all the power traces that satisfy  $C[1] = 3a$ .

Then for each guess of  $sk_{10}[0] \oplus sk_{10}[1]$  as  $\Delta G$ , for each sample point, the correlation coefficient between  $\text{MeanTrace}1_i$  and  $\text{MeanTrace}2_{i+\Delta G}$  for  $i = \{0, 1, \dots, 255\}$  is calculated. The attackers expect that when  $\Delta G$  is equal to the real value,  $\Delta G = sk_{10}[0] \oplus sk_{10}[1]$ , the calculated correlation will have a peak in the attack results. Recall that  $\text{MeanTrace}1_i$  is the mean

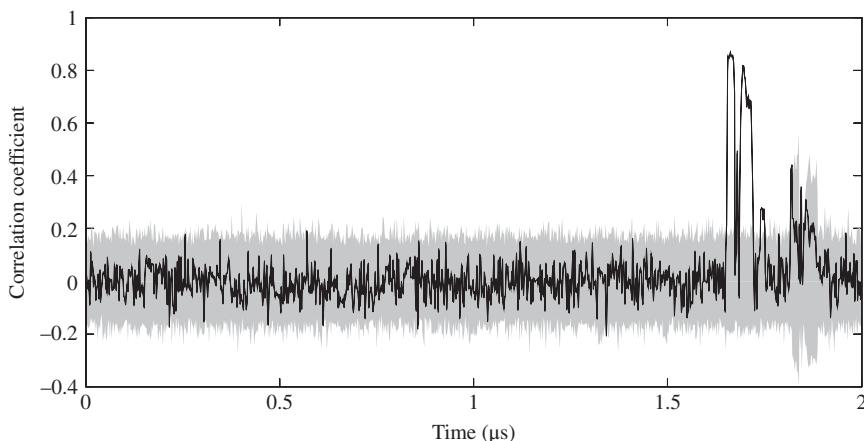
traces of all the power traces that satisfy  $C[0] = i$ ,  $\text{MeanTrace2}_{i \oplus \Delta G}$  is the mean trace that satisfy  $C[1] = i \oplus \Delta G$ . When  $\Delta G = sk_{10}[0] \oplus sk_{10}[1]$ ,  $i \oplus (i \oplus \Delta G) = sk_{10}[0] \oplus sk_{10}[1]$ . Thus, for all  $i = 0, 1, \dots, 255$ , the  $\text{MeanTrace1}_i$  and  $\text{MeanTrace2}_{i \oplus \Delta G}$  satisfy  $C[0] \oplus C[1] = sk_{10}[0] \oplus sk_{10}[1]$ . So for each of the 256 pairs of  $\text{MeanTrace1}_i$  and  $\text{MeanTrace2}_{i \oplus \Delta G}$ , the similar side-channel leakage from S-box 0 and S-box 1 is expected. When  $\Delta G$  is correct, high correlation can be expected as it is the correlation of power consumptions for 256 pairs of S-boxes with the same data input. When  $\Delta G$  is not correct, low correlation can be expected as it is the correlation of power consumptions for 256 pairs of S-boxes with different data inputs. Therefore, similar to the previous DPA and CPA attacks, it is expected that a collision-based EF can be used to identify the correct key byte difference here.

The attack result by applying Algorithm 5.9 is demonstrated on AES-pprm1 and AES-comp. The attack result on AES-pprm1 is shown in Figures 5.61 and 5.62 for AES-pprm1. The attack result on AES-comp is shown in Figures 5.63 and 5.64 for AES-comp.

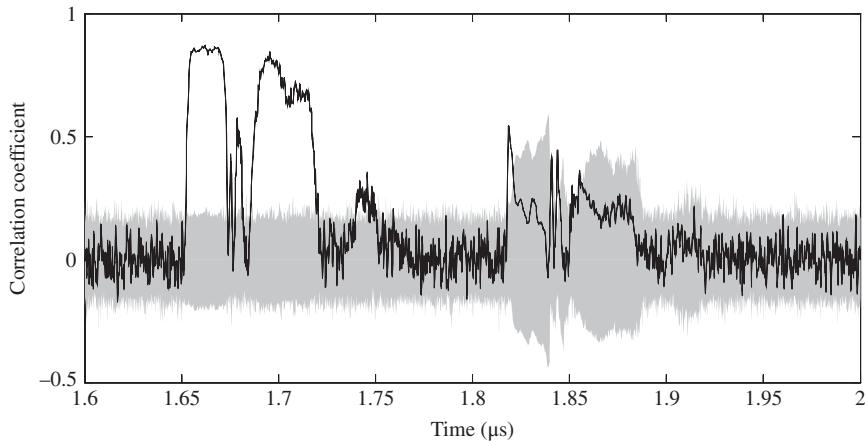
The attack results show that correlation-enhanced power analysis collision attack works well for both AES-pprm1 and AES-comp. The correct key byte difference can be clearly identified. Note that as only the value of the S-box input is used in the attack, the successful attack results imply that the side-channel leakage that depends on the last AES round input exists for both AES-comp and AES-pprm1.

In the previous attack on AES-comp, the HW-model-based DPA attack does not show any good attack result. The reason is that the HW model is not accurate to describe the leakage rather than there is no related side-channel information leakage related to the S-box input. The correlation-enhanced power analysis collision attack is a very powerful attack method as it enables an efficient secret key recovery without using a LM.

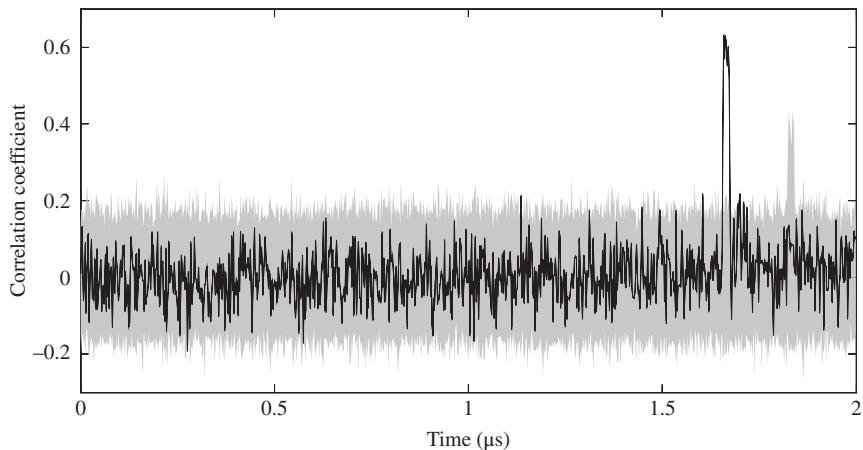
Note that for the collision-based analysis have two aspects, only the difference between key bytes can be identified instead of direct recovery of the key values. By repeating recovering the difference between key bytes, the key space of AES can be restricted to the size of  $2^8$ . An exhaustive search based on a pair of plaintext and ciphertext can be used to identify the



**Figure 5.61** Correlation-enhanced power analysis collision attack result targeting  $sk_{10}[0] \oplus sk_{10}[1]$  for AES-pprm1



**Figure 5.62** Zoomed Figure 5.61 in last two clock cycles

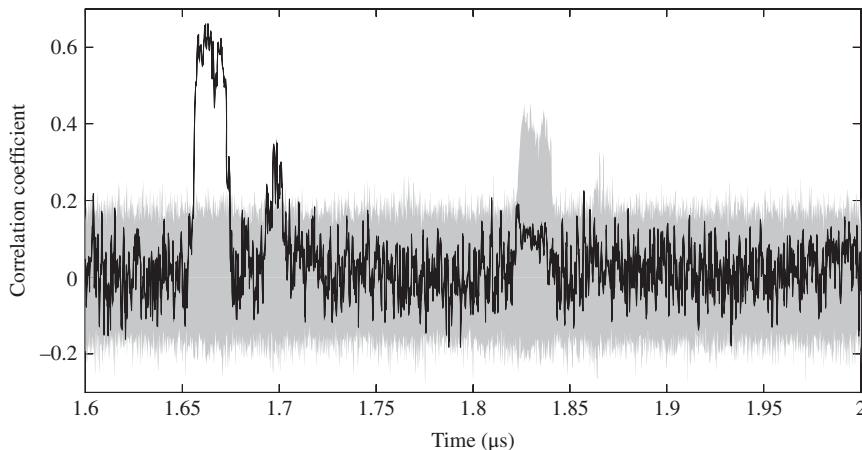


**Figure 5.63** Correlation-enhanced power analysis collision attack result targeting  $sk_{10}[0] \oplus sk_{10}[1]$  for AES-comp

right key. In addition, the success of this attack requires at least a certain amount of power traces to ensure that the collision exists. When the attackers cannot control the plaintext, it is difficult to achieve any secret key recovery with only a few power traces using collision-based power analysis. However, for an attacker who has a very accurate LM of the device, a few power traces are enough for the key recovery.

## 5.4 Basics of Fault Analysis

For the active attacks, the attackers need to interact with the target device to perform the key recovery. In this chapter, the active attacks using fault analysis are explained, in which



**Figure 5.64** Zoomed Figure 5.63 in last two clock cycles

the attackers disturb the cryptographic calculation and inject a fault to extract the key value. Without any attempt of a fault injection, the target device performs an ordinary cryptographic algorithm and outputs a correct calculation result. In fault analysis, the device is forced to operate an abnormal calculation.

The attackers may be able to guess the effects on the calculation caused by the fault injection, for example, what kind of change is applied to the intermediate value when the fault injection is performed. In addition, the attackers may be able to observe the behavior of the device under the fault injection, for example, to obtain the value of the faulty calculation result. By combining the information of the fault injection and the information of the abnormal behavior, the key recovery attack is performed. More precisely, if the attackers' guess of the fault injection and the observed calculation results are reasonable, the secret key could be recovered.

Note that all of the fault injections cannot be used for recovering the secret key. For instance, it is significantly difficult to perform fault analysis on AES-128 when random faults are injected over several round operations or when 1-byte fault is injected at the beginning of the fifth round of AES-128. Therefore, the attackers must consider the fault injection method to have a preferable fault in terms of fault analysis.

For example, the FA using a laser beam, which is regarded as the semi-invasive attack, is possible as it can alter the intermediate value during the cryptographic calculation. However, the effective position for the laser shot requires the knowledge about the layout of a cryptographic implementation in the target device. In addition, the laser-based fault injection has a high probability of causing permanent damage to the chip and requires relatively high-cost equipment.

In comparison, noninvasive fault injection methods are considered by providing an irregular power supply or an irregular clock signal to the pins of the IC chip that are normally exposed to the attackers. If such irregular signals affect the cryptographic calculation, the setup-time violation is likely to occur in the device. These fault injection methods are relatively simple and cheap to perform, and a high timing accuracy of the fault injection can be achieved.

Therefore, in this section, several noninvasive fault injection methods based on the setup-timing violations are explained mainly. Semi-invasive fault injections using laser shots are briefly introduced later.

### 5.4.1 Faults Caused by Setup-Time Violations

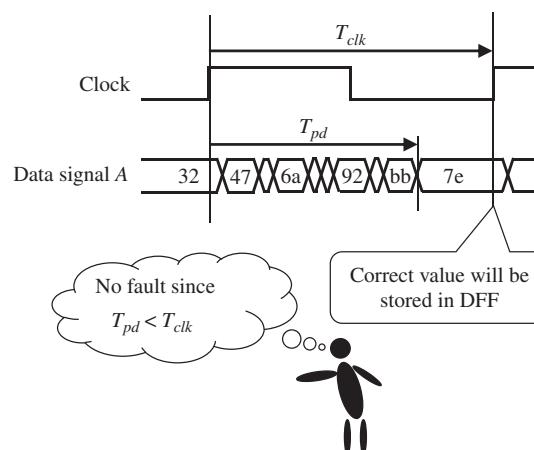
When faults based on the setup-time violations occur, an unfinished calculation result by a combinatorial logic is stored in the DFFs. At the beginning of a clock cycle, the combinatorial logic takes a new input value for the evaluation. As explained in Section 2.5, the calculation of the current clock cycle needs a certain amount of time to finish the evaluation. The time required for all the input signals to be evaluated in the combinatorial logic is called path delay, denoted by  $T_{pd}$ . When the period of a clock cycle, denoted by  $T_{clk}$ , is shorter than the path delay, a setup-time violation occurs.

For a data signal A, when its path delay is shorter than the period of the clock cycle, that is,  $T_{pd} < T_{clk}$ , there is enough time for the calculation to finish. Thus, the correct calculation result will be stored in the DFFs. An illustration of this case is shown in Figure 5.65.

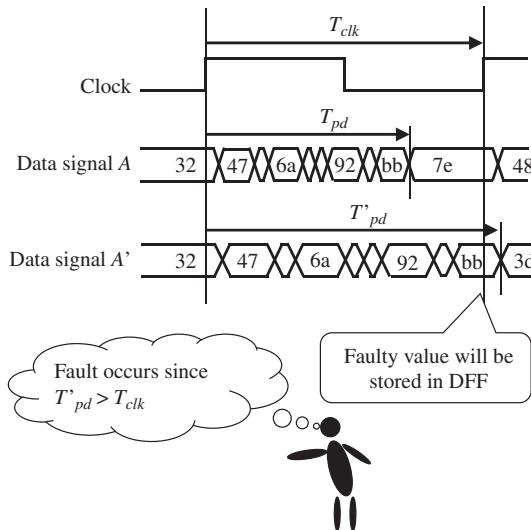
For the case that the fault injection causes a setup-time violation, the path delay is longer than the clock period, that is,  $T_{pd} > T_{clk}$ . In this case, the calculation is not finished so that a faulty calculation result will be stored in the DFFs.

There are mainly two approaches to achieve the setup-time violation. One approach is by increasing the delay timing of the calculation as shown in Figure 5.66. The data signal  $A'$  has a longer path delay than that of the data signal A owing to some fault injection. In this case, a faulty calculation result is stored in the DFFs for the data signal  $A'$  as  $T'_{pd} > T'_{clk}$ . In practice, this type of fault injections can be achieved by decreasing the voltage of the power supply or by increasing the environment temperature for the target device.

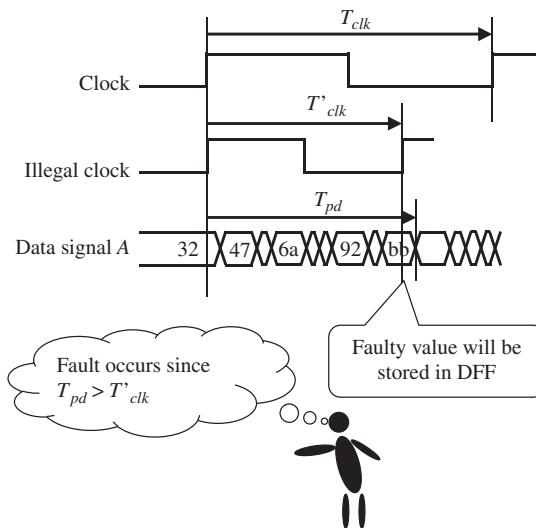
The other approach to achieve the setup-time violation is by decreasing the clock period as shown in Figure 5.67. The **illegal clock** signal has a shorter period,  $T'_{clk}$ , than that of a normal



**Figure 5.65** Signal transitions without setup-time violation



**Figure 5.66** Setup-time violation by increasing path delay



**Figure 5.67** Setup-time violation by decreasing the clock period

clock. If the clock period is short enough such that  $T'_{clk} < T_{pd}$ , a faulty calculation result for the data signal  $A$  will be stored in the DFFs. In practice, this type of fault injections can be achieved using an illegal clock signal with an irregular clock period as the clock supply to the target device.

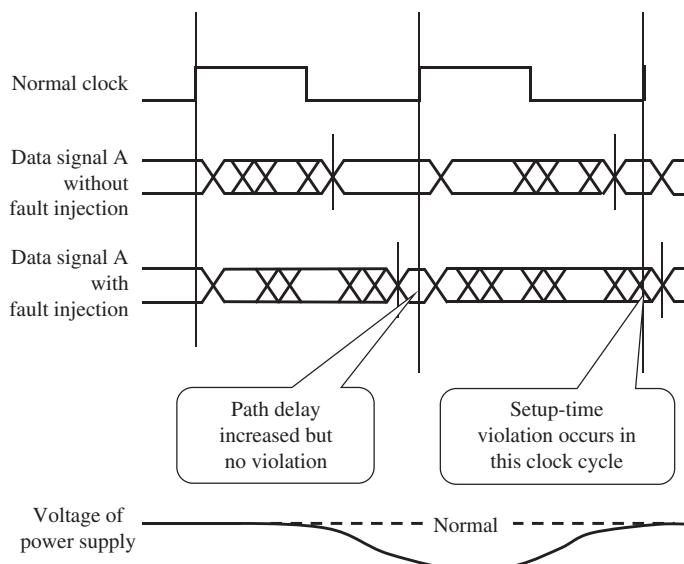
### 5.4.1.1 Fault Injection Using Low Supply Voltage and Its Low Timing Accuracy

The basic mechanism of the fault injection by supplying a low voltage (under power) is shown in Figure 5.68. When the voltage of the power supply is lowered, it takes more time to charge the signal values than usual owing to the parasitic capacitance in the circuit, and the path delay of the calculation becomes longer than the normal case. If the path delay becomes longer than the period of the clock signal, a setup-time violation occurs and an incorrect calculation result is stored in the DFFs. In the circuit with a loop architecture, the faulty calculation result will be propagated to the following calculations and the final calculation result also becomes a faulty one.

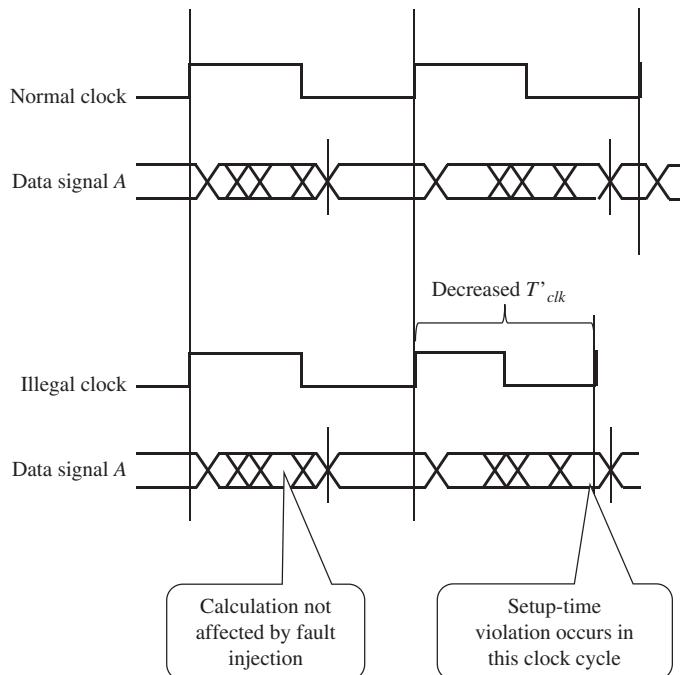
The accuracy of the fault injection timing is related to the duration of the voltage alteration. In practice, the timing control of the voltage alteration is difficult to achieve an under-power fault injection at a specific clock cycle. Normally, a few clock cycles will be affected by the voltage alteration. As shown in Figure 5.68, the path delay of the data signals is affected by a low voltage during two clock cycles. The setup-time violation occurs only in the second clock cycle. In the first cycle, the setup-time violation does not occur. Compared to the under-power fault injection, fault injections using illegal clock can achieve a higher timing accuracy, which is explained in the next section.

### 5.4.1.2 Fault Injection Using Illegal Clock Signal and Its High Timing Accuracy

The basic mechanism of the fault injection using the illegal clock signal is shown in Figure 5.69. The attackers prepare an illegal clock signal that has a shorter clock period than



**Figure 5.68** Setup-time violation based on under-power fault injection



**Figure 5.69** Setup-time violation based on illegal clock supply

the normal case, for example, the second clock cycle in Figure 5.69. When the decreased clock period is shorter than the path delay of a calculation, a setup-time violation occurs.

For the illegal clock signal, the attackers can specify the **illegal clock cycle** so that the fault injection can achieve a high timing accuracy. For example, in Figure 5.69, the second clock cycle is the illegal clock cycle. Thus, the fault only occurs in the second clock cycle. The calculation in the first clock cycle is not affected at all.

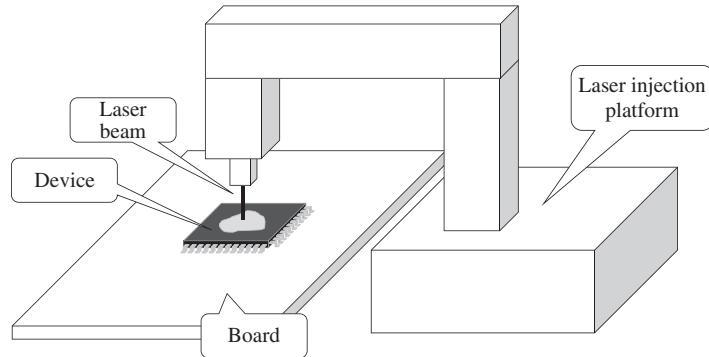
### 5.4.2 Faults Caused by Data Alteration

Another major type of fault injection is based on laser injection. The equipment for this type of fault injection is usually expensive. However, the performed fault injection can be very accurate and even flip a bit value stored in a DFF or SRAM. The illustration of a laser fault injection platform is shown in Figure 5.70. In the laser fault injection, depackaging the chip is required. Moreover, the laser injection platform needs to manage a high accuracy of adjusting the position of laser injection as well as the timing and the strength of the laser beam.

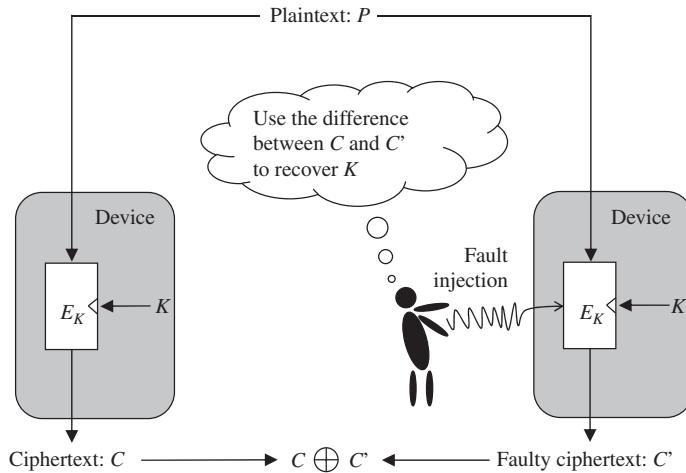
## 5.5 Fault Analysis on Block Ciphers

### 5.5.1 Differential Fault Analysis

Among different types of fault analysis for AES, the most discussed one is **differential fault analysis (DFA)**. In this section, the basic concept of DFA is explained. Further discussions about the optimization for the DFA attack on AES-128 will be explained in Chapter 6.



**Figure 5.70** Example of laser fault injection platform



**Figure 5.71** Overview of differential fault analysis

In DFA, the attackers investigate the difference in values generated from the fault-free calculation and the faulty calculation to recover the secret information. The setup for the DFA attack on the encryption of a block cipher is explained as follows. The attackers perform encryption of the same plaintext,  $P$ , at least twice. As shown in Figure 5.71, the first execution of the encryption is without any faults so that a fault-free ciphertext,  $C$ , is obtained. Then, the encryption of the same plaintext,  $P$ , is repeated, during which the fault injection is performed by the attackers. After a successful fault injection occurs, a faulty ciphertext,  $C'$ , is obtained. The DFA attacks recover the key using the difference of  $C$  and  $C'$ , that is,  $\Delta C$ , and the knowledge about the injected fault.

### 5.5.1.1 Fault Model

**Fault model** is an important concept for DFA. The fault model describes the consequence of a fault injection in the calculation of a cryptographic algorithm. For example, in a fault model

for AES-128, it can be assumed that the attackers alter the value of 1 bit of the intermediate value at certain timing. If the position of 1-bit fault is unknown, the attackers need to assume 128 possible faults.

With different fault models, the efficiency in the key recovery of DFA is largely varied. Under a specific fault model, the key recovery algorithm is highly related to the cryptanalysis introduced in Chapter 4. The reasonable fault models are the ones that can be achieved with a fault injection setup for real devices. In this regards, 1-byte fault is more reasonable compared to 1-bit fault in the previous example if the device performs on byte-oriented operations. Therefore, this section focuses on the fault model that the attackers alter the value of a byte of the intermediate value into a random faulty value. This fault model is called **random byte-fault model**. In this section, it is further assumed that the attackers can control the position of the altered byte. Chapter 6 will discuss the case that the position of the altered byte is unknown.

For the random byte-fault model in the fixed byte position, the possible difference between the fault-free intermediate value and the faulty intermediate value is restricted to a set of size 255. The only 1 active byte has 255 possible values in the difference, whereas the other inactive bytes have no difference.

### 5.5.1.2 Key Recovery Concept of DFA on AES-128

The key recovery concept for DFA on AES-128 can be understood as follows. Under the random byte-fault model in the fixed byte position, the attackers know that there is 1 active byte at the position where the fault is injected. In addition, the attackers obtain the values of the fault-free and the faulty ciphertexts.

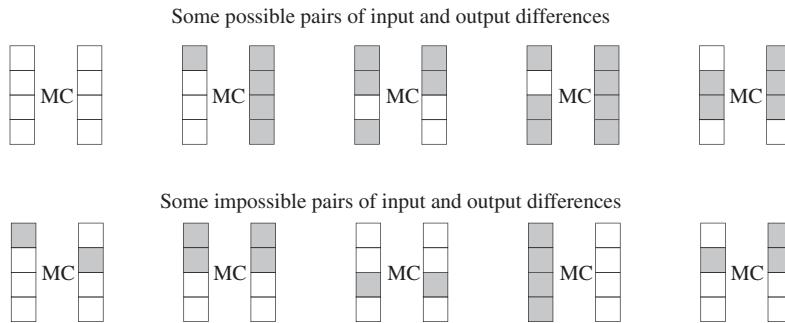
The attackers first prepare an expected propagation of the active bytes under the AES-128 specification. Then, the attackers guess subkey values in the last several rounds. For each key candidate, the attackers use the partial decryption of AES-128 with the fault-free and the faulty ciphertexts to calculate the origin of the active byte. In detail, using a key candidate and the fault-free ciphertext, the fault-free intermediate values can be calculated. Moreover, using the key candidate and the faulty ciphertext, the faulty intermediate values are calculated as well. Then the differences between the fault-free and the faulty intermediate values are calculated.

The calculated propagation of the active bytes using a key candidate is checked whether or not 1 active byte is located at an intended position specified by the fault model. If the calculated active byte position does not match the fault model, the corresponding key candidate can be discarded from the key space. After exhaustively checking all the possible key candidates, the key space can be narrowed down. In practice, this key recovery algorithm is optimized with divide-and-conquer algorithm to make the computational cost practical.

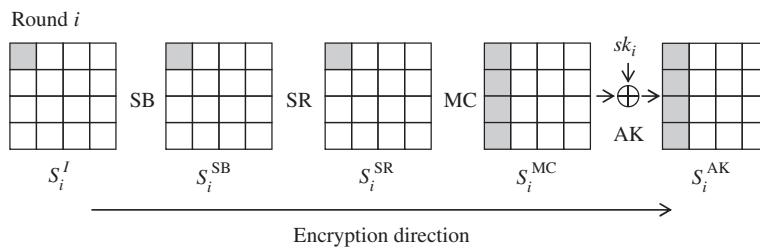
### 5.5.1.3 Propagation of Active Bytes in AES-128

The active bytes propagate in accordance with the specification of AES-128, which has been explained in Chapter 1. Here, the propagation of the active bytes by one AES round, which consists of SubBytes, ShiftRows, MixColumns, and AddRoundKey, is briefly reviewed.

For the SubBytes operation, the input and output states have the same number of active bytes at the same positions. Recall that AES S-box is bijective, which infers that two different S-box inputs never have the same output value. Thus, an active byte remains active after the SubBytes operation.



**Figure 5.72** Propagation patterns for each column of MixColumns calculation



**Figure 5.73** Propagation of active bytes in one-round operation (encryption direction)

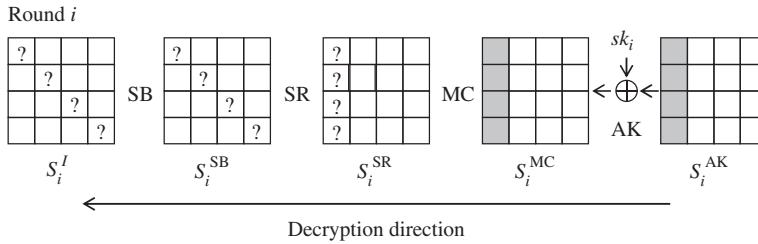
For the ShiftRows operation, the input and output states have the same number of active bytes as only the positions of the active bytes are changed.

For the MixColumns operation, first recall that MixColumns is a column-wise operation. The input and output columns related to the MixColumns operation have 8 bytes in total. An important feature is that these 8 bytes either have 0 active bytes or have 5 or more active bytes due to the property of MDS introduced in Chapter 4. Following this rule, one can see that if the input column of MixColumns has 1 active byte, the output column must have 4 active bytes. Figure 5.72 shows some possible and impossible propagation patterns of the active bytes for the input and output columns of the MixColumns operation.

For the AddRoundKey operation, the active bytes keep their values of difference and positions as AddRoundKey only performs XOR with a subkey.

With the above-mentioned explanations, Figure 5.73 shows the propagation of the active bytes via 1 AES round starting with 1 active byte at  $S_i^I$ . The 1 active byte at the AES round input remains active after the SubBytes operation, changes its position after the ShiftRows operation, and propagates to 4 active bytes in the MixColumns operation. After the AddRoundKey operation, the AES round output has 4 active bytes in a column. The propagation of the active bytes follows Figure 5.73 regardless of the value of AES states and the secret key.

Let us see the propagation of 4 active bytes following the decryption direction of AES as shown in Figure 5.74. Suppose that 4 active bytes are located at one column in the output of the AES round, that is,  $S_i^{AK}$ . The active bytes at the input for the AddRoundKey operation is the same as ones at  $S_i^{AK}$ . The active bytes at the input for the AddRoundKey operation,  $\Delta S_i^{SR}$ , are



**Figure 5.74** Propagation of active bytes in one-round operation (decryption direction)

uncertain owing to the property of MDS. According to the propagation rule of MixColumns, the first column of  $S_i^{\text{SR}}$  could have 1, 2, 3, or 4 active bytes. Hence, the difference at  $S_i^I$  could have 1, 2, 3, or 4 active bytes as well. The number of active bytes and their positions at  $S_i^I$  requires the value of the key to be determined.

The DFA attack against first round of AES can be considered as follows. Suppose that the fault is injected to a fixed byte at the beginning of the AES round,  $S_i^I$ , according to the random byte-fault model. As shown in Figure 5.73, the number of active bytes and their positions are fixed at the output of the AES round. The fault-free value and the faulty value at  $S_i^{\text{AK}}$  are denoted by  $C$  and  $C'$ , respectively. For a key candidate  $sk_i$ , the difference at  $S_i^I$  can be calculated by

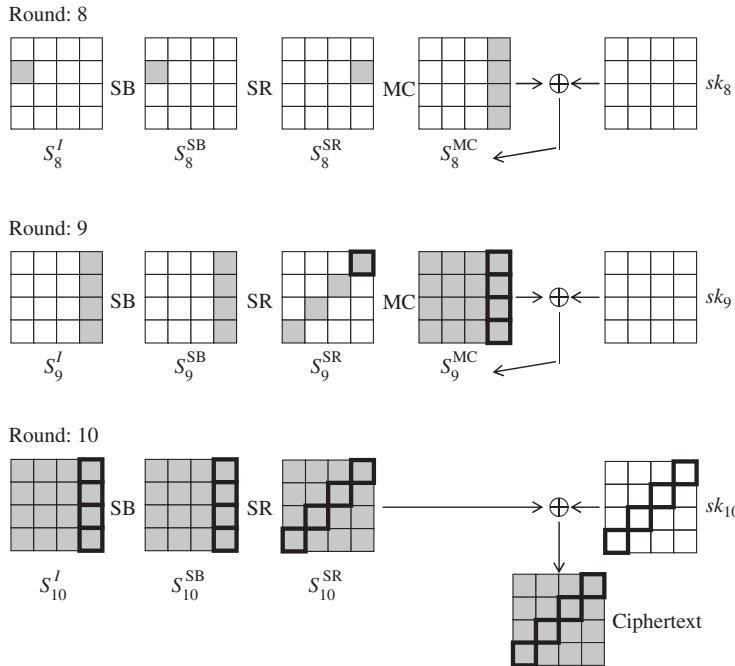
$$\Delta S_i^I = \text{SB}^{-1}(\text{SR}^{-1}(\text{MC}^{-1}(C \oplus sk_i))) \oplus \text{SB}^{-1}(\text{SR}^{-1}(\text{MC}^{-1}(C' \oplus sk_i))). \quad (5.3)$$

If the result of  $\Delta S_i^I$  does not have a difference at  $S_i^I$  as shown in Figure 5.73, the current key candidate cannot be the correct key, and therefore it can be removed from the key space. Actually, the attackers only need to check the difference at  $S_i^{\text{SR}}$  as the propagation from  $S_i^{\text{SR}}$  to  $S_i^I$  is fixed in the decryption direction. Note that the MDS property of the MixColumns operation is the point for the key recovery of DFA on AES-128.

For the DFA attack on first round of AES explained above, only 4 bytes of  $sk_i$  are related to the active bytes at  $S_i^{\text{AK}}$ . Before the DFA attack, there are  $2^{32}$  possible key candidates. The size of the remaining key candidates after the DFA attack can be roughly estimated. In Figure 5.73, the number of the possible difference at  $S_i^{\text{SR}}$  is  $255 \approx 2^8$ . In contrast, the number of the possible values of the difference at  $S_i^{\text{MC}}$  is  $255^4 \approx 2^{32}$ . Thus, for a random mapping from the difference at  $S_i^{\text{MC}}$  to the difference at  $S_i^{\text{SR}}$ , the probability of a match is estimated as  $2^8/2^{32} = 2^{-24}$ . Therefore, about  $2^{32} \cdot 2^{-24} = 2^8$ , key candidates will remain after the DFA attack on one round of AES with one pair of fault-free and the faulty ciphertexts. The above-mentioned estimation matches the average of the experimental results.

#### 5.5.1.4 DFA Attack on AES-128 with a Byte Fault at $S_8^I$

Let us consider the case where the fault is injected to a fixed byte at the beginning of the eighth AES round accordingly to the random byte-fault model. This fault model has been discussed in many literatures, such as Piret and Quisquater (2003), Tunstall *et al.* (2011) and Mukhopadhyay (2009). Following the encryption direction of the AES algorithm, the propagation of the active bytes is shown in Figure 5.75. The 1 active byte at  $S_8^I$  will propagate to 4 active bytes



**Figure 5.75** Propagation of active bytes for AES-128

at state  $S_9^I$  as a column. Then these 4 active bytes are distributed to 4 columns at  $S_9^{SR}$ . Each active byte at  $S_9^{SR}$  is propagated to 4 active bytes at the ciphertext. For example, the active bytes marked by bold lines correspond to the fourth column of the MixColumns operation of the ninth AES round, which are used as a group in the key recovery.

The key recovery algorithm for DFA on AES-128 is illustrated in Algorithm 5.10, which can be divided into two parts.

For the first part, that is, steps 1 to 10 in Algorithm 5.10, the key recovery considers only the last two rounds of AES, that is, ninth and tenth rounds. The key recovery is performed separately for each column of MixColumns of the ninth round. With a key candidate of 4 bytes of  $sk_{10}$  that are related to 1 column of the MixColumns of the ninth round, the difference of this column at  $S_9^{SR}$  can be calculated using  $C$  and  $C'$ . The calculated difference is compared with the expected pattern of the active byte. For each column of MixColumns of the ninth round, the size of the key space for the 4 bytes of  $sk_{10}$  is expected to be  $2^8$  as there are  $2^{32}$  key candidates and the probability of a match is about  $2^{-24}$ . After the first part of the attack, the key space of  $sk_{10}$  will be restricted to about  $2^{8 \times 4}$  or  $2^{32}$ . The computational cost of the first part is about four times of exhaustive verifications of  $2^{32}$  key candidates, which can be computed by a standard personal computer.

The second part of the DFA attack on AES-128, that is, steps 11 to 18 in Algorithm 5.10, considers the last three rounds of AES. After the first part, the size of the remaining key space of  $sk_{10}$  is  $2^{32}$ . For each key candidate of  $sk_{10}$ , one can calculate the difference at  $S_8^{SR}$  using  $C$  and  $C'$ . At the MixColumns of the eighth round, the probability of a match is about  $2^{-24}$ .

**Algorithm 5.10** Key Recovery Algorithm for DFA on AES-128**Input:** Fault-free ciphertext state  $C$ , faulty ciphertext state  $C'$ **Output:** Restricted key space,  $\kappa$ 

```

1: for  $i = 1$  to  $4$  do
2:   Locate the related key bytes of  $sk_{10}$  for column  $i$  of  $S_9^{\text{SR}}$ ;
3:   for each  $G = 0$  to  $2^{32} - 1$  of related key bytes do
4:     Calculate the difference at column  $i$  of  $S_9^{\text{SR}}$ ;
5:     if column  $i$  of  $S_9^{\text{SR}}$  has 1 active byte at intended position then
6:       Add  $G$  to  $\kappa_i$ ;
7:     end if
8:   end for
9: end for
10:  $\kappa \leftarrow$  combinations of elements in  $\kappa_1, \kappa_2, \kappa_3$ , and  $\kappa_4$ ;
11: for each  $G$  in  $\kappa$  do
12:   Calculate the difference at  $S_8^{\text{SR}}$ ;
13:   if difference of  $S_8^{\text{SR}}$  has 1 active byte at intended position then
14:     Keep  $G$  in  $\kappa$ ;
15:   else
16:     Discard  $G$  from  $\kappa$ ;
17:   end if
18: end for
19: return  $\kappa$ ;

```

Thus, after the second part of the DFA attack, the size of the key space is narrowed down to around  $2^{32} \cdot 2^{-24} = 2^8$ .

Finally, the attackers need some additional information to identify the key. One way to identify the key is to use a plaintext–ciphertext pair to verify all the remaining key candidates. The other way is to perform another fault injection and run Algorithm 5.10 again and prepare another set for the key candidates whose size is  $2^8$ . The correct key can be identified with high probability by checking the intersection of those key candidate sets.

**Exercise 5.9** *If the attackers can only inject random-byte fault in the fixed byte position at the ninth round of AES-128, how many fault injections are required to recover the key?*

**Exercise 5.10** *If the attackers can only inject random-byte fault in the fixed byte position at the tenth round of AES-128, is the key recovery in DFA still possible? Explain the reason of your answer.*

**Exercise 5.11** Which of the following fault model is better in the key recovery with DFA, the random-byte fault model in the fixed byte position or the bit-fault model in the fixed bit position? In the bit-fault model in the fixed bit position, the attackers can flip the value of a specific bit of an intermediate value.

### 5.5.2 Fault Sensitivity Analysis †

This section discusses a relatively new type of fault analysis called **fault sensitivity analysis (FSA)**. For FSA, the attackers use fault injections to obtain the information leakage called **fault sensitivity (FS)** from the device. FS describes how strong the intensity of a fault injection is required to successfully inject a fault in the target device. In other words, FS describes how much the current calculation is sensitive to the attempt of a fault injection. With the FS data, the key recovery algorithm used for side-channel analysis can be directly applied to achieve the key recovery in FSA.

A special feature of FSA compared to DFA is that the value of the faulty calculation result is not required to recover the key. This feature is useful to attack implementations with the countermeasures against the DFA attack, which ensures that the attackers cannot obtain the value of the faulty calculation result. This type of countermeasure is not effective against the FSA attack.

#### 5.5.2.1 Fault Injection Intensity

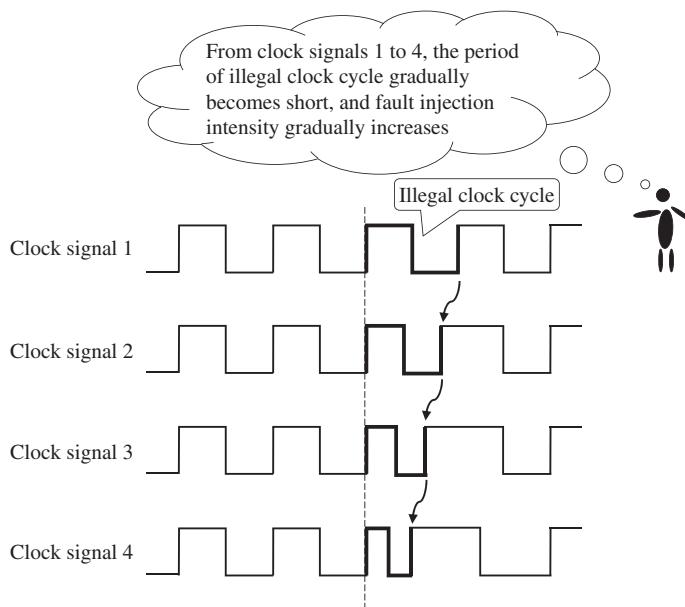
To understand the FS, it is also required to understand the concept of **fault injection intensity**. In the default working environment of a device, no fault is expected for the calculation. When the attackers attempt to perform a fault injection, the working environment is changed. Fault injection intensity describes how much the environment is changed in the attempt of injecting a fault.

For a laser-based fault injection, the stronger the power level of the laser shot is, the higher the fault injection intensity is. For an under-power-based fault injection, the lower the voltage of the power supply is, the higher the fault injection intensity is. For an illegal clock-based fault injection, the shorter the period of the illegal clock cycle is, the higher the fault injection intensity is. In general, the larger the potential of a faulty calculation is, the higher the fault injection intensity is.

Figure 5.76 shows an example of the fault injection intensity for the illegal clock signal. There are four clock signals listed in Figure 5.76, from clock signals 1 to 4, the clock period of the illegal clock cycle decreases gradually. Thus, the fault injection intensity gradually increases.

#### 5.5.2.2 Critical Fault Injection Intensity as Fault Sensitivity

In practice, FS is measured as the **critical fault injection intensity**. FSA assumes that the same calculation, for example, the encryption of the same plaintext, is repeated under the fault



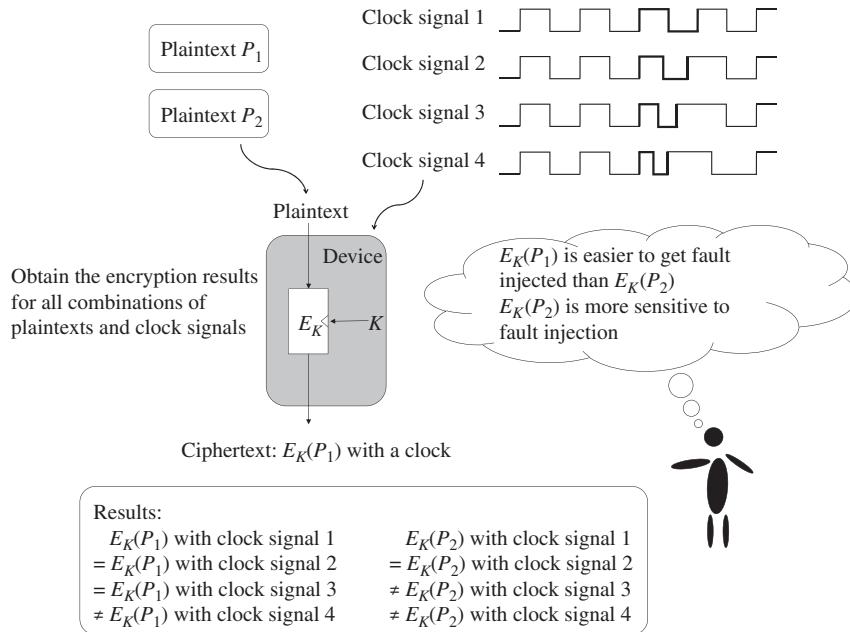
**Figure 5.76** Fault injection intensity for illegal clock signal

injections with controlled fault injection intensities. It is possible for the attackers to understand the threshold of the fault injection intensity between the device's normal and abnormal behaviors, which is the critical fault injection intensity or FS. When the fault injection intensity is lower than the FS, the device performs the correct calculation and outputs the fault-free calculation result. When the fault injection intensity is higher than the FS, the device performs a faulty calculation and outputs a faulty calculation result.

In Figure 5.77, the measurement of FS using illegal clock signals is illustrated. For plaintext  $P_1$ , the encryption is repeated four times with clock signals 1 to 4. The encryption results using clock signals 1 to 3 are the same with each other, and the encryption result using clock signal 4 is a different value that is a faulty value. One can see that for plaintext  $P_1$ , the clock signal 4 is the threshold to trigger the faulty ciphertext output. For plaintext  $P_2$ , the same process is performed and the clock signal 3 is the threshold to trigger the faulty ciphertext output.

As shown in Figure 5.77, by varying the fault injection intensity and observing the calculation results, the FS can be measured. In practice, for different data, the measured fault sensitivities are expected to be different. In the example shown in Figure 5.77, the calculation using  $P_2$  is more sensitive to the fault injections compared to that of  $P_1$ . The difference of FS when processing different data is the point of the key recovery in FSA.

When illegal clock signal is used in the FSA attack, the setup-time violations occur in the device. Thus, the measured FS data is related to the path delay of a calculation in the clock cycle. For the setup-time violation, whether a fault occurs depends on the relations between the path delay  $T_{pd}$  and the clock period  $T_{clk}$ . The attackers control  $T_{clk}$  and observe the calculation results to measure the FS that is related to  $T_{pd}$ . As  $T_{pd}$  is related to the processed data, the measured FS is related to the processed data as well.



**Figure 5.77** Fault sensitivity measured as critical fault injection intensity

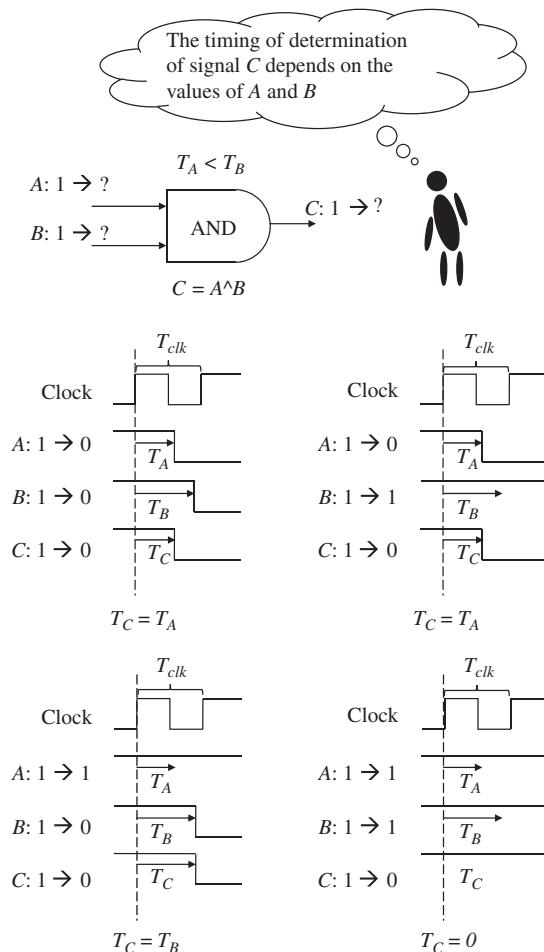
### 5.5.2.3 Data Dependency of Signal Delays

A simple example is given to show that the path delays of signals depend on the value of the processed data. Consider a two-input AND gate with input signals  $A$  and  $B$  and the output signal  $C = A \wedge B$ . Assume that the signal  $A$  always arrives earlier than the signal  $B$  to the AND gate, that is,  $T_A < T_B$ .

Assume that the initial values of  $A$  and  $B$  are both 1 and they are transmitted into the next values in a certain clock cycle. The path delay of  $C$ ,  $T_C$ , is discussed for four possible cases of the next values of  $A$  and  $B$  as shown in Figure 5.78.

- When the next values of  $A$  and  $B$  are 0 and 0, the next value of  $C$  is determined when signal  $A$  arrives at the AND gate as a zero input already decides the output of the AND gate. Thus,  $T_C$  is determined only with  $T_A$ , that is,  $T_C = T_A$ .
- When the next values of  $A$  and  $B$  are 0 and 1, similarly  $T_C$  is determined only with  $T_A$ , that is,  $T_C = T_A$ .
- When the next values of  $A$  and  $B$  are 1 and 0, the next value of  $C$  is determined when signal  $B$  arrives at the AND gate. Thus,  $T_C$  is determined with  $T_B$ , that is,  $T_C = T_B$ .
- When the next values of  $A$  and  $B$  are 1 and 1, the next value of  $C$  is not changed from the initial value. Thus,  $T_C = 0$ .

From the above-mentioned discussion, it is confirmed that the path delay of the AND gate largely depends on the values of the input signals. Note that the delay of the AND gate itself is ignored in the above-mentioned discussion for simplicity.



**Figure 5.78** Example of data dependency of path delays

**Exercise 5.12** For all the other 12 possible combinations for  $A$  and  $B$  in Figure 5.78, calculate the signal transition and the path delay of  $C$  and complete Table 5.2.

**Exercise 5.13** Following the discussion of Figure 5.78 and Table 5.2, calculate the signal transition and the path delay of the output for a two-input OR gate and a two-input XOR gate with different input data.

**Table 5.2** Signal transitions and path delay for AND gate

A	B	C	$T_C$
$0 \rightarrow 0$	$0 \rightarrow 0$	$0 \rightarrow$	
$0 \rightarrow 0$	$0 \rightarrow 1$	$0 \rightarrow$	
$0 \rightarrow 1$	$0 \rightarrow 0$	$0 \rightarrow$	
$0 \rightarrow 1$	$0 \rightarrow 1$	$0 \rightarrow$	
$0 \rightarrow 0$	$1 \rightarrow 0$	$0 \rightarrow$	
$0 \rightarrow 0$	$1 \rightarrow 1$	$0 \rightarrow$	
$0 \rightarrow 1$	$1 \rightarrow 0$	$0 \rightarrow$	
$0 \rightarrow 1$	$1 \rightarrow 1$	$0 \rightarrow$	
$1 \rightarrow 0$	$0 \rightarrow 0$	$0 \rightarrow$	
$1 \rightarrow 0$	$0 \rightarrow 1$	$0 \rightarrow$	
$1 \rightarrow 1$	$0 \rightarrow 0$	$0 \rightarrow$	
$1 \rightarrow 1$	$0 \rightarrow 1$	$0 \rightarrow$	
$1 \rightarrow 0$	$1 \rightarrow 0$	$1 \rightarrow 0$	$T_A$
$1 \rightarrow 0$	$1 \rightarrow 1$	$1 \rightarrow 0$	$T_A$
$1 \rightarrow 1$	$1 \rightarrow 0$	$1 \rightarrow 0$	$T_B$
$1 \rightarrow 1$	$1 \rightarrow 1$	$1 \rightarrow 1$	0

#### 5.5.2.4 Collection of Fault Sensitivity Data

The first task for the attackers to perform the FSA attack is to collect the FS data. The method is written in Algorithm 5.11, where  $F$  denotes the fault injection intensity. For each plaintext as the input, the encryption requires to be performed several times. For the first encryption as shown in step 4 in Algorithm 5.11, no fault injection is performed, that is,  $F = 0$ . Then in steps 6 and 7, the encryption is repeated several times until the encryption has a fault successfully injected. For each repetition of the encryption, the fault injection intensity  $F$  is increased by a certain amount. For example, one can shorten the period of the illegal clock cycle by 5%. The fault injection intensity that corresponds to the successful fault injection is stored as the FS as shown in step 9 in Algorithm 5.11.

When AES-128 is the target of FSA, the attackers can perform the fault injection at the last round of AES so that FS of the last round AES calculation can be measured. Note that the measured FS data is not in the form of a trace. The FS data can be measured for each S-box in the last AES round independently by observing the correctness of each byte of ciphertext independently. Therefore, each  $\text{FS}_i$ ,  $i = 1, 2, \dots, N$ , in Algorithm 5.11 can be extended to a set of 16 FS data corresponding to 16 S-boxes, that is,  $\text{FS}_i = \{\text{FS}_i[0], \text{FS}_i[1], \dots, \text{FS}_i[15]\}$ .

#### 5.5.2.5 FSA Key Recovery Algorithm Using LMs

After the collection of FS data, the attackers use the key recovery algorithm to identify the secret key. Similar to the side-channel information such as the power consumption, the FS data also depends on the processed intermediate value during the cryptographic algorithm. Thus, the key recovery techniques used in side-channel analysis can be applied to the FSA attack. This section explains the FSA key recovery algorithm based on the LMs. One can review Section 5.5.3 to recall the general DPA key recovery algorithm.

**Algorithm 5.11** Collection of Fault Sensitivity Information

---

**Input:** Plaintexts:  $P_1, P_2, \dots, P_N$ 
**Output:**  $N$  pairs of ciphertext and FS data :  $(C_1, \text{FS}_1), (C_2, \text{FS}_2), \dots, (C_N, \text{FS}_N)$ 

```

1: for  $i = 1$  to  $N$  do
2:   Set the plaintext as  $P_i$ ;
3:   Reset fault injection intensity  $F \leftarrow 0$ ;
4:    $C_i \leftarrow E_K(P_i)$  under fault injection intensity  $F = 0$  (No fault injection);
5:   repeat
6:     Increase  $F$  by a certain amount, for example,  $T_{clk} \leftarrow 0.95 \times T_{clk}$ ;
7:      $\text{tmp} \leftarrow E_K(P_i)$  under fault injection intensity  $F$ ;
8:   until  $\text{tmp} \neq C_i$ 
9:    $\text{FS}_i \leftarrow F$ ;
10:  end for
11: return  $(C_i, \text{FS}_i)$  for  $i = 1, 2, \dots, N$ ;

```

---

First of all, the key recovery is performed with divide-and-conquer algorithm. For AES, the key is usually recovered byte by byte. The attackers first guess the value of the target key byte as  $G$ . Then the key guess  $G$  is used with the public data such as ciphertexts to calculate the target intermediate values using the SF. The calculated target intermediate values are used with an LM to obtain the temporary results,  $T$ . The calculated  $T$  is compared with the measured FS data using an EF, and the evaluation result is obtained. For each possible key guess, the mentioned calculation for obtaining the evaluation result is performed. The attackers know the expected result of the evaluation for the correct key guess. Thus, the attackers compare the evaluation results for all key candidates and pick up the one that is the most likely to be the correct key as the attack result.

The key recovery algorithm using FS data targeting the first byte of  $sk_{10}$  of AES-128 is shown in Algorithm 5.12. For each key candidate of  $G$ , the rough estimation of the FS data is performed by  $\text{LM}(S^{-1}(C_i \oplus G))$ , in which  $S^{-1}(C_i \oplus G)$  is the calculation of the target intermediate value with the SF. As for the LM used in step 3 in Algorithm 5.12, the LMs such as the HW and HD models can be used. The effectiveness of the used LM largely depends on the implementation method.

- It is found that for the FSA attack against AES-pprm1, the HW model is an effective LM. There is an array of AND gates at the beginning of the S-box of AES-pprm1. For each AND gate, the output can be determined if any of the inputs is determined to be zero. Each zero input to an AND gate has a potential to make the output gets determined earlier. Therefore, in general, the output of the AND gate array is like to get determined earlier when the input has more zero bits.
- It is also found that clockwise collision model is effective for the FSA attack on AES-comp as the clockwise collision leads to not only a low-power consumption but also a short path delay for the AES-comp S-box.

In Algorithm 5.12, the correlation between the rough estimation of the FS data and the measured FS data is used as the EF. Only when  $G$  is the real one, the calculated intermediate values are correct; thus, the estimated FS data can be expected to be correlated with the measured FS

**Algorithm 5.12** Key Recovery Algorithm Using Fault Sensitivity and Leakage Model

**Input:**  $N$  pairs of ciphertext and FS data :  $(C_1, \text{FS}_1), (C_2, \text{FS}_2), \dots, (C_N, \text{FS}_N)$

**Output:** Recovered key:  $sk_i[0]$

```

1: for  $G = 0$  to  $2^8 - 1$  do
2:   for  $i = 1$  to  $N$  do
3:      $T_{i,G} \leftarrow \text{LM}(S^{-1}(C_i \oplus G));$ 
4:   end for
5:   Prepare a sequence  $X$ , where  $X_i = \text{FS}_i[0]$  for  $i = 0, 1, \dots, 255$ ;
6:   Prepare a sequence  $Y$ , where  $Y_i = T_{i,G}$  for  $i = 0, 1, \dots, 255$ ;
7:    $\text{Cor}_G \leftarrow \rho(X, Y);$ 
8: end for
9:  $sk_i[0] \leftarrow \arg \max_G \text{Cor}_G;$ 
10: return  $sk_i[0];$ 
```

---

data. Thus, the key guess that corresponds to the highest correlation is expected to be the correct key as shown in step 9 in Algorithm 5.12. Note that Algorithm 5.12 is similar to the key recovery algorithms for previously introduced side-channel analysis.

### 5.5.2.6 FSA Key Recovery Algorithm Using Collision Model

When there is no appropriate LM for the FSA attack, it is natural to consider using the collision model in the key recovery. In the collision model, the attackers do not care about the relations between the intermediate value and the FS data. The attackers take advantages of the fact that when two S-boxes of the same implementation have the same input value, the FS of the S-box calculations is similar to each other. One can review Section 5.3.8 to recall the power analysis using the collision model.

The key recovery algorithm using collision model for AES-128 is written in Algorithm 5.13 for the FSA attack, which was proposed in Moradi *et al.* (2011). The target is set to the difference between the first two key bytes of the last round key, that is,  $sk_{10}[0] \oplus sk_{10}[1]$ . Two group separations for  $\text{FS}_i[0]$  and  $\text{FS}_i[1]$  are performed according to the values of  $C_i[0]$  and  $C_i[1]$ , respectively. For the 256 groups for group separation 1, the means of the FS data are calculated as  $\text{MeanFS1}_i$ ,  $i = 0, 1, \dots, 255$ . For the 256 groups for group separation 2, the means of the FS data are calculated as  $\text{MeanFS2}_i$ ,  $i = 0, 1, \dots, 255$ .

With a guess of the key byte difference  $\Delta G$ , the correlation between  $\text{MeanFS1}_i$  and  $\text{MeanFS2}_{i \oplus \Delta G}$  for  $i = 1, 2, \dots, 255$  is calculated. When  $\Delta G$  is the same with the real one, each pair of  $\text{MeanFS1}_i$  and  $\text{MeanFS2}_{i \oplus \Delta G}$  for any  $i$  is a pair of FS data for two S-boxes with the same input value. Thus, a high correlation is expected for the correct  $\Delta G$ . To understand this, recall that the related two S-box inputs of the last AES round can be calculated as  $S^{-1}(C_i[0] \oplus sk_{10}[0])$  and  $S^{-1}(C_i[1] \oplus sk_{10}[1])$ . These two S-box inputs are the same if

$$S^{-1}(C_i[0] \oplus sk_{10}[0]) = S^{-1}(C_i[1] \oplus sk_{10}[1]),$$

or

$$C_i[0] \oplus C_i[1] = sk_{10}[0] \oplus sk_{10}[1].$$

**Algorithm 5.13** Key Recovery Algorithm Using Fault Sensitivity and Collision Model

**Input:**  $N$  pairs of ciphertext and FS data :  $(C_1, \text{FS}_1), (C_2, \text{FS}_2), \dots, (C_N, \text{FS}_N)$

**Output:** Recovered key byte difference:  $sk_{10}[0] \oplus sk_{10}[1]$

```

1: for  $i = 1$  to  $N$  do
2:   In group separation 1, move  $\text{FS}_i[0]$  to the group  $C_i[0]$ ;
3:   In group separation 2, move  $\text{FS}_i[1]$  to the group  $C_i[1]$ ;
4: end for
5: for  $i = 0$  to 255 do
6:   MeanFS1 $_i \leftarrow$  Mean FS of group  $i$  in group separation 1;
7:   MeanFS2 $_i \leftarrow$  Mean FS of group  $i$  in group separation 2;
8: end for
9: for  $\Delta G = 0$  to 255 do
10:  Prepare a sequence  $X$ , where  $X_i = \text{MeanFS1}_i$  for  $i = 0, 1, \dots, 255$ ;
11:  Prepare a sequence  $Y$ , where  $Y_i = \text{MeanFS2}_{i \oplus \Delta G}$  for  $i = 0, 1, \dots, 255$ ;
12:   $\text{Cor}_{\Delta G} \leftarrow \rho(X, Y)$ ;
13: end for
14:  $sk_{10}[0] \oplus sk_{10}[1] \leftarrow \arg \max_G \text{Cor}_{\Delta G}$ ;
15: return  $sk_{10}[0] \oplus sk_{10}[1]$ ;

```

In addition, recall that the ciphertexts for  $\text{MeanFS1}_i$  satisfy  $C_i[0] = i$ , and the ciphertexts for  $\text{MeanFS2}_{i \oplus \Delta G}$  satisfy  $C_i[1] = i \oplus \Delta G$ . When  $\Delta G = sk_{10}[0] \oplus sk_{10}[1]$ , for  $\text{MeanFS1}_i$  and  $\text{MeanFS2}_{i \oplus \Delta G}$ , it is satisfied that

$$C_i[0] \oplus C_i[1] = i \oplus i \oplus \Delta G = sk_{10}[0] \oplus sk_{10}[1].$$

Thus, for  $\text{MeanFS1}_i$  and  $\text{MeanFS2}_{i \oplus \Delta G}$ , the S-box inputs are the same when  $\Delta G$  is the same with the real one. At step 14 in Algorithm 5.13, the  $\Delta G$  corresponding to the highest correlation is set to the attack result.

The successful key recovery of the FSA attack using collision model has been applied to many AES implementations. On the one hand, with the FSA attack using collision model, the attackers' required knowledge for a successful key recovery is largely reduced. On the other hand, as shown in both Algorithms 5.12 and 5.13, the value of the faulty ciphertext  $C'$  is not used in the key recovery of FSA. Therefore, some DFA countermeasures are also valid targets under the attack concept of the FSA attack.

**Exercise 5.14** Summarize the difference of the attack requirements for the DFA attack and the FSA attack such as the value of  $C'$ , the fault model, the control of the fault injection intensity, and so on.

**Exercise 5.15** Compare the algorithmic noise for power analysis and the FSA attacks explained in this chapter.

## Acknowledgment

The authors greatly appreciate Prof. Tsutomu Matsumoto at Yokohama National University, Japan, for kind permission to use the power consumption data measured in his research group, which can be found at (Research Center for Information and Physical Security, Yokohama National University). It highly enhanced the accomplishment level of this chapter.

## Bibliography

- Ali S and Mukhopadhyay D 2011 A Differential Fault Analysis on AES Key Schedule Using Single Fault In *FDTC* (ed. Breveglieri L, Guillet S, Koren I, Naccache D and Takahashi J), pp. 35–42. IEEE.
- Brier E, Clavier C and Olivier F 2004 Correlation power analysis with a leakage model. *CHES 2004*, vol. 3156 of *Lecture Notes in Computer Science*, pp. 16–29. Springer-Verlag.
- Kocher PC, Jaffe J and Jun B 1999 Differential power analysis *CRYPTO 1999*, vol. 1666 of *Lecture Notes in Computer Science*, pp. 388–397. Springer-Verlag.
- Moradi A, Mischke O and Eisenbarth T 2010 Correlation-enhanced power analysis collision attack In *CHES* (ed. Mangard S and Standaert FX), vol. 6225 of *Lecture Notes in Computer Science*, pp. 125–139. Springer-Verlag.
- Moradi A, Mischke O, Paar C, Li Y, Ohta K and Sakiyama K 2011 On the power of fault sensitivity analysis and collision side-channel attacks in a combined setting In *CHES* (ed. Preneel B and Takagi T), vol. 6917 of *Lecture Notes in Computer Science*, pp. 292–311. Springer-Verlag.
- Morioka S and Satoh A 2002 An optimized S-box circuit architecture for low power AES design In *CHES* (ed. Kaliski BS Jr., Koç ÇK and Paar C), vol. 2523 of *Lecture Notes in Computer Science*, pp. 172–186. Springer-Verlag.
- Mukhopadhyay D 2009 An improved fault based attack of the advanced encryption standard In *AFRICACRYPT* (ed. Preneel B), vol. 5580 of *Lecture Notes in Computer Science*, pp. 421–434. Springer-Verlag.
- Piret G and Quisquater JJ 2003 A differential fault attack technique against SPN structures, with application to the AES and KHAZAD *CHES 2003*, vol. 2779 of *Lecture Notes in Computer Science*, pp. 77–88. Springer-Verlag.
- Research Center for Information and Physical Security, Yokohama National University n.d. Power Consumption Data Measured Using LSI on SASEBO-R <http://ipsr.ynu.ac.jp/wxf/index.html>.
- Satoh A, Morioka S, Takano K and Munetoh S 2001 A compact Rijndael hardware architecture with S-box optimization In *ASIACRYPT* (ed. Boyd C), vol. 2248 of *Lecture Notes in Computer Science*, pp. 239–254. Springer-Verlag.
- Tunstall M, Mukhopadhyay D and Ali S 2011 Differential fault analysis of the advanced encryption standard using a single fault In *Information Security Theory and Practice. Security and Privacy of Mobile Devices in Wireless Communication* (ed. Ardagna C and Zhou J), vol. 6633 of *Lecture Notes in Computer Science*, pp. 224–233. Springer-Verlag Berlin and Heidelberg.



# 6

## Advanced Fault Analysis with Techniques from Cryptanalysis

As introduced in Chapter 5, fault analysis causes a faulty computation during the encryption process. The basic differential fault analysis (DFA) exploits the difference of the correct computation and the faulty computation and recovers the key using the principle of differential cryptanalysis. Once the desired fault is injected to the middle of the computation, the remaining analysis is basically the same as the theoretical cryptanalysis. In fact, many of the techniques developed for the theoretical cryptanalysis can also be used in fault analysis.

On the contrary, fault analysis requires to deal with several problems that do not appear in the theoretical cryptanalysis. For example, in the theoretical differential cryptanalysis, the value of input difference can be chosen by the attacker because the input difference is the one between two plaintexts that are chosen by the attacker in the chosen plaintext attack model. On the other hand, in the DFA, the value of input difference cannot be chosen by the attacker because the fault injection cannot be controlled by the attacker in the random fault model.

The purpose of this chapter is applying the techniques for the theoretical cryptanalysis to fault analysis in order to optimize the attack. There are two approaches of optimizing fault analysis.

**Relaxing the fault model:** This is very important for fault analysis. To recover the key in practice, the fault model must match the effect that can actually be caused by physical fault injections. In general, the more expensive devices are used, the more accurate and complicated faults can be injected. Relaxing the fault model is meaningful in the sense that fault analysis becomes more feasible.

**Reducing the cost to recover the key:** Different from theoretical cryptanalysis, the goal of the fault analysis is recovering the key in practice, rather than finding a shortcut attack, which is infeasible but theoretically faster than the brute force attack. For example, for AES with a 128-bit key, the attack with a computational cost of  $2^{120}$  AES operations is regarded as a vulnerability of the AES algorithm in the sense of theoretical cryptanalysis, whereas operating  $2^{120}$  AES encryptions are computationally infeasible with the current computation

resources, and thus is not important in the sense of fault analysis. There seems to exist a consensus in the current cryptographic community that performing  $2^{64}$  operations is hard or at least too expensive for fault analysis. In another aspect, a small number of fault injections is very important for fault analysis. This is because fault analysis is an active attack, that is, it actually gives unusual physical impact to the device. The device can be broken by such a physical impact, and then the attacker faces two problems:

1. The attacker cannot continue the key recovery procedure.
2. The attacker may be detected by the owner of the device that the device is being attacked.

## 6.1 Optimized Differential Fault Analysis

The basic DFA in Chapter 5 only introduced the basic concept, but did not explain the full description of the attack or advanced techniques for optimization. This section aims to optimize DFA.

### 6.1.1 Relaxing Fault Model

Recall Figure 5.75 in Chapter 5, which is a differential propagation in basic DFA. The assumed fault model in this attack is as follows:

*1 byte of fault is injected at the beginning of the 8th round of AES-128. The faulty byte position is known to attackers, while the faulty value is not known to attackers.*

As mentioned before, relaxing the fault model is important for fault analysis, hence optimized DFA relaxes it as follows:

*1 byte of fault is injected at the beginning of the 8th round of AES-128. The faulty byte position is **unknown** to attackers. The faulty value, either.*

The assumption is relaxed in optimized DFA, that is, the faulty byte position becomes unknown to the attackers. This is very meaningful for applications in practice. The attacker indeed cannot detect the faulty byte position after some physical impact is added to the device. Several experimental reports also indicate that causing the fault on a fixed target byte position with probability 1 is infeasible. Hence, DFA that can deal with unknown faulty byte position is practically meaningful.

#### 6.1.1.1 Obtaining $(P, C, C')$

The generation of correct and faulty ciphertexts is basically the same as basic DFA. The attacker assumes that a plaintext  $P$  is encrypted twice. For the first time, the attacker observes the corresponding ciphertext  $C$  without injecting fault, and achieves a pair of  $(P, C)$ . For the second time, the attacker injects a 1-byte fault at the beginning of round 8, and obtains the corresponding faulty ciphertext  $C'$ . Note that the timing of the fault injection (the beginning

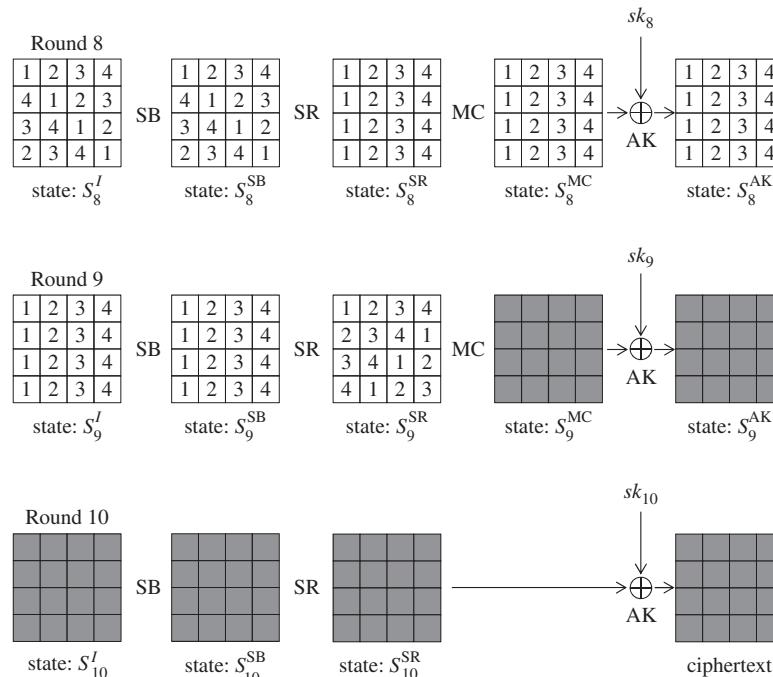
of round 8) should be measured before the analysis. It is widely known that the timing can be measured quite accurately. In summary, the attacker obtains a pair of  $(P, C)$  and the faulty ciphertexts  $C'$  generated by the fault satisfying the fault model.

### 6.1.2 Four Classes of Faulty Byte Positions

On the basis of this fault model, the differential propagation in Figure 5.75 is reviewed. The new differential propagation is depicted in Figure 6.1. From the new fault model, the 1-byte fault is injected at state  $S_8^I$  but its position is unknown. The straightforward method for the attacker is guessing the faulty byte position exhaustively, and performing the basic DFA for all the guesses. Because the state consists of 16 bytes, the number of guesses is at most 16, which roughly concludes that DFA with the new fault model can finish with 16 times as much cost as basic DFA.

The attack cost can be improved more. The 16 candidates of the faulty byte position are divided into the following four classes:

- Class 1:** byte positions 0, 5, 10, and 15.
- Class 2:** byte positions 3, 4, 9, and 14.
- Class 3:** byte positions 2, 7, 8, and 13.
- Class 4:** byte positions 1, 6, 11, and 12.



**Figure 6.1** Differential propagation for four classes of faulty byte position at  $S_8^I$

The numbers in each state in Figure 6.1 show the differential propagation for each class.

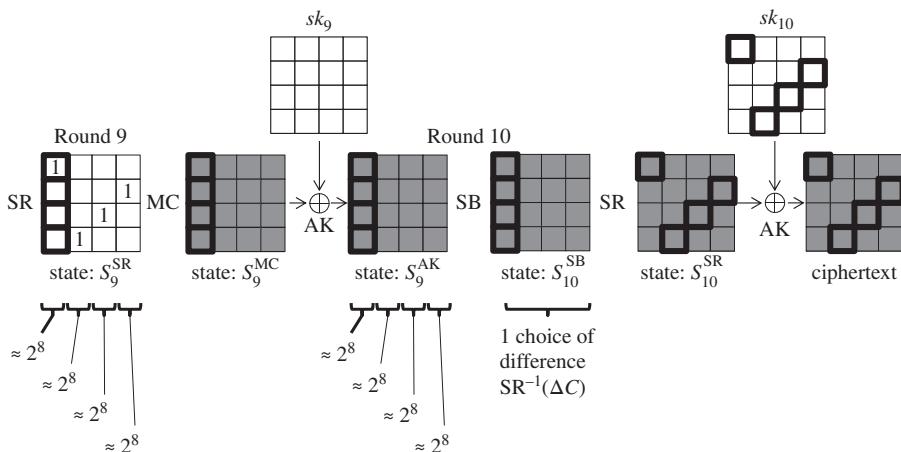
Four positions in each class can be analyzed with exactly the same procedure. Therefore, the attack cost can be at most four times as much as basic DFA. Suppose that the fault is injected to any 1 byte of  $S_8^I[0], S_8^I[5], S_8^I[10], S_8^I[15]$ . The corresponding active byte position after the SubBytes operation is 1 byte of  $S_8^{SB}[0], S_8^{SB}[5], S_8^{SB}[10], S_8^{SB}[15]$ , and the corresponding active byte position after the ShiftRows operation is 1 byte of  $S_8^{SR}[0], S_8^{SR}[1], S_8^{SR}[2], S_8^{SR}[3]$ . The important fact is that in any of the four cases, the input state to the MixColumns operation,  $S_8^{MC}$ , contains 1 active byte in the left most column. The MDS property of the MixColumns operation guarantees that the number of active bytes in the input and the output columns is greater than or equal to 5. Therefore, in any of the four cases, all the 4 bytes in the left most column (labeled as “1”) of  $S_8^{MC}$  are active. In the remaining part of the attack, only the information of 4 active byte positions in  $S_8^{MC}$  is used. Thus, 4 faulty byte positions in class 1 are equivalent for the analysis. Similarly, 4 faulty byte positions in each of the other classes can be shown to be equivalent for the analysis. Classes 2, 3, and 4 contain 4 active bytes labeled as “2,” “3,” and “4” at  $S_8^{MC}$ , respectively.

### 6.1.3 Recovering Subkey Candidates of $sk_{10}$

The key recovery procedure is iterated four times by exhaustively guessing the class of the faulty byte position. In the following, the analysis for class 1 is explained.

DFA first aims to recover the last subkey  $sk_{10}$ . The overall picture for this process is shown in Figure 6.2.

The attacker analyzes the difference of two encryption processes, one is for computing  $C$  and the other is for computing  $C'$ . When the faulty byte position at  $S_8^I$  belongs to class 1, 4 bytes of  $S_9^{SR}[0, 7, 10, 13]$  have nonzero difference. From the MDS property of the MixColumns operation, all bytes of  $S_9^{MC}$  have nonzero difference. The number of possible differences for each input column is  $255 \approx 2^8$ , thus the number of possible differences for



**Figure 6.2** Recovery of 4 bytes of  $sk_{10}$  for class 1

each output column is also  $2^8$ . Because the AddRoundKey operation does not affect the difference, the same difference is preserved until  $S_9^{AK}$ , which is equivalent to  $S_{10}^I$ . From the analysis so far, the input difference to the S-box in round 10,  $\Delta S_{10}^I$ , is determined uniquely for each possibility of  $\Delta S_9^{SR}$ .

The output difference from the S-box in round 10,  $\Delta S_{10}^{SB}$ , can be computed from the ciphertext pair  $(C, C')$ .  $C$  and  $C'$  are known values to the attacker, thus its difference  $\Delta C = C \oplus C'$  can be computed. During the backward computation, the values are soon hidden by the last subkey XOR with  $sk_{10}$ , however, the difference can still be traced. In fact  $\Delta S_{10}^{SB}$  can be computed by  $SR^{-1}(\Delta C)$ .

As explained in differential cryptanalysis in Section 4.2, for a pair of input and output differences for S-box, the paired values can be obtained using the differential distribution table (DDT) of the AES S-box. The idea was introduced in Figure 4.22 as a technique for driving key suggestions without guess. For DFA, subkey candidates for  $sk_{10}$  are derived with this technique, and the analysis is processed column by column. The analysis for the left most column is stressed by bold line in Figure 6.2.

How to derive the paired values of the left most column of  $S_{10}^{SB}$  is explained below. The 4-byte output difference is uniquely fixed by  $SR^{-1}(\Delta C)$ , and there are about  $2^8$  choices for the 4-byte input difference of  $\Delta S_{10}^I$ . According to the DDT of the AES S-box, a pair of randomly determined input and output differences of the S-box has

- two solutions with probability 126/256,
- four solutions with probability 1/256, and
- no solution with probability 129/256.

For the sake of simplicity, the above can be roughly approximated such that a pair of randomly determined input and output differences of the S-box has two solutions with probability 1/2, otherwise, the pair does not have a solution. A pair of input and output differences is called “matched” when they have solutions to satisfy its propagation. For each of the  $2^8$  choices of  $\Delta S_{10}^I$ , the existence of the solution to achieve  $SR^{-1}(\Delta C)$  is checked with DDT. Because the match is examined for 4 bytes, the probability that there exist solutions for all bytes is  $(1/2)^4 = 2^{-4}$ . Once the match is verified for 4 bytes, 4 byte values can be fixed to any of the solutions. The number of solution for each byte is 2, and any combination of the solutions of 4 bytes can make a valid value of  $S_{10}^{SB}$ . Thus,  $2^4$  solutions of  $S_{10}^{SB}$  are obtained from each match. In total,  $2^8 \cdot 2^{-4}$  choices of the input difference can match in all of the 4 bytes, and  $2^8 \cdot 2^{-4} \cdot 2^4 = 2^8$  solutions are obtained, which yields  $2^8$  values of the left most column of  $S_{10}^{SB}$ .

$2^8$  values  $S_{10}^{SB}[0, 1, 2, 3]$  will move to  $S_{10}^{SR}[0, 7, 10, 13]$  by the subsequent ShiftRows operation. Finally,  $2^8$  candidates of  $sk_{10}[0, 7, 10, 13]$  can be computed as

$$sk_{10}[0, 7, 10, 13] \leftarrow S_{10}^{SR}[0, 7, 10, 13] \oplus C[0, 7, 10, 13]. \quad (6.1)$$

### 6.1.3.1 Obtaining Subkey Candidates for Entire $sk_{10}$

After obtaining  $2^8$  candidates of  $sk_{10}[0, 7, 10, 13]$ , the analysis is performed for the other columns independently. The analysis is almost the same. The only difference is that the input difference comes from a different byte position of  $S_9^{SR}$ , which does not give significant impact to the attack procedure. Thus,  $2^8$  candidates of  $sk_{10}[1, 4, 11, 14]$ ,  $2^8$  candidates of

$sk_{10}[2, 5, 8, 15]$ , and  $2^8$  candidates of  $sk_{10}[3, 6, 9, 12]$  are obtained by iterating the same procedure for the second, third, and the fourth column of the SubBytes operation in the last round.

### 6.1.3.2 Recovery of the Original Key $K (= sk_0)$

The correct value of  $sk_{10}$  is now limited to  $2^8$  candidates per diagonal, which makes  $2^{32}$  candidates of the entire  $sk_{10}$ . The correctness of those  $2^{32}$  candidates can be verified by the exhaustive search. The attacker uses the pair of  $(P, C)$ . For each of the  $2^{32}$  candidates of  $sk_{10}$ , the attacker computes the corresponding  $sk_0$  by inverting the key schedule function. The correctness of the candidate can be checked by matching  $\text{AES}_{sk_0}(P)$  and  $C$ , where  $\text{AES}_{sk_0}(P)$  represents that plaintext  $P$  is processed by AES under the key  $sk_0$ . This is a match of 128-bit values. With  $2^{32}$  matching trials, only the correct key value can result in the match, and thus the correct key is recovered.

### 6.1.3.3 Remarks

The analysis of the subkey or key recovery is performed for each guess of the class of the faulty byte position. Hence, the procedure is iterated four times. If the guess of the class is wrong, the attacker fails the validity check for all of  $2^{32}$  candidates of  $sk_{10}$ . Only if the guess of the class is correct, the attacker obtains the correct suggestion of  $sk_0$ .

## 6.1.4 Attack Procedure

The attack procedure in an algorithmic form is described in Algorithm 6.1.

### 6.1.4.1 Complexity Evaluation

In Algorithm 6.1, the computational complexity inside the loop of step 4 is  $2^8$  column operations, which would require about 1/4 cost of the round function. Then, owing to the loop in step 3, the column-wise operations are iterated four times, which makes the cost inside the loop of step 3  $2^8$  round function operations. The computational cost for the loop of step 15 is  $2^{32}$  AES computations. Finally, the computational cost for the loop of step 2 becomes  $4 \cdot (2^8 + 2^{32}) \approx 2^{34}$  AES computations.

The data complexity is a valid pair of plaintext  $P$  and its correct ciphertext  $C$  and the faulty ciphertext  $C'$ . The number of fault injection is only 1. The required memory amount is storing  $4 \cdot 2^8$  32-bit values for  $L_1, L_2, L_3, L_4$ , which is about  $2^8$  AES state values.

The attack complexity is summarized as follows:

$$(Data, Time, Memory, \#Faults) = (2, 2^{34}, 2^8, 1). \quad (6.2)$$

The attack complexity is feasible, which meets the requirement for side-channel analysis and fault analysis.

**Algorithm 6.1** Optimized DFA

---

**Input:**  $P, C, C'$   
**Output:**  $K (= sk_0)$

- 1: Compute  $\Delta S_{10}^{\text{SB}} \leftarrow \text{SR}^{-1}(C \oplus C');$
- 2: **for** class  $i \in \{1, 2, 3, 4\}$  of the faulty byte position **do**
- 3:   **for** column  $j \in \{0, 1, 2, 3\}$  of the SubBytes operation in round 10 **do**
- 4:     **for** about  $2^8$  possibilities of the 1-byte difference at  $S_9^{\text{SR}}$  **do**
- 5:       Compute the corresponding difference  $\Delta S_{10}^I$  for column  $j$ ;
- 6:       **if** the input and output differences have solutions in all four S-boxes of column  $j$  in round 10 **then**
- 7:         Obtain  $2^4$  solutions of  $S_{10}^{\text{SB}}$  and the corresponding value of  $S_{10}^{\text{SR}}$ ;
- 8:         **for**  $2^4$  solutions of  $S_{10}^{\text{SR}}$  **do**
- 9:           Compute  $sk_{10} \leftarrow S_{10}^{\text{SR}} \oplus C$  for the 4 bytes in position  $\text{SR}(\text{Col}(j))$ ;
- 10:          Store the computed 4 bytes of subkey  $sk_{10}[\text{SR}(\text{Col}(j))]$  in a list  $L_j$ ;
- 11:         **end for**
- 12:       **end if**
- 13:     **end for**
- 14:   **end for**
- 15: **for**  $2^{32}$  values of  $sk_{10}$  generated with all possible combinations of  $L_0, L_1, L_2, L_3$  **do**
- 16:     Compute  $sk_0$  with the inverse of the key schedule function;
- 17:     **if**  $\text{AES}_{sk_0}(P) = C$  **then**
- 18:       **return**  $sk_0$ ;
- 19:     **end if**
- 20: **end for**
- 21: **end for**

---

### 6.1.5 Probabilistic Fault Injection

In a real environment, injecting the fault in the byte position with probability 1 is hard. For example,

- The fault may be injected in a different timing.
- The fault may be injected in several bytes nearby the target byte position.

If the desired fault is not guaranteed with probability 1, the attack requires more fault injections and its complexity increases. Fault injection that can succeed only probabilistically is called **noisy fault injection**, meaning that the intended fault only occurs with probability  $p$ , and the useless fault (noise) is obtained with probability  $1 - p$ .

Optimized DFA has a good characteristic against noisy fault injection. Namely, it still can recover the key practically. The attacker performs  $p^{-1}$  fault injections and collects  $p^{-1}$  tuples of  $(P, C, C')$ . Because the impact of the fault at state  $S_8^I$  propagates to all bytes at the ciphertext, there is no method to efficiently detect the desired or undesired fault injection. Thus, the attacker analyzes all the tuples by assuming that each tuple is obtained by the desired fault injection. Note that if the pair is the wrong pair, the key recovery procedure does not return anything, and thus the attacker eventually finds the right tuple caused by the desired fault injection.

This increases the attack cost only linearly to the number of pairs analyzed. In summary, the key is recovered with the following cost:

$$(Data, Time, Memory, \#Faults) = (2 \cdot p^{-1}, 2^{34} \cdot p^{-1}, 2^8 \cdot p^{-1}, p^{-1}). \quad (6.3)$$

According to the experiment of the fault injection,  $p$  is not so small. Assuming  $p = 1/10$  is sufficient. Then, the attack complexity is only 10 times of the original optimized DFA, which is still feasible.

**Exercise 6.1** When a random byte fault is accidentally injected in a later timing, say at the beginning of the 9th or 10th round ( $S_9^I, S_{10}^I$ ), the attacker can detect the occurrence of this event easily. Answer the reason.

**Exercise 6.2** In Figure 6.1, assume a more powerful attacker who can inject a desired difference value  $\Delta$  to a target byte at  $S_8^{\text{SB}}$ .

- (a) How does it affect the attack complexity?
- (b) The attacker can recover the key without knowing the plaintext value. Answer the reason.

### 6.1.6 Optimized DFA with the MixColumns Operation in the Last Round †

It is often misunderstood that the absence of the MixColumns operation in the last round enables optimized DFA to recover the last round key efficiently. In other words, it is often misunderstood that, if the AES encryption algorithm computed the MixColumns operation in the last round, efficient DFA could be prevented.

This section explains that, even with the MixColumns operation in the last round, optimized DFA can recover the key with exactly the same cost as the case of the original AES. Note that the modified AES is inconsistent with the original AES, namely, they do not return the same ciphertext for the same pair of plaintext and key. The purpose of this discussion is understanding the impact of the last MixColumns operation.

For clarity, the structure of the modified AES along with the differential characteristic for DFA is described in Figure 6.3.

#### 6.1.6.1 Hardness of Straightforward Application to Modified AES

What does cause the misunderstanding? In order to explain it, Figure 6.2 and Algorithm 6.1 are reviewed. Optimized DFA for the original AES analyzes the internal state value column by column over the SubBytes operation in round 10, and then the corresponding 4-byte subkey

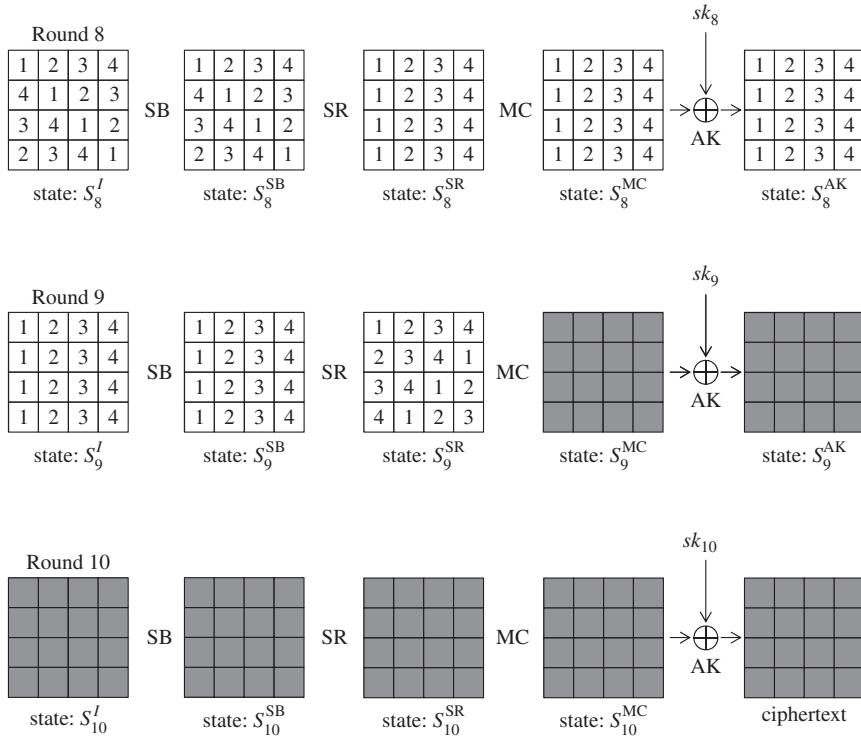


Figure 6.3 Differential propagation against modified AES

candidates of  $sk_{10}$  are obtained. This corresponds to step 9 of Algorithm 6.1, and is depicted in Figure 6.2.

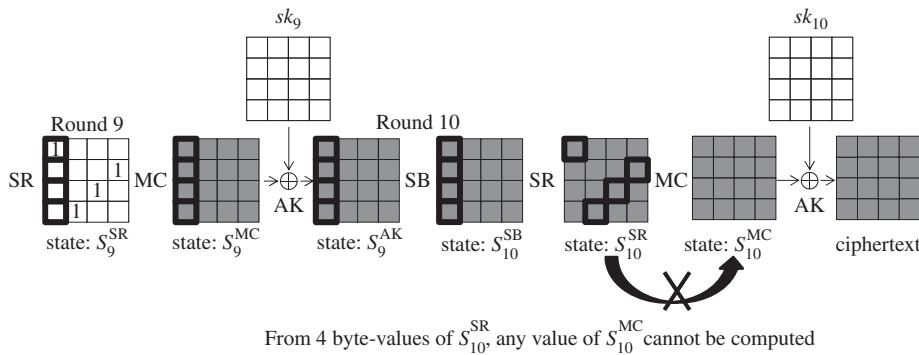
In the original AES, computing 4 bytes of  $S_{10}^{SR} \oplus C$  in the byte position  $SR(Col(j))$  can be trivially done. However, if the MixColumns operation is used in round 10, the computation becomes as depicted in Figure 6.4. Namely, the 4 bytes in a column obtained at state  $S_{10}^{SB}$  will move to four different columns with the subsequent ShiftRows operation. Then, the attacker cannot compute the subsequent MixColumns operation without knowing the values of other bytes. This prevents the attacker from obtaining 4 byte values of  $sk_{10}$ .

Although optimized DFA on the original AES cannot be applied to the modified AES in a straightforward manner, the attack procedure can be modified so that the key is recovered with the same efficiency. There are two approaches of modifying the attack procedure.

### 6.1.6.2 Storing Internal State Values

The idea is very simple. To recover the key, storing 4 bytes of  $sk_{10}$  after the analysis of each column over the SubBytes operation in the last round is not necessary.

In the original AES, if the internal state value of  $S_{10}^{SB}[Col(j)]$  is recovered, the corresponding  $sk_{10}[SR(Col(j))]$  is obtained accordingly using the ciphertext. In other words, the internal state value of  $S_{10}^{SB}[Col(j)]$  can be converted into the  $sk_{10}[SR(Col(j))]$  by 1-to-1 mapping.



**Figure 6.4** Impossibility of straightforward recovery of  $sk_{10}$  for class 1

With this property, the core idea of optimized DFA for the original AES can be summarized in Algorithm 6.2.

Considering that the conversion from the internal state value of  $S_{10}^{SB}[Col(j)]$  to  $sk_{10}[SR(Col(j))]$  can be applied at any timing, the pseudocode can be changed as shown in Algorithm 6.3.

Nothing is changed but for the timing of the conversion from  $S_{10}^{SB}$  to  $sk_{10}$ . The important learning from those two algorithms is that the mechanism of reducing the candidates in optimized DFA is obtaining only  $2^8$  candidates of  $S_{10}^{SB}$  using differential cryptanalysis. It is easy to see that storing 4 bytes of  $sk_{10}$  for each diagonal is not the main mechanism.

With this observation, DFA for the modified AES with the MixColumns operation in the last round is constructed. Adding the MixColumns operation in the last round does not affect the core mechanism of DFA, that is, reducing the internal state values during the SubBytes operation in the last round. Hence, the key should be recovered as efficiently as the original AES. Actually, the procedure in Algorithm 6.2 can be applied to the modified AES quite

---

#### Algorithm 6.2 Pseudo-code for Core Idea of Optimized DFA

---

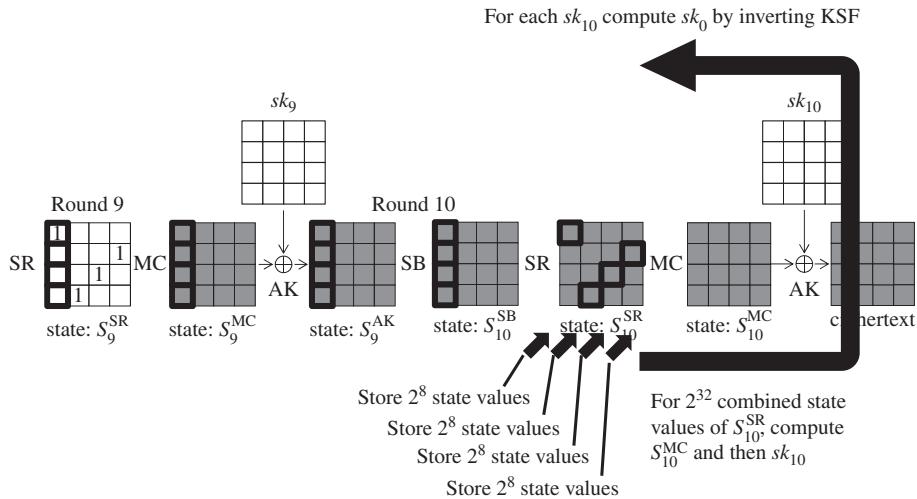
- 1: **for** 4 columns of the SubBytes operation in the last round **do**
  - 2:   Obtain  $2^8$  candidates of  $S_{10}^{SB}$  for each column;
  - 3:   Convert them into each diagonal of  $sk_{10}$  from the 1-to-1 relationship and store them;
  - 4: **end for**
  - 5: Combine 4 diagonals, and exhaustively test the correctness of  $2^{32}$  candidates of  $sk_{10}$ ;
- 

---

#### Algorithm 6.3 Changing Conversion Timing of Optimized DFA

---

- 1: **for** 4 columns of the SubBytes operation in the last round **do**
  - 2:   Obtain  $2^8$  candidates of  $S_{10}^{SB}$  for each column and store them;
  - 3: **end for**
  - 4: Combine 4 columns to make  $2^{32}$  candidates of  $S_{10}^{SB}$ ;
  - 5: Convert them into  $sk_{10}$  by using the 1-to-1 relationship, and check the correctness;
-



**Figure 6.5** Storing  $2^8$  internal state values for each diagonal against modified AES

simply. After  $2^8$  internal state values are obtained as a solution of the differential transition through the S-box for each column, the attacker stores the corresponding 4 bytes in the diagonal of state  $S_{10}^{\text{SR}}$ . After obtaining  $2^8$  candidates for each diagonal of  $S_{10}^{\text{SR}}$ , the attacker generates the exhaustive combinations of four diagonals, which generates  $2^{32}$  candidates of the entire  $S_{10}^{\text{SR}}$ . For each of the  $2^{32}$  values of  $S_{10}^{\text{SR}}$ , the last subkey  $sk_{10}$  is recovered as

$$sk_{10} \leftarrow MC(S_{10}^{\text{SR}}) \oplus C. \quad (6.4)$$

Then, the corresponding secret key  $sk_0$  can be computed by inverting the key schedule function, and its correctness can be verified by checking if  $C$  matches  $\text{AES}_{sk_0}(P)$  with a pair of plaintext and ciphertext  $(P, C)$ . The attack is depicted in Figure 6.5.

The procedure of DFA against the modified AES with the MixColumns operation in the last round is given in Algorithm 6.4.

Owing to the similarity of the procedure, the detailed complexity evaluation is omitted. The correct  $sk_0$  can be recovered with the same complexity as Algorithm 6.1.

### 6.1.6.3 Equivalent Transformation of Subkey Addition

The first approach with storing the internal state values illustrated the core mechanism of DFA very well. The second approach does not mention the core mechanism but seems to be much simpler.

In the second approach, the technique of the equivalent transformation of subkey addition is used. This technique was used for the impossible differential cryptanalysis in Section 4.3, which represents the computations of the AES round function in an alternative method. In particular, the order of two linear computations MixColumns and AddRoundKey is exchanged.

By taking  $S_{10}^I$  and  $sk_{10}$  as input, the computation for round 10 can be written as follows:

$$C \leftarrow (MC \circ SR \circ SB(S_{10}^I)) \oplus sk_{10}. \quad (6.5)$$

**Algorithm 6.4** DFA against AES with MixColumns in the Last Round

---

**Input:**  $P, C, C'$   
**Output:**  $K (= sk_0)$

- 1: Compute  $\Delta S_{10}^{\text{SB}} \leftarrow \text{SR}^{-1} \circ \text{MC}^{-1}(C \oplus C');$
- 2: **for** class  $i \in \{1, 2, 3, 4\}$  of the faulty byte position **do**
- 3:   **for** column  $j \in \{0, 1, 2, 3\}$  of the SubBytes operation in round 10 **do**
- 4:     **for** about  $2^8$  possibilities of the 1-byte difference at  $S_9^{\text{SR}}$  **do**
- 5:       Compute the corresponding difference  $\Delta S_{10}^I$  for column  $j$ ;
- 6:       **if** the input and output differences have solutions in all four S-boxes in round 10 **then**
- 7:         Obtain  $2^4$  solutions of  $S_{10}^{\text{SB}}[\text{Col}(j)]$ ;
- 8:         Store the corresponding 4 bytes in  $S_{10}^{\text{SR}}[\text{SR}(\text{Col}(j))]$  in a list  $L_j$ ;
- 9:       **end if**
- 10:      **end for**
- 11:     **end for**
- 12:   **for**  $2^{32}$  values of  $S_{10}^{\text{SR}}$  generated with all possible combinations of  $L_0, L_1, L_2, L_3$  **do**
- 13:     Compute  $sk_{10} \leftarrow \text{MC}(S_{10}^{\text{SR}}) \oplus C$ ;
- 14:     Compute  $sk_0$  from  $sk_{10}$  with the inverse of the key schedule function;
- 15:     **if**  $\text{AES}_{sk_0}(P) = C$  **then**
- 16:       **return**  $sk_0$ ;
- 17:     **end if**
- 18:   **end for**
- 19: **end for**

---

Let  $sk_{10}^*$  be  $\text{MC}^{-1}(sk_{10})$ . Then, the above-mentioned equation can be converted as follows:

$$C \leftarrow \text{MC}(\text{SR} \circ \text{SB}(S_{10}^I) \oplus sk_{10}^*). \quad (6.6)$$

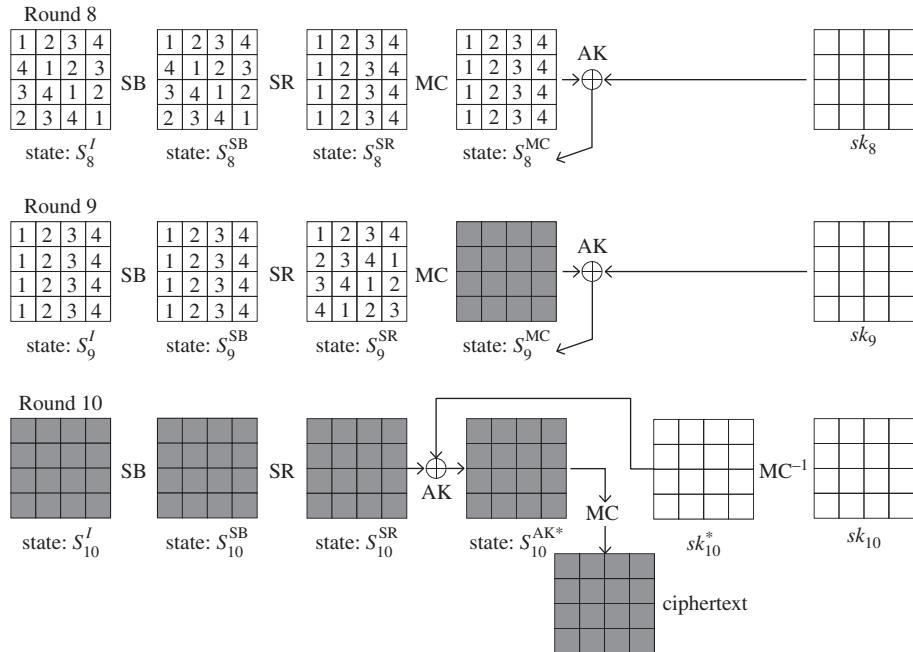
Namely, the order of the MixColumns and AddRoundKey operations is exchanged. The computation structure along with the differential propagation for DFA is described in Figure 6.6.

For each of the obtained ciphertexts  $C$  and  $C'$ , the corresponding state  $S_{10}^{\text{AK}*}$  is easily computed by the inverse MixColumns operation. Because the order of the MixColumns and AddRoundKey operations is exchanged,  $\text{MC}^{-1}(C)$  can be computed without the knowledge of the key.

Then, the remaining structure is almost the same as the DFA against the original AES, which is drawn in Figure 6.1. The only difference is that the recovered subkey is  $sk_{10}^* = \text{MC}^{-1}(sk_{10})$ , instead of  $sk_{10}$ . However, when  $2^{32}$  candidates of  $sk_{10}$  are exhaustively examined in the original AES, the attacker can first apply the MixColumns operation to  $sk_{10}^*$ , and then invert it with the key schedule function. Finally, the correct  $sk_0$  can be recovered with the same complexity as DFA against the original AES.

### 6.1.7 Countermeasures against DFA and Motivation of Advanced DFA

As explained in this section, DFA is a strong attack, and thus the countermeasures are often implemented. The details will be explained in Chapter 7. An example is that the system computes the same plaintext twice and checks if the results of the two computations are always the



**Figure 6.6** DFA against modified AES with equivalent transformation of subkey addition

same. If the computation results do not match, the system halts the computation immediately, and never outputs the faulty ciphertext  $C'$ . While this countermeasure can detect DFA, the efficiency loss is not small. The overhead is 100%, which is not acceptable for some environment.

In order to minimize the overhead, the number of rounds computed twice can be minimized. If the fault injection can be prevented for the last three rounds, the DFA cannot recover the key efficiently. Considering DFA against the AES decryption algorithm, protecting the first three rounds and the last three rounds, in total protecting six rounds, is enough to prevent DFA, which reduces the overhead to 60%.

Given the situation, cryptographers have developed other types of DFA in order to recover the key even if the first and last three rounds are protected. In the remaining sections, such advanced DFA will be explained.

## 6.2 Impossible Differential Fault Analysis

As long as the fault is injected during the last three rounds under the random byte-fault model, the optimized DFA is highly efficient to be practical, and thus the motivation of developing other types of fault analysis is weak. It is assumed that the last three rounds of the AES encryption are protected against the fault injection. The fault can be injected only in the 7th round or earlier during the encryption process. This section explains that the impossible differential cryptanalysis enables the attacker to recover the key with byte fault injected at the beginning of the seventh round. Hereafter, the combination of DFA with impossible differential cryptanalysis is called **impossible DFA**.

### 6.2.1 Fault Model

Because the last three rounds are protected, the attacker injects the fault at the beginning of the 7th round  $S_7^I$ . Two types of fault model can be considered, and the attack cost depends on the strength of the fault model assumed. The first fault model is as follows:

*1 byte of fault is injected at the beginning of the 7th round of AES-128. The faulty byte position is unknown to attackers. The faulty value, either. Every time the fault is injected, the faulty byte position may change, and the (unknown) fault value is assumed to be determined accordingly to the uniform distribution for 1-byte values.*

The second fault model is as follows:

*1 byte of fault is injected at the beginning of the 7th round of AES-128. The faulty byte position is unknown to attackers. The faulty value, either. Every time the fault is injected, the attacker can cause the fault in the same but unknown byte position. Every time the fault is injected, the (unknown) fault value is assumed to be determined accordingly to the uniform distribution for 1-byte values.*

Compared with DFA, impossible DFA requires more fault injections. Thus, the assumption for multiple fault injections is added in the fault model. The second fault model assumes stronger property than the first one. Indeed, owing to the assumed ability, that is, the ability of targeting an identical faulty byte position, the attack cost can be smaller than the one in the first model. In the following, impossible DFA with the first assumption is explained. Then, impossible DFA with the second assumption is explained.

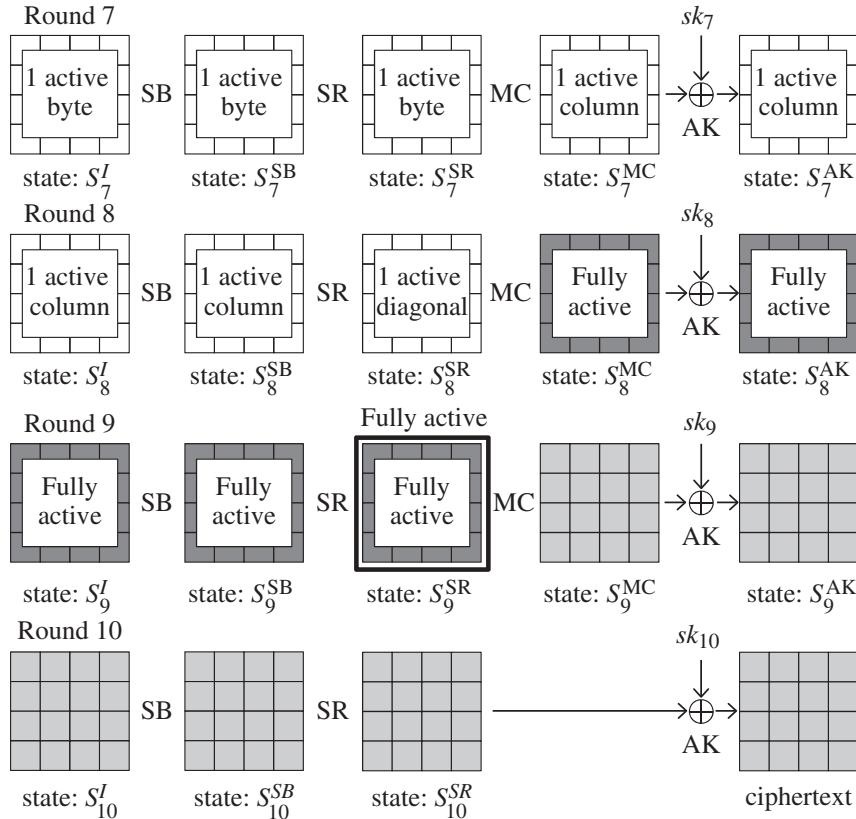
### 6.2.2 Impossible DFA with Unknown Faulty Byte Positions

Similarly to impossible differential cryptanalysis in Section 4.3, impossible DFA requires to collect several pairs. The attack assumes that several plaintexts  $P_1, P_2, \dots, P_i$  for some  $i$  are available, and each of them is encrypted twice. For each plaintext, for the first time, the attacker observes the corresponding ciphertext  $C$  without injecting fault, and achieves a pair of  $(P_i, C_i)$ . For the second time, the attacker injects 1-byte fault at the beginning of round 7, and obtains the corresponding faulty ciphertext  $C'_i$ . In summary, the attacker obtains a tuple of  $(P_i, C_i, C'_i)$  for some  $i$  generated by the fault satisfying the first fault model (faulty byte position is unknown and may change every time).

#### 6.2.2.1 Differential Characteristic

On the basis of the fault model, the differential propagation during the last four rounds of AES is shown in Figure 6.7.

The differential propagation starts from the 1-byte fault injected at state  $S_7^I$ , but its position is unknown. An important property is that for any active byte position at state  $S_7^I$ , all bytes become active, that is, fully active, after two rounds. (The detailed analysis was given

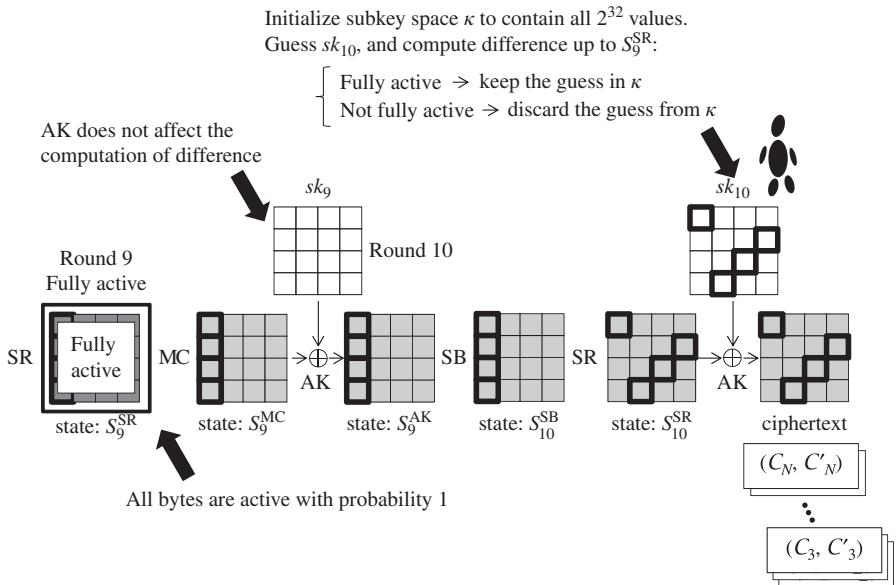


**Figure 6.7** Differential propagation for impossible DFA

in Section 4.3, and thus omitted here.) Such a probability 1 property is important for impossible differential cryptanalysis. In Figure 6.7, the fully active states are marked by dark gray. The fully active state will be broken after the MixColumns operation in round 9 with a relatively low probability. The probability that a byte becomes inactive is about  $2^{-8}$  for each byte. Hence, the ciphertext may not be fully active. In Figure 6.7, the bytes that can be either active or inactive are marked by light gray.

### 6.2.2.2 Mechanism of Reducing Subkey Space

The mechanism of reducing the subkey space of the last subkey  $sk_{10}$  is principally the same as impossible differential cryptanalysis for theoretical cryptanalysis. The attacker applies the partial decryption for the ciphertext pair by guessing the last subkey  $sk_{10}$ , and obtains the corresponding difference at state  $S_9^{SR}$ . For wrong guesses, the result of the partial decryption behaves randomly, and thus some bytes at  $S_9^{SR}$  will have no difference. This contradicts the differential propagation depicted in Figure 6.7. Hence, the guess is detected to be wrong and can be discarded from the subkey space. With several pairs of ciphertexts  $C_i, C'_i$ , the



**Figure 6.8** Key recovery mechanism of impossible DFA

attacker iterates the analysis until only one value remains in the subkey space. The mechanism of impossible DFA is described in Figure 6.8.

### 6.2.2.3 Key Recovery Procedure

As depicted in Figure 6.8, the subkey value of  $sk_{10}$  is recovered for each of the diagonally located 4 bytes. The same procedure to recover 4 bytes can be applied in parallel to recover the other 12 bytes. In the following, to be consistent with Figure 6.8, the procedure to recover  $sk_{10}[0, 7, 10, 13]$  is explained.

In a simple method, the attacker first collects several pairs of correct and faulty ciphertexts. Let  $N$  be the number of collected pairs. Then, the ciphertext pairs are denoted by  $(C_1, C'_1), (C_2, C'_2), \dots, (C_N, C'_N)$ .

1. Subkey space  $\kappa$  for  $sk_{10}[0, 7, 10, 13]$  is initialized to all the possible  $2^{32}$  values.
2. For each of  $(C_i, C'_i)$ , the attacker exhaustively guesses the subkey value of  $sk_{10}[0, 7, 10, 13]$  and computes the corresponding difference at state  $S_9^{\text{SR}}$ .
  - (a) If all bytes are active, do nothing (keep the guess in  $\kappa$ ).
  - (b) If at least one byte is inactive, discard the guess from  $\kappa$ .
3. Repeat the above until the subkey space  $\kappa$  becomes 1.

The above-mentioned procedure requires the computational cost of  $N \times 2^{32}$  inverse round function computations. The attack works, but has a room to be improved owing to step 2a,

**Algorithm 6.5** Construction of Target Differences (Look-Up Table  $T$ )

---

**Input:** Three byte variables  $X_0, X_1$ , and  $X_2$   
**Output:** A look-up table  $T$  containing  $2^{26}$  target differences for  $\Delta S_9^{\text{AK}}$

- 1: Initialize  $T$  to empty;
- 2: **for**  $X_0 = 1, 2, \dots, 255$  **do**
- 3:   **for**  $X_1 = 1, 2, \dots, 255$  **do**
- 4:     **for**  $X_2 = 1, 2, \dots, 255$  **do**
- 5:       Compute  $\text{MC}(0 \| X_2 \| X_1 \| X_0)$  and store the result in  $T$ ;
- 6:       Compute  $\text{MC}(X_2 \| 0 \| X_1 \| X_0)$  and store the result in  $T$ ;
- 7:       Compute  $\text{MC}(X_2 \| X_1 \| 0 \| X_0)$  and store the result in  $T$ ;
- 8:       Compute  $\text{MC}(X_2 \| X_1 \| X_0 \| 0)$  and store the result in  $T$ ;
- 9:     **end for**
- 10:   **end for**
- 11: **end for**

---

which does nothing as a result of some computation. This part can be optimized so that ineffective computations can be avoided. In short, the attacker chooses contradictory difference at state  $S_9^{\text{SR}}$  without guessing the subkey values, and then derive the corresponding internal state values through the SubBytes computation in round 10 to derive the corresponding wrong subkey.

In the optimized procedure, the attacker first prepares the look-up table  $T$  of  $2^{26}$  target differences for  $\Delta S_9^{\text{AK}}$ . The construction of  $T$  starts from choosing  $2^{26}$  impossible difference at  $S_9^{\text{SR}}[0, 1, 2, 3]$ . Namely, all the differences with at least one inactive byte are collected. When the byte position 0 is chosen to be inactive and the other three bytes are active, there are  $255^3 \approx 2^{24}$  ways to choose the active 3-byte differences. Similarly, roughly  $2^{24}$  differences are obtained for the cases that the inactive-byte position is set to byte positions 1, 2, and 3. In total,  $2^{26}$  impossible differences are chosen. Actually, there are  $\binom{4}{2} \cdot 255^2$  differences for two inactive-byte patterns and  $\binom{4}{1} \cdot 255$  differences for three inactive-byte patterns. Because they only have a small factor, those differential patterns are ignored here for simplifying the description. For the  $2^{26}$  impossible differences for  $S_9^{\text{SR}}[0, 1, 2, 3]$ , the corresponding difference at  $S_9^{\text{AK}}[0, 1, 2, 3]$  can uniquely be computed by linearly propagating the differences. Those are stored in the look-up table  $T$ . In detail,  $T$  is constructed by Algorithm 6.5.

Using the  $2^{26}$  target differences in the look-up table  $T$ , only the wrong subkey guesses are obtained for each correct and faulty ciphertexts pair  $(C_i, C'_i)$ . From the  $\Delta C[0, 7, 10, 13]$ , the corresponding  $\Delta S_{10}^{\text{SB}}[0, 1, 2, 3]$  can be uniquely computed owing to the linearity of the operations. Then, for each of the target difference in  $T$ , the corresponding internal state values are obtained by looking up the DDT of the S-box. Each S-box has about two solutions satisfying the given input and output differences with probability about  $2^{-1}$ . After trying  $2^{26}$  target difference for  $\Delta S_9^{\text{AK}}[0, 1, 2, 3]$ ,  $2^{26} \cdot (2^{-1})^4 = 2^{22}$  target differences have solutions for all the 4 bytes, and the number of obtained solutions is  $2^{22} \cdot 2^4 = 2^{26}$ . In the end,  $2^{26}$  wrong key suggestions are obtained for each pair of  $(C_i, C'_i)$ .

Let  $N$  be the number of collected ciphertext pairs, which are determined later. The optimized attack procedure is summarized in Algorithm 6.6.

**Algorithm 6.6** Optimized Subkey Recovery Procedure of Impossible DFA

---

**Input:**  $2^{26}$  target differences stored in  $T$ ,  $N$  ciphertext pairs  $(C_i, C'_i)$  where  $i = 1, 2, \dots, N$ ;  
**Output:** Correct value for  $sk_{10}[0, 7, 10, 13]$ ;

- 1: Initialize subkey space  $\kappa$  for  $sk_{10}[0, 7, 10, 13]$  to all the possible  $2^{32}$  values;
- 2: **for**  $i = 1, 2, \dots, N$  **do**
- 3:   Set  $\Delta S_{10}^{\text{SB}}[0, 1, 2, 3] \leftarrow \Delta C_i[0, 13, 10, 7]$ ;
- 4:   **for**  $2^{26}$  target differences in the look-up table  $T$  **do**
- 5:     **for**  $z = 0, 1, 2, 3$  **do**
- 6:       Look up DDT to find solutions of  $\Delta S_9^{\text{AK}}[z] \xrightarrow{\text{S-box}} \Delta S_{10}^{\text{SB}}[z]$ ;
- 7:     **end for**
- 8:     **if** Solutions exist for all of  $z = 0, 1, 2, 3$  **then**
- 9:       **for** All combinations of the solutions in 4 bytes **do**
- 10:         Fix 4-byte value of  $S_{10}^{\text{SB}}[0, 1, 2, 3]$  to the solution;
- 11:         Compute  $sk_{10}[0, 7, 10, 13] \leftarrow S_{10}^{\text{SB}}[0, 3, 2, 1] \oplus C_i[0, 7, 10, 13]$ ;
- 12:         Discard  $sk_{10}[0, 7, 10, 13]$  from  $\kappa$ ;
- 13:       **end for**
- 14:     **end if**
- 15:   **end for**
- 16: **end for**
- 17: **return** the remaining value in  $\kappa$ ;

---

**6.2.2.4 Evaluation of the Number of Ciphertext Pairs**

In the following, the number of ciphertext pairs necessary to reduce the subkey space to 1 is evaluated. Because each pair generates  $2^{26}$  wrong subkey suggestions, by analyzing  $2^6$  pairs,  $2^{32}$  wrong subkey suggestions are obtained. However, different ciphertext pairs may derive identical wrong key suggestions. Considering the overlap, more than  $2^6$  ciphertext pairs are necessary. The analysis of the number of the necessary pairs is the same as impossible differential cryptanalysis in Section 4.3.

At the initial stage, the size of the remaining subkey space is  $2^{32}$ . By deriving one wrong key suggestion, the size of the remaining subkey space becomes

$$2^{32} \cdot \left(1 - \frac{1}{2^{32}}\right). \quad (6.7)$$

After discarding  $r$  wrong key suggestions, the size of the remaining subkey space becomes

$$2^{32} \cdot \left(1 - \frac{1}{2^{32}}\right)^r. \quad (6.8)$$

When the number of wrong key suggestions is  $2^{32} \cdot r$ , the size of the remaining subkey space becomes

$$2^{32} \cdot \left(1 - \frac{1}{2^{32}}\right)^{2^{32} \cdot r} \approx 2^{32} \cdot \left(\frac{1}{e}\right)^r. \quad (6.9)$$

For  $r = 32$ , Equation (6.9) becomes  $2^{-14.16}$ . Therefore, the size of the subkey space is reduced to 1 by obtaining  $2^{32} \cdot 32 = 2^{37}$  wrong key suggestions. The suitable choice of the number of

ciphertexts pairs,  $N$ , can be obtained by the following inequation:

$$N \times 2^{26} \geq 2^{37}, \quad (6.10)$$

which concludes that  $N = 2^{11}$  is the necessary number of correct and faulty ciphertext pairs.

### 6.2.2.5 Complexity Evaluation

Algorithm 6.5 firstly runs before the attack starts. Its computational cost is  $2^{26}$  round function computations and its memory requirement is  $2^{26}$  4-byte values. Because it is performed offline, the data complexity is 0 and the number of fault injection is 0 at this stage. In any type of the attack complexity, Algorithm 6.5 requires much smaller cost than Algorithm 6.6.

In Algorithm 6.6, initializing the subkey space  $\kappa$  requires a memory to store  $2^{32}$  4-byte values. The computational complexity inside the loop of step 4 is  $2^{11} \cdot 2^{26} = 2^{37}$  one-round operations for a single column, which would require about 1/4 cost of the round function. Then, the column-wise operations of Algorithm 6.6 are iterated four times for all the columns, which makes the entire computational cost  $2^{37}$  round function operations.

$2^{11}$  different plaintexts  $P_i$  must be processed twice to obtain the corresponding  $C_i$  and  $C'_i$ . Thus, the data complexity is  $2^{11} \cdot 2 = 2^{12}$  plaintexts. The number of fault injection is  $2^{11}$ .

The attack complexity is summarized as follows:

$$(Data, Time, Memory, \#Faults) = (2^{12}, 2^{37}, 2^{32}, 2^{11}). \quad (6.11)$$

The attack complexity is feasible, which meets the requirement for side-channel analysis and fault analysis.

**Exercise 6.3** *In theoretical impossible differential cryptanalysis in Chapter 4, 3.5-round impossible differential characteristics can take multiple active bytes as input. However, impossible DFA requires to have only one faulty byte. Explain the reason of the difference between two attacks.*

### 6.2.2.6 Memory-Efficient Attack Variant

The impossible DFA starts with preparing the subkey space  $\kappa$  containing  $2^{32}$  values. It seems to indicate that the attack cannot work without a memory to store  $2^{32}$  4-byte values. However, the impossible DFA can recover the subkey with a much less memory requirement. This variant of the attack requires a memory only for collecting correct and faulty ciphertext pairs. Hence, it only requires  $2 \cdot 2^{11} = 2^{12}$  ciphertexts, which is equivalent to  $2^{16} = 65536$  bytes, although it requires a little bit more computational cost than Algorithm 6.6.

The attack procedure is simple. Suppose that the attacker collects  $2^{11}$  correct and faulty ciphertext pairs  $(C_i, C'_i)$  for  $i = 1, 2, \dots, 2^{11}$ . The attacker guesses the 4 bytes of  $sk_{10}[0, 7, 10, 13]$  exhaustively, and then computes the corresponding difference at  $S_9^{\text{SR}}$  for each ciphertext pair. If the guess is wrong, the result will have inactive bytes for at least a pair. If the guess is correct, the result is always fully active for all the  $2^{11}$  pairs. Thus, the correct

**Algorithm 6.7** Memory Efficient Attack Variant of Impossible DFA

---

**Input:**  $2^{11}$  ciphertext pairs  $(C_i, C'_i)$  where  $i = 1, 2, \dots, 2^{11}$ ;  
**Output:** Correct value for  $sk_{10}[0, 7, 10, 13]$ ;

- 1: **for**  $sk_{10}[0, 7, 10, 13] = 0, 1, \dots, 2^{32} - 1$  **do**
- 2:   **for**  $i = 1, 2, \dots, 2^{11}$  **do**
- 3:      $\text{tmp1} \leftarrow MC^{-1} \circ SB^{-1}(sk_{10}[0, 13, 10, 7] \oplus C_i[0, 13, 10, 7])$ ;
- 4:      $\text{tmp2} \leftarrow MC^{-1} \circ SB^{-1}(sk_{10}[0, 13, 10, 7] \oplus C'_i[0, 13, 10, 7])$ ;
- 5:     **if** ( $\text{tmp1} \oplus \text{tmp2}$ ) contains an inactive byte **then**
- 6:       Discard  $sk_{10}[0, 7, 10, 13]$ ;
- 7:     **end if**
- 8:   **end for**
- 9:   **if** ( $\text{tmp1} \oplus \text{tmp2}$ ) are fully active for all the  $2^{11}$  pairs **then**
- 10:     **return**  $sk_{10}[0, 7, 10, 13]$ ;
- 11:   **end if**
- 12: **end for**

---

subkey values are obtained. The attack procedure for the memory-efficient attack variant is given in Algorithm 6.7.

Algorithm 6.7 requires the same data complexity as Algorithm 6.6. The memory requirement is reduced to  $2^{12}$  for storing correct and faulty ciphertext pairs. The computational cost increases to  $2^{32} \cdot 2^{11} \cdot 2 = 2^{44}$  round function computations. Note that Algorithm 6.5 cannot be performed with the limited memory requirement. In summary, the attack complexity of the memory-efficient attack variant is as follows:

$$(Data, Time, Memory, \#Faults) = (2^{12}, 2^{44}, 2^{12}, 2^{11}). \quad (6.12)$$

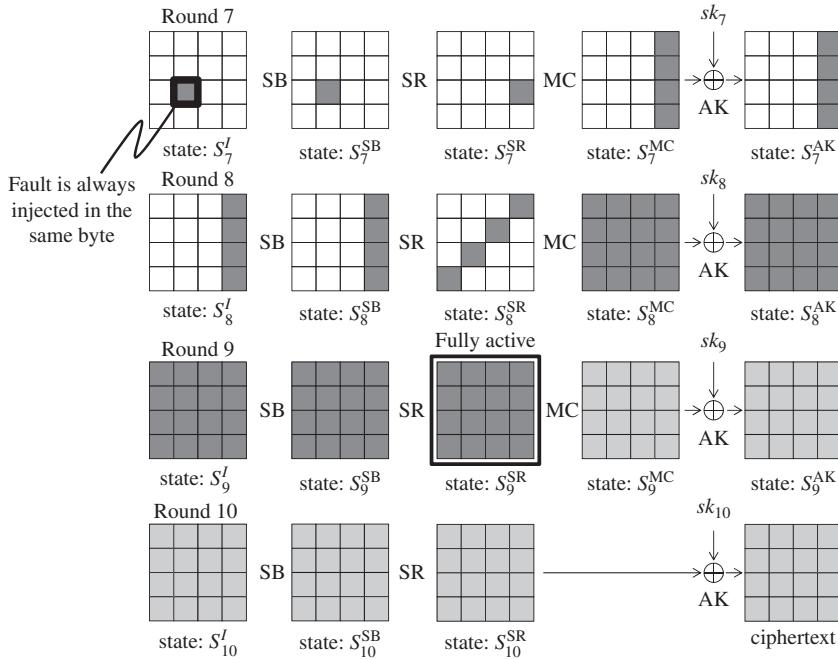
### 6.2.3 Impossible DFA with Fixed Faulty Byte Position

Suppose that the attacker can target an identical byte position for every fault injection trials. The assumption of such a stronger attacker's ability enables to improve impossible DFA.

The attack assumes that an identical plaintext  $P$  is encrypted many times. For the first time, the attacker observes the corresponding ciphertext without injecting fault. Let  $C_1$  be the correct ciphertext. For the second time or later, the attacker injects 1-byte fault at the beginning of round 7 in the same byte position. The faulty byte position is not necessarily to be known to the attacker as long as the position is fixed. It is assumed that the injected fault value is determined accordingly to the uniform distribution. Let  $C_i$  be the obtained faulty ciphertext with the  $i$ th fault injection. In summary, after the encryption of the plaintext  $i$  times and fault injection  $i - 1$  times, the attacker obtains  $i$  ciphertexts  $(C_1, C_2, \dots, C_i)$ . Note that the attack in this fault model does not distinguish whether the obtain ciphertext is correct or faulty. The differential propagation in this fault model is described in Figure 6.9.

#### 6.2.3.1 Reducing Data Complexity

The core mechanism and the subkey recovery procedure of this attack is exactly the same as the one in the previous fault model. The main improvement that can be utilized in this fault



**Figure 6.9** Differential propagation for impossible DFA with fixed faulty byte position. The figure describes the case in which the byte  $S_7^I[6]$  has a fault. The attack can work for any byte position as long as the faulty byte position is fixed. Moreover, the attacker does not have to know the faulty byte position as long as it is fixed.

model is that any of the two ciphertexts among  $(C_1, C_2, \dots, C_i)$  can be used to derive wrong subkey suggestions. To be more precise, from  $i$  distinct ciphertexts,

$$\binom{i}{2} = \frac{i^2 - i}{2}$$

ciphertext pairs are constructed, and all of the  $(i^2 - i)/2$  pairs can suggest the wrong subkey suggestions.

Recall that impossible DFA requires to collect  $N = 2^{11}$  ciphertext pairs in the previous fault model in Equation (6.10). The data complexity  $i$  and the number of fault injections  $i - 1$  in this model can be computed as

$$\binom{i}{2} \geq 2^{11}.$$

By setting  $i = 64 (= 2^6)$ ,  $\binom{i}{2} = 2016 \approx 2^{11}$ . In the end, the attack complexity can be reduced as follows:

$$(Data, Time, Memory, \#Faults) = (2^6, 2^{37}, 2^{32}, 2^6 - 1). \quad (6.13)$$

### 6.3 Integral Differential Fault Analysis

Similarly to impossible DFA, the motivation is to attack the AES implementation in which the last three rounds of the AES encryption are protected against the fault injection. Thus,

the fault can be injected only in the 7th round or earlier during the encryption process. This section explains that the integral cryptanalysis enables the attacker to recover the key with fault injected at the beginning of the 7th round. Hereafter, the combination of DFA with integral cryptanalysis is called **integral DFA**.

Integral DFA has been improved several times. The first integral DFA assumes the bit-fault model, in which the attacker can flip any bit of the internal state with probability 1. The fault model is very strong if the real environment is considered. Then, the bit-fault model was relaxed to the random byte model with more sophisticated analysis of the key recovery mechanism. Finally, the fault model in integral DFA was further relaxed to accept the noise during fault injection trials. Integral DFA can recover the key even if the fault injection can succeed only probabilistically.

### 6.3.1 Fault Model

Because the last three rounds are protected, the attacker injects the fault at the beginning of the 7th round  $S_7^I$ . Three types of fault model have been considered depending on the progress of the attack theory. The first fault model is the so-called **bit-fault model**, explained as follows:

*1 bit of fault is injected at the beginning of the 7th round of AES-128. The attacker can choose the target bit and the value of the faulty bit is flipped owing to the fault.*

The second fault model is a **random byte-fault model** on the fixed unknown byte position, which is the same as the second fault model in impossible DFA:

*1 byte of fault is injected at the beginning of the 7th round of AES-128. The faulty byte position is unknown to attackers. The faulty value, either. Every time the fault is injected, the attacker can cause the fault in the same but unknown byte position. Every time the fault is injected, the (unknown) fault value is assumed to be determined accordingly to the uniform distribution for 1-byte values.*

The third fault model is a **noisy random byte-fault model** on the fixed unknown byte position, explained as follows:

*1 byte of fault is intended to be injected at the beginning of the 7th round of AES-128. The attacker intends to target an identical byte position for every fault injection trials. The faulty byte position is unknown to attackers. The faulty value, either. The fault injecting attempt may fail with some probability. The attacker cannot know whether or not the intended fault is injected. Every time the intended fault is injected, the (unknown) fault value is assumed to be determined accordingly to the uniform distribution for 1-byte values.*

The assumption for the attacker's ability is getting more and more relaxed. In the noisy random fault model, the attacker obtains a set of ciphertexts, in which some of them are derived by the intended fault and the others are completely random noise. The attacker cannot distinguish which of the collected ciphertexts are the intended ones, but still needs to recover the key.

### 6.3.2 Integral DFA with Bit-Fault Model

Recall integral cryptanalysis in Section 4.4, in particular a set of 256 plaintexts in Equation (4.160). The attack requires to collect a set of 256 intermediate values in which only 1 byte of the state takes all the 256 possible values and the other 15 bytes are fixed to the unique value among 256 intermediate values.

The attack assumes that an identical plaintext  $P$  is encrypted at least 256 times under the fixed key. For the first time, the attacker observes the corresponding ciphertext  $C_0$  without injecting fault. For the second time, the attacker injects 1-bit fault to the least significant bit of the target byte so that the second state value has difference 1 compared to the original state value. The attacker obtains the corresponding ciphertext  $C_1$ . For the third time, the attacker injects 1-bit fault to the second least significant bit of the target byte so that the third state value has difference 2 compared to the original state value. The attacker obtains the corresponding ciphertext  $C_2$ . For the fourth time, the attacker injects 1-bit fault to the least significant bit and the second least significant bit of the target byte simultaneously so that the second state value has difference 3 compared to the original state value. The attacker obtains the corresponding ciphertext  $C_3$ . This procedure is iterated 255 times to collect 256 internal state values containing 256 different values on the target byte of the state at the beginning of round 7. In summary, the attacker obtains a tuple of  $(P, C_0, C_1, \dots, C_{255})$ . The attacker does not have to know the faulty byte position as long as the target faulty bit is located in the same byte position. Note that the attack in this fault model does not distinguish whether the obtain ciphertext is correct or faulty.

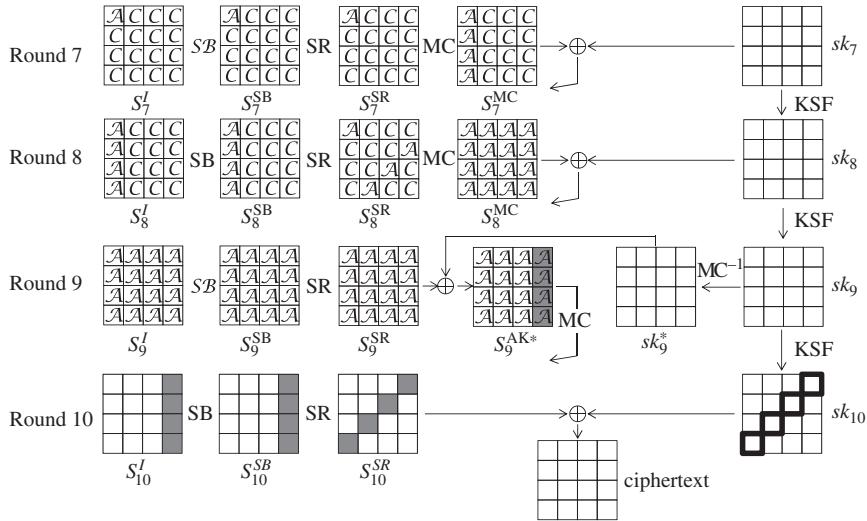
#### 6.3.2.1 Integral Property

The collected 256 intermediate state values are propagated toward the ciphertext. With the same notations as Section 4.4, the 2.5-round integral property from round 7 is shown in Figure 6.10.

The propagation of the property up to state  $S_9^{\text{SR}}$  is exactly the same as the theoretical cryptanalysis in Section 4.4. Namely, all bytes have the “all” property at  $S_9^{\text{SR}}$ . In round 9, the order of the MixColumns and AddRoundKey operations is exchanged by linearly transforming  $sk_9$ . Because the subkey XOR does not affect the integral property, the all property is maintained in all bytes at state  $S_9^{\text{AK}*}$ .

In Figure 6.10, the propagation of the integral property finishes at state  $S_9^{\text{AK}*}$ , while the theoretical cryptanalysis in Section 4.4 propagates over one more MixColumns operation. Recall Figure 4.41. After the subsequent MixColumns operation, all bytes still hold the “balanced” property. Actually, the same analysis can be applied about the propagation in integral DFA. Namely, all bytes at state  $S_{10}^I$  in Figure 6.10 satisfy the balanced property.

Why such a property is not considered in integral DFA? The problem here is that the balanced property occurs even for wrong guesses with a relatively high probability. For a wrong guess, the sum of the decryption results among 256 texts can be 0 in each byte with probability  $2^{-8}$ , which may not be enough to reduce the subkey space. In theoretical cryptanalysis, the goal of the attacker is recovering the key faster than the exhaustive search. The feasibility of the attack in practice is not an important issue. However, in side-channel analysis, the goal is recovering the key in practice, which may prefer a much stronger property than the one used in theoretical cryptanalysis.



**Figure 6.10** Integral property for integral DFA in bit fault model

### 6.3.2.2 Key Recovery Procedure

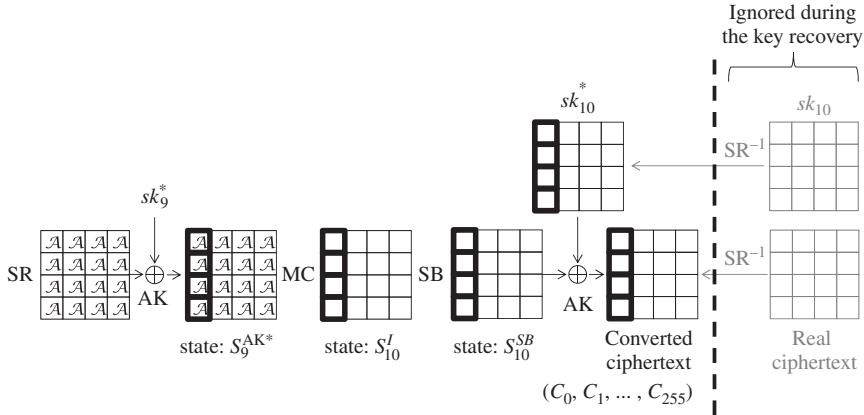
The mechanism of reducing the subkey space of the last subkey  $sk_{10}$  is similar to the theoretical integral cryptanalysis, but the subkey space can be reduced much faster. The attacker first collects 256 ciphertexts ( $C_0, C_1, \dots, C_{255}$ ), in which the corresponding internal state at 1 byte of  $S_7^I$  takes all 256 values owing to the bit fault.

In order to simplify the attack evaluation as much as possible, the order of the ShiftRows operation and AddRoundKey is exchanged by applying the linear transformation  $SR^{-1}$  to subkey  $sk_{10}$ . Then, the partial decryption from the ciphertext to state  $S_9^{AK*}$  becomes as Figure 6.11.

$sk_{10}$  is converted to  $sk_{10}^*$  by the inverse of the ShiftRows operation. Hereafter, integral DFA first aims to recover the value of  $sk_{10}^*$ . Note that if  $sk_{10}^*$  is recovered, the corresponding  $sk_{10}$  can be computed easily, and eventually the original key  $sk_0$  is recovered by computing the inverse of the key schedule function. Not only  $sk_{10}$ , but also each ciphertext is converted by the inverse of ShiftRows operation. Hereafter, the converted ciphertexts are renamed as ( $C_0, C_1, \dots, C_{255}$ ). During the key recovery procedure, the converted ciphertexts are used instead of the real ciphertexts.

Because the ShiftRows operation in round 10 is ignored now, the partial decryption from converted ciphertexts to  $S_9^{AK*}$  is done column by column. The related bytes to the analysis of the first column are stressed by bold line in Figure 6.11.

The attacker exhaustively guesses the first column of  $sk_{10}^*$ , and performs the partial decryption for 256 converted ciphertexts. Let  $S_9^{AK*}[0, 1, 2, 3]$  be the 4 byte values of  $S_9^{AK*}[0, 1, 2, 3]$  corresponding to  $C_i$ . For each guess, the attacker computes  $S_9^{AK*}[0, 1, 2, 3]$  for  $i = 0, 1, \dots, 255$  and stores the results. Then, the attacker checks if  $S_9^{AK*}[0]$  for  $i = 0, 1, \dots, 255$  covers all the 256 possibilities. Similarly, the occurrence of the same event is checked for byte positions 1, 2, and 3. Because the bit-fault model can inject the fault on a



**Figure 6.11** Key recovery procedure for integral DFA

target bit with probability 1, the 256 texts in the set always satisfy the “all” property in state  $S_9^{AK*}$ . Therefore, if the guess is correct, the results of the partial decryption up to  $S_9^{AK*}$  for 256 ciphertexts will contain all the 256 values in each byte. If the guess is wrong, the results of the partial decryption show a random behavior. The probability that 256 texts will result in 256 different state values is very low (later evaluated in details). From this reason, the attack can recover the correct value of  $sk_{10}^*$ .

### 6.3.2.3 Evaluation of the Remaining Subkey Space

In integral DFA, if the guess of  $sk_{10}^*$  is correct, the results of the partial decryption always satisfy the “all” property at state  $S_9^{AK*}$ . Hence, false negatives never occur. On the other hand, the “all” property at state  $S_9^{AK*}$  may happen to be satisfied with a low probability for wrong guesses. Thus, it is necessary to evaluate the probability of the false positive.

Assume that the partial decryption of  $C_i$  with a wrong guess of  $sk_{10}^*$  behaves randomly and independently for different  $i$ . Then, the occurrence of false positives in a single byte of  $S_9^{AK*}$  is equivalent to the occurrence of the following event.

*Let  $\mathcal{N}$  be a byte value space, that is,  $\mathcal{N} \in \{0, 1, \dots, 255\}$ . Let “random pick up” be a procedure to randomly choose 1 element from  $\mathcal{N}$  according to the uniform distribution. The false positive in a single byte is equivalent to the event that after doing the random pick up 256 times, all of the 256 elements are chosen once.*

The random pick up trial corresponds to obtaining a result of the partial decryption in one byte for one ciphertext. The evaluation of the occurrence of this event is widely known. In the first pick up, any value can be chosen. Thus, the success probability of the first pick up is 256/256. In the second pick up, any value but for the already appeared one can be chosen. Thus, the success probability of the second pick up is 255/256. Similarly, the success probability for the

$(i + 1)$ th pick up is  $(256 - i)/256$ . In the end, the probability of the event is evaluated as

$$\prod_{i=0}^{255} \frac{(256 - i)}{256}. \quad (6.14)$$

This test is applied to 4 bytes in a column simultaneously. Thus, the probability of the false positive of this attack is

$$\left( \prod_{i=0}^{255} \frac{(256 - i)}{256} \right)^4 \approx (2^{-364})^4 = 2^{-1456} \approx 0. \quad (6.15)$$

The probability of the false positive is negligible, which indicates that only the correct guess can satisfy the “all” property at 4 bytes of state  $S_9^{\text{AK}*}$ .

#### 6.3.2.4 Attack Procedure

The attack procedure of integral DFA in the bit-fault model is described in an algorithmic form in Algorithm 6.8.

After the fist column of  $sk_{10}^*$  is recovered, the same procedure is iterated three times in order to recover the second, the third, and the fourth column of  $sk_{10}^*$ . Then,  $sk_{10}$  is computed

---

#### Algorithm 6.8 Key Recovery Procedure of Integral DFA in Bit Fault Model

---

**Input:** 256 ciphertexts  $C_0, C_1, \dots, C_{255}$

**Output:**  $sk_{10}^*[0, 1, 2, 3]$

```

1: for  $i = 0, 1, \dots, 255$  do
2:    $C_i \leftarrow \text{SR}^{-1}(C_i);$ 
3: end for
4: for  $sk_{10}^*[0, 1, 2, 3] = 0, 1, \dots, 2^{32} - 1$  do
5:   for  $i = 0, 1, \dots, 255$  do
6:      $\text{tmp}_0 \leftarrow S^{-1}(C_i[0] \oplus sk_{10}^*[0]);$ 
7:      $\text{tmp}_1 \leftarrow S^{-1}(C_i[1] \oplus sk_{10}^*[1]);$ 
8:      $\text{tmp}_2 \leftarrow S^{-1}(C_i[2] \oplus sk_{10}^*[2]);$ 
9:      $\text{tmp}_3 \leftarrow S^{-1}(C_i[3] \oplus sk_{10}^*[3]);$ 
10:     $S_{9,i}^{\text{AK}*}[0, 1, 2, 3] \leftarrow \text{MC}^{-1}(\text{tmp}_0, \text{tmp}_1, \text{tmp}_2, \text{tmp}_3);$ 
11: end for
12: if  $S_{9,j}^{\text{AK}*}[0]$  contains all the 256 values for  $j \in \{0, 1, \dots, 255\}$  then
13:   if  $S_{9,j}^{\text{AK}*}[1]$  contains all the 256 values for  $j \in \{0, 1, \dots, 255\}$  then
14:     if  $S_{9,j}^{\text{AK}*}[2]$  contains all the 256 values for  $j \in \{0, 1, \dots, 255\}$  then
15:       if  $S_{9,j}^{\text{AK}*}[3]$  contains all the 256 values for  $j \in \{0, 1, \dots, 255\}$  then
16:         return the current  $sk_{10}^*[0, 1, 2, 3];$ 
17:       end if
18:     end if
19:   end if
20: end if
21: end for

```

---

by SR( $sk_{10}^*$ ), and eventually  $sk_0$  is recovered by computing the inverse of the key schedule function.

### 6.3.2.5 Complexity Evaluation

Owing to the exhaustive guess of  $sk_{10}^*[0, 1, 2, 3]$  at step 4 and 256 converted ciphertexts at step 5, the computational cost of this attack is about  $2^{32} \cdot 2^8 = 2^{40}$  round function computations for one column. After iterating Algorithm 6.8 four times, the computational cost becomes  $2^{40}$  round function computations, which is equivalent to  $2^{40}/10 \approx 2^{36.7}$  AES computations. The memory complexity is only 256 state values. Regarding the data complexity, the same plaintext must be encrypted 256 times. The attacker needs to cause intended bit faults for 255 times out of 256 opportunities. In the end, the attack complexity is summarized as follows:

$$(Data, Time, Memory, \#Faults) = (2^8, 2^{36.7}, 2^8, 2^8 - 1). \quad (6.16)$$

The attack complexity is feasible, which meets the requirement for side-channel analysis and fault analysis.

### 6.3.3 Integral DFA with Random Byte-Fault Model

Similarly to the bit-fault model, the attack in the random byte-fault model injects many faults in a fixed byte at the beginning of round 7 (state  $S_7^I$ ), while the same plaintext is encrypted under the same key many times. The main difference from the bit-fault model is that the attacker cannot collect all the 256 values at a fixed byte of  $S_7^I$  because the attacker does not have an ability to control the fault bit by bit.

To solve this problem, integral DFA in a random byte-fault model introduces a similar, but different, integral property from Figure 6.10. Here, the attacker does not collect 256 texts containing all the 256 values in a target byte. Instead, the attacker collects only  $d$  texts containing distinct  $d$  values in the target byte, where  $d \leq 256$ . Note that when  $d = 256$ , the property becomes the same as Figure 6.10.

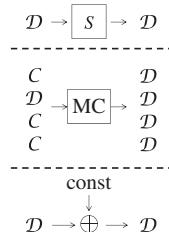
#### 6.3.3.1 “Distinct” Property $D$

Consider a set of  $d$  byte values  $x_0, x_1, \dots, x_{d-1}$  satisfying  $x_i \neq x_j$  such that  $i \neq j, 0 \leq i, j \leq d - 1$ . Such a set of byte values is defined to have **distinct property**, or in short  $D$ . The attacker tries to make  $d$  distinct state values at state  $S_7^I$  in which only one byte satisfies the “distinct” property and all the other bytes satisfy the constant property. The intuition behind is that, for a relatively small  $d$ ,  $d$  distinct fault values can be caused efficiently even with the random byte-fault model. After those  $d$  texts are obtained, the corresponding properties in several AES operations are considered.

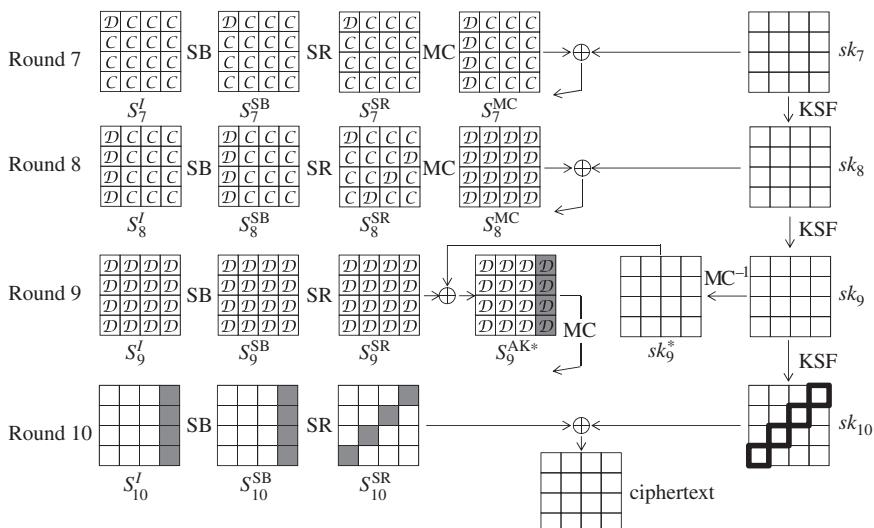
- Suppose that  $d$  distinct byte values are processed by the S-box transformation. Because S-box is a bijective mapping,  $d$  distinct input values always result in  $d$  distinct output values. Therefore, the  $D$  property never be broken by the S-box transformation.

- The ShiftRows operation only changes the byte positions in a state. It does not affect the value inside each byte. Therefore, the ShiftRows operation never breaks the  $D$  property.
- Suppose that  $d$  distinct 4-byte values consisting of 1 byte with the  $D$  property (in any position) and 3 bytes with the  $C$  property are input to the MixColumns operation. Then, all of the 4 output bytes from the MixColumns operation will have the  $d$  property. Note that this holds only when the number of byte with the  $d$  property is 1. If more than 1 byte with the  $d$  property are input to the MixColumns operation, no property can be exploited.
- Suppose that  $d$  distinct byte values are XORed with some constant byte values. Obviously, the output-byte values take  $d$  distinct byte values. Therefore, the  $D$  property is never broken by the constant addition.

Those properties are summarized in Figure 6.12. With the distinct property, the previous integral property from  $S_7^I$  in Figure 6.10 is updated for the random byte-fault model. The updated property is shown in Figure 6.13.



**Figure 6.12** Propagation of distinct property



**Figure 6.13** Integral property for integral DFA in random byte fault model

### 6.3.3.2 Key Recovery Procedure and the Number of Distinct Fault Values

The key recovery procedure basically follows the one in the bit-fault model. The attacker exhaustively guesses  $sk_{10}^*$  column by column and decrypts the collected ciphertexts up to state  $S_9^{AK*}$ . For the right guess,  $d$  distinct values appear for each of 4 bytes of  $S_9^{AK*}$  in the target column. The main difference is that only  $d$  ciphertexts are obtained instead of 256 ciphertexts. The key recovery procedure is shown in Figure 6.14. The smaller number of ciphertexts decreases the attack complexity, while the effect of reducing the key space is reduced as well. For wrong guesses, the event now occurs with a higher probability than the bit-fault model.

The probability that  $d$  distinct values can be obtained after the partial decryption of  $d$  ciphertexts for a wrong guess can be evaluated easily with Equation (6.15). Considering that the property is examined for 4 bytes in parallel, the probability of this event denoted by  $P_d$  for each of the subkey guess is

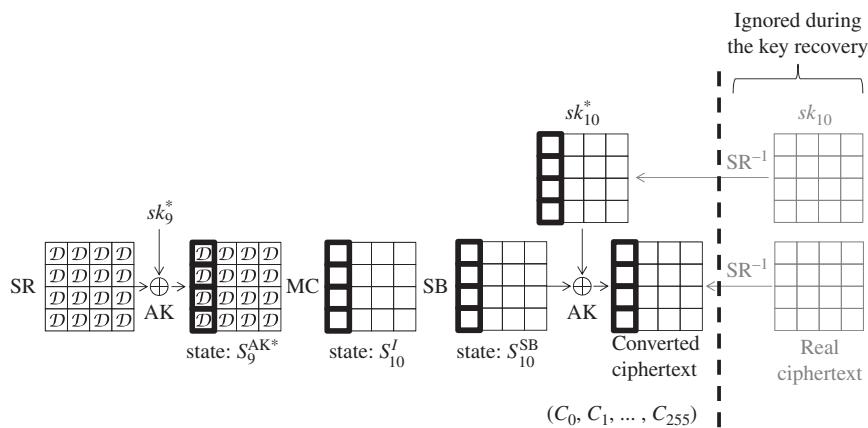
$$P_d = \left( \prod_{i=0}^{d-1} \frac{(256 - i)}{256} \right)^4. \quad (6.17)$$

If this event is sufficiently small to discard all the  $2^{32} - 1$  wrong guesses, the right key can be identified. Hence, the condition for the number of  $d$  that the attack requires is written as

$$(2^{32} - 1) \times \left( \prod_{i=0}^{d-1} \frac{(256 - i)}{256} \right)^4 < 1. \quad (6.18)$$

For  $d \geq 44$ ,  $P_d$  in Equation (6.17) becomes smaller than  $2^{-32}$ , and thus only one correct subkey can be expected.

The detailed attack procedure is almost the same as Algorithm 6.8, thus omitted here. In precise, the loops in steps 1 and 5 of Algorithm 6.8 are iterated  $d (= 44)$  times. Then, from steps 12 to 15 check if  $d$  distinct values are obtained.



**Figure 6.14** Integral property for integral DFA in random byte fault model

### 6.3.3.3 Complexity Evaluation

Owing to the column-wise exhaustive guesses of  $sk_{10}^*$  at step 4 and 44 converted ciphertexts at step 5, the computational cost of this attack is  $44 \cdot 2^{32}$ , which is less than  $2^{38}$  round function computations for one column. After iterating the attack four times, the computational cost becomes  $2^{38}$  round function computations, which is equivalent to  $2^{38}/10 \approx 2^{34.7}$  AES computations. The memory complexity is only 44 state values. Regarding the data complexity, the same plaintext must be encrypted 44 times. The attacker needs to cause intended byte faults for 43 times out of 44 opportunities. In the end, the attack complexity is summarized as follows:

$$(Data, Time, Memory, \#Faults) = (44, 2^{34.7}, 44, 43). \quad (6.19)$$

It clearly shows that the fault model is relaxed, and the attack complexity is reduced compared to the one in the bit-fault model.

### 6.3.3.4 Remarks

The “distinct” property is interesting in the sense that it is particular to fault analysis. In the theoretical cryptanalysis in Section 4.4, the attacker can choose any number of plaintexts in the chosen plaintext attack model, and thus it does not have any motivation to investigate “distinct” property.

Moreover, the theoretical cryptanalysis prefers to use the “all” property because the number of distinguished rounds can be extended by introducing the “balanced” property. The “balanced” property is not suitable for integral DFA because it can be satisfied with a relatively high probability for wrong guesses, whereas the goal of the theoretical cryptanalysis is finding shortcut attacks that can be infeasible as long as the attack cost is faster than the generic attack, or exhaustive search of the key.

Integral DFA in a random byte-fault model uses a lot of features particular to the practical analysis. In the next topic, more features particular to the practical analysis will be discussed.

**Exercise 6.4** *The mechanism of the key recovery in integral DFA with the random byte-fault model and impossible DFA with fixed faulty byte position are essentially the same.*

- (a) *Explain the reason why those two attacks are essentially the same.*
- (b) *The attack complexity of those two attacks, Equations (6.11) and (6.19), are slightly different. What does cause the difference? Which evaluation is more precise?*

### 6.3.4 Integral DFA with Noisy Random Byte-Fault Model †

Finally, the fault model is relaxed so that the fault injection succeeds only probabilistically, with some probability  $p$ . The fault injecting setting is exactly the same as the previous ones

in the bit-fault model and the random byte-fault model. While the same plaintext is encrypted under the same key many times, the attacker tries to inject the fault to a fixed byte at the beginning of round 7, or state  $S_7^I$ . Let  $n$  be the number of fault injection trials. The attacker obtains  $np$  intended faults and  $n(1 - p)$  unintended faults. Hereafter, unintended faults are called **noise**. In this attack, noise is regarded as a completely random value. Thus, after a noisy fault injection, the whole state value will become completely random at state  $S_7^I$ . As indicated by Figure 6.10 or Figure 6.13, the impact of the fault at state  $S_7^I$  will expand to the entire state until the computation reaches ciphertext. The attacker cannot detect if each obtained ciphertext is an intended one or noise. The attacker's goal is recovering the key when a set of  $n$  ciphertexts are given, in which  $np$  ciphertexts are the intended ones.

The success probability of the attack and its efficiency depends on the success probability of the fault injection, or the ratio of  $n$  and  $np$ . Hereafter, let  $\alpha$  be the number of intended ciphertexts, that is,  $\alpha = np$ .

#### 6.3.4.1 Key Recovery Mechanism Based on Coupon Collector's Problem

Remember the key recovery mechanism in the bit-fault model. With the right guess, the attacker obtains 256 distinct values only with 256 trials, while with wrong guesses, the probability of this event is too low.

Then, let us consider a slightly different situation:

The attacker obtains 256 ciphertexts and 1 noise ciphertext. Among those 257 ciphertexts, the attacker does not know which ciphertext is noise.

To recover the key, the attacker anyway partially decrypts 257 ciphertexts under the guessed subkey  $sk_{10}^*$ . Owing to the 256 intended ciphertexts, the results of the partial decryption cover all the 256 possibilities, and then 1 overlap owing to the noise. If the probability of this event is sufficiently low for wrong guesses, the attacker can recover the correct subkey value. Actually, this probability is very low. There are

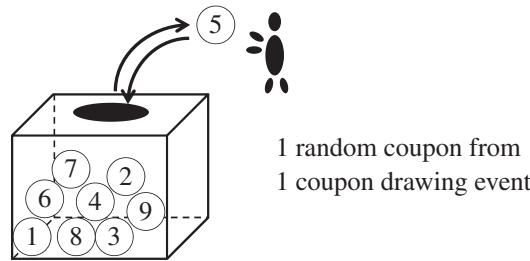
$$\binom{257}{256} = 257 \approx 2^8 \quad (6.20)$$

ways to choose 256 ciphertexts from a set of 257 ciphertexts, and for each of the 256 choices, Equation (6.15) is evaluated. Therefore, the probability of covering all the 256 values in the 4 bytes at  $S_7^I$  is  $2^8 \cdot 2^{-1456} = 2^{-1448}$ , which is small enough to filter out all the  $2^{32} - 1$  wrong guesses.

This probabilistic event is equivalent to the widely known problem called **Coupon Collector's Problem**.

**Definition 6.3.1 (Coupon Collector's Problem)** Suppose that there are  $\beta$  coupons in a box. In a coupon-drawing event, a coupon chosen accordingly to the uniform distribution is taken from the box. The coupon is returned to the box after the event. How many coupon-drawing events are expected until all  $\beta$  coupons are drawn at least once?

The coupon collector's problem is also illustrated in Figure 6.15.



**Figure 6.15** Illustration of coupon collector's problem

Considering the application to integral DFA, the parameters in the coupon collector's problem correspond to integral DFA as follows:

- The number of coupon-drawing events correspond to the number of ciphertexts (sum of the numbers of partial decryption for intended and noisy ciphertexts). When the probability of obtaining an intended fault is  $p$ ,  $256 \cdot p^{-1}$  ciphertexts must be collected to obtain the 256 intended faults. The number of coupon-drawing events is  $256 \cdot p^{-1}$ .
- $\beta$  corresponds to 256, which is the possible number of values that each byte can take.

Integral DFA checks the occurrence of this event for 4 bytes at the same time. Thus, by denoting the success probability of the coupon collector's problem by  $p_{coupon}$ , the probability that each subkey candidate is regarded as the right guess is written by  $(p_{coupon})^4$ . If  $(p_{coupon})^4$  is smaller than  $2^{32}$ , all the wrong guesses will be discarded with a good probability, and thus integral DFA can recover the key. In other words, the condition to be a valid integral DFA is that the success probability of the coupon collector's problem with  $256 \cdot p^{-1}$  events should be lower than  $2^{-8}$ .

It is also widely known that, when there are  $\beta$  coupons, the coupon collector's problem requires about

$$\beta \cdot \ln \beta \quad (6.21)$$

Coupon-drawing events to complete all the  $\beta$  coupons. When  $\beta = 256$ ,  $\beta \cdot \ln \beta \approx 1420$ . The corresponding  $p$  is evaluated as

$$\beta / (\beta \cdot \ln \beta) = 1 / \ln \beta \approx 0.18, \quad (6.22)$$

which indicates that the coupon collector's problem will succeed with a reasonably high probability when integral DFA can inject intended faults with  $p = 0.18$ . In other words, in order to recover the key with integral DFA, the probability of the fault injection should be higher than 0.18.

In general, to evaluate the number of acceptable noisy fault injections in integral DFA, a success probability of the coupon collector's problem with a given number of coupon-drawing event is needed. Before going into the evaluation, a generalization of the coupon collector's problem is considered below.

### **Generalized Coupon Collector's Problem**

The discussion in the previous section is the probabilistic fault injection for the bit-fault model, that is, the discussion for the noisy bit-fault model. In order to relax the strong assumption of the bit-fault model, the probabilistic fault injection for the random byte-fault model is needed.

The major issue here is that the attacker cannot expect to collect all the 256 values in the fixed byte at state  $S_7^I$ . The integral property is based on the “distinct” property rather than the “all” property. Remember that, in the random byte-fault model, the key recovery mechanism is based on the low probability of the event that  $d$  distinct byte values are obtained by decrypting  $d$  different ciphertexts. In the noisy fault model, the attacker obtains  $n$  texts in total, and  $\alpha = np$  of them are intended ciphertexts. Thus, to recover the key, the probability of the event that at least  $np$  distinct byte values are obtained by decrypting  $n$  different ciphertexts must be sufficiently small so that all the wrong guesses are filtered out. In the context of the coupon collector’s problem, the new event is defined as follows.

**Definition 6.3.2 (Generalized Coupon Collector’s Problem)** *Suppose that there are  $\beta$  coupons in a box. Suppose that a player wants to collect at least  $\alpha$  coupons. In the coupon-drawing event, one coupon chosen accordingly to the uniform distribution is taken from the box. The coupon is returned to the box after the event. How many coupon-drawing events are expected until  $\alpha$  coupons are drawn at least once?*

Note that the coupon collector’s problem is a subset of the generalized coupon collector’s problem. Indeed, the coupon collector’s problem is a special case of the generalized version when  $\alpha = \beta$ .

#### 6.3.4.2 Probability Evaluation of Generalized Coupon Collector’s Problem

The answer for the above-mentioned generalized coupon collector’s problem is given as a relation between the number of coupon-drawing events and the success probability. Suppose that the number of coupon-drawing events is represented as a variable  $n$ . Then, the success probability should be calculated as a function of  $\alpha$ ,  $\beta$ , and  $n$ , in which  $\alpha$  and  $\beta$  are the number of coupons to be collected and the number of all coupons, respectively. As long as the application to integral DFA on AES is discussed,  $\beta$  is fixed to 256. Here, for the sake of generality,  $\beta$  is treated as a variable. In the following, the probability that at least  $\alpha$  distinct coupons are collected after  $n$  coupon-drawing events is evaluated.

**Lemma 6.3.3** *Let  $Q(\alpha, n)$  be the number of permutations of  $n$  coupons including  $\alpha$  distinct coupons ( $1 \leq \alpha \leq n$ ). Then,  $Q(\alpha, n)$  can be represented as follows.*

$$Q(\alpha, n) = \sum_{k=1}^{\alpha} ((-1)^{\alpha-k} \binom{\alpha}{k} k^n). \quad (6.23)$$

The proof is omitted in this book. Using Lemma 6.3.3, the success probability of the generalized coupon collector’s problem for given parameters  $\alpha$ ,  $\beta$ , and  $n$  can be written in a form of the recursive equation as follows.

**Lemma 6.3.4** *Let  $P(\alpha, \beta, n)$  be the probability that one collects at least  $\alpha$  out of  $\beta$  coupons through  $n$  trials. Then, the following equation holds for  $\alpha \geq 2$ :*

$$P(\alpha, \beta, n) = \binom{\beta}{\alpha} \binom{\alpha}{1} \sum_{i=\alpha}^n \frac{Q(\alpha-1, i-1)}{\beta^i}. \quad (6.24)$$

*Proof.* Let us consider the probability that  $\alpha$  coupons out of  $\beta$  choices are collected exactly at the  $i$ th event. This probability is equivalent to the one that  $\alpha - 1$  coupons are collected through the  $i - 1$  events, which is written as

$$\frac{Q(\alpha - 1, i - 1) \binom{\beta}{\alpha - 1} \binom{\beta - \alpha + 1}{1}}{\beta^i}. \quad (6.25)$$

Here, consider the following two equations

$$\binom{\beta}{\alpha} = \frac{\beta \times (\beta - 1) \times (\beta - 2) \times \cdots \times (\beta - (\alpha - 2)) \times (\beta - (\alpha - 1))}{\alpha \times (\alpha - 1) \times (\alpha - 2) \times \cdots \times 1} \quad (6.26)$$

and

$$\binom{\beta}{\alpha - 1} = \frac{\beta \times (\beta - 1) \times (\beta - 2) \times \cdots \times (\beta - (\alpha - 2))}{(\alpha - 1) \times (\alpha - 2) \times \cdots \times 1}. \quad (6.27)$$

Then,  $\binom{\beta}{\alpha}$  can be represented with  $\binom{\beta}{\alpha - 1}$  as follows

$$\binom{\beta}{\alpha} = \binom{\beta}{\alpha - 1} \cdot \frac{\alpha}{\beta - \alpha + 1}. \quad (6.28)$$

Finally, Equation (6.25) is converted into

$$\binom{\beta}{\alpha} \binom{\alpha}{1} \frac{Q(\alpha - 1, i - 1)}{\beta^i}. \quad (6.29)$$

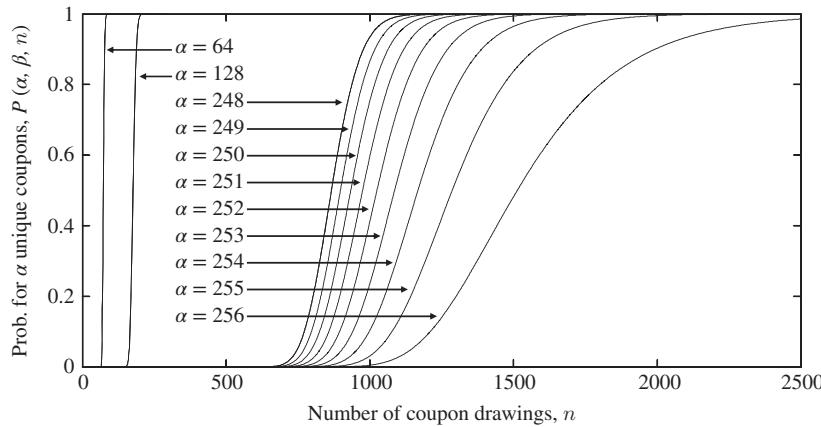
The probability  $P(\alpha, \beta, n)$  can be computed by considering all the timing that  $\alpha$  coupons are completed. Hence, the probability of  $P(\alpha, \beta, n)$  equals the summation of the above-mentioned probabilities for  $i = \alpha, \alpha + 1, \dots, n$ , which derives Equation (6.24).  $\square$

The evaluation results of  $P(\alpha, \beta, n)$  for various  $\alpha$  and  $n$  are given in Figure 6.16. Considering the application to integral DFA on AES,  $\beta$  is fixed to 256. The parameter with a low probability indicates that wrong key values can be discarded efficiently with integral DFA. For example, for  $\alpha = 256$ , even if the attack requires 1500 fault injections (a relatively large number of noise), the key space can be reduced by 1 bit for each byte. On the other hand, when only  $\alpha = 64$  distinct values can be obtained from the fault injection, only a few number of noisy fault injections spoil the attack.

### 6.3.4.3 Subkey Recovery Procedure

How the key space of AES-128 is reduced with the noisy random byte-fault model is explained here.  $\beta$  is fixed to 256. Then, the attack depends on the parameters  $(\alpha, n)$ . In practice, the suitable choice of  $(\alpha, n)$  should be determined depending on the attacker's ability. For example, if the attack can spend a lot of cost for the fault injection, the attacker may be able to inject various fault values in the target byte at the beginning of round 7, and the success probability of injecting the intended fault can be high. Namely,  $\alpha$  can be big and  $p$  is close to 1.

In the following explanation, the parameters are set to  $(\alpha, \beta, n) = (256, 256, 1440)$ , which indicates that the corresponding probability of obtaining the intended fault,  $p$ , is 0.18 as evaluated in Equation (6.22). From Equation (6.24), the corresponding probability is calculated as  $P(\alpha, \beta, n) = 1/2$ . In this parameter, the key space is gradually reduced by analyzing eight



**Figure 6.16** Probability evaluation of generalized coupon collector's problem

data sets, and thus the behavior of the attack can be explained clearly. Note that the attack can work for various parameters in general.

### Collecting Ciphertexts

1. The attacker obtains a pair of plaintext and ciphertext denoted by  $P^{(1)}, C_0^{(1)}$  without injecting the fault, and stores it in a list.
2. While the same plaintext  $P^{(1)}$  is processed, the attacker tries to inject a fault in the target fixed byte position at the beginning of round 7. If the obtained ciphertext does not overlap with the already stored ciphertexts in a table, add it to the table.
3. Repeat the second step until  $n$  ciphertexts  $C_0^{(1)}, C_1^{(1)}, \dots, C_{n-1}^{(1)}$  are obtained.

For one plaintext, a set of  $n$  ciphertexts is constructed. The same procedure is iterated eight times to collect eight sets of  $n$  ciphertexts by changing the plaintext to  $P^{(2)}, P^{(3)}, \dots, P^{(8)}$ . Remember that the attacker does not have to distinguish the intended ciphertexts from noisy ones.

### Detecting Correct Subkeys

In each set of  $n = 1440$  ciphertexts  $C_0, C_1, \dots, C_{1439}$ ,  $\alpha = 256$  values are included in the target fixed byte at  $S_7^I$ . The detailed procedure of the key recovery phase is as follows.

1. Compute  $C'_i \leftarrow \text{SR}^{-1}(C_i)$  for  $0 \leq i \leq 1439$ .
2. For each column, exhaustively guess the value for  $sk_{10}^*$  and compute the corresponding value of  $S_9^{\text{AK}*}$ .
3. If only less than 256 distinct values are observed in at least one byte of each column of  $S_9^{\text{AK}*}$ , discard the guessed  $sk_{10}^*$  from the key candidates.
4. Repeat the above three steps by 8 times by changing the text set for  $P^{(2)}, P^{(3)}, \dots, P^{(8)}$ .

Because  $P(\alpha, \beta, n) = 1/2$ , the probability that a wrong guess happens to collect  $\alpha = 256$  values for 4 bytes in a column is  $P(\alpha, \beta, n)^4 = 2^{-4}$ . Hence, the subkey space is reduced by

4 bits per set of 1440 ciphertexts. After iterating the analysis eight times,  $2^{32}$  subkey space is expected to be reduced to 1, that is, only the correct guess will remain. Note that the subkey space does not have to be reduced to exactly 1. By combining the exhaustive search, reducing the subkey space to a sufficiently small size is sufficient.

### ***Complexity Evaluation***

In order to collect the data, eight sets of 1440 ciphertexts are generated. Hence, the data complexity is  $8 \cdot 1440 = 11,520 \approx 2^{13.5}$  chosen plaintexts, and the number of fault injection is  $11,512 \approx 2^{13.5}$ . The computational cost is  $8 \cdot 1440 \cdot 2^{32}$  AES round function computations, which is approximately  $(8 \cdot 1440 \cdot 2^{32})/10 \approx 2^{42.2}$  AES computations. The attack requires the memory to store less than  $2^{28}$  AES states to count the remaining key space.

The small optimization of the computational cost can be applied. Once a subkey candidate is identified to be wrong, it does not have to be examined again for different sets of ciphertexts. Therefore, the complexity becomes

$$1440 \cdot (2^{32} + 2^{28} + 2^{24} + \cdots + 2^4)/10 \approx 2^{39.3} \quad (6.30)$$

AES computations. In summary, the attack complexity of the integral DFA in this fault model is as follows:

$$(Data, Time, Memory, \#Faults) = (2^{13.5}, 2^{39.3}, 2^{28}, 2^{13.5}). \quad (6.31)$$

## **6.4 Meet-in-the-Middle Fault Analysis**

The **meet-in-the-middle differential fault analysis (MitM DFA)** is another fault analysis that injects faults at the beginning of the seventh round. In short, MitM DFA recovers the last subkey  $sk_{10}$  with about 10 plaintexts,  $2^{40}$  computational cost,  $2^{40}$  amount of memory, and 10 fault injections. In particular, the MitM DFA has an advantage for a small data complexity and the number of fault injections.

In this section, the concept of the MitM attack against block cipher is first introduced. Then, its application in DFA is explained.

### ***6.4.1 Meet-in-the-Middle Attack on Block Ciphers***

#### ***6.4.1.1 Introduction of Meet-in-the-Middle Attack***

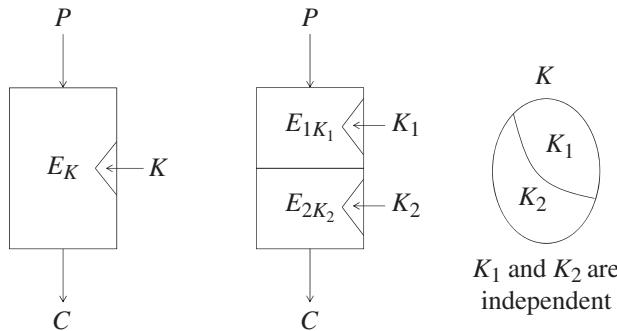
The MitM attack is a classical cryptanalytic technique against block ciphers. Let  $E_K$  be an encryption algorithm under the key  $K$ . The MitM attack can be applied efficiently when  $E_K$  can be represented as a sequence of two subencryption functions  $E_1$  and  $E_2$ , namely

$$E_K(\cdot) = E_{2K_2} \circ E_{1K_1}(\cdot), \quad (6.32)$$

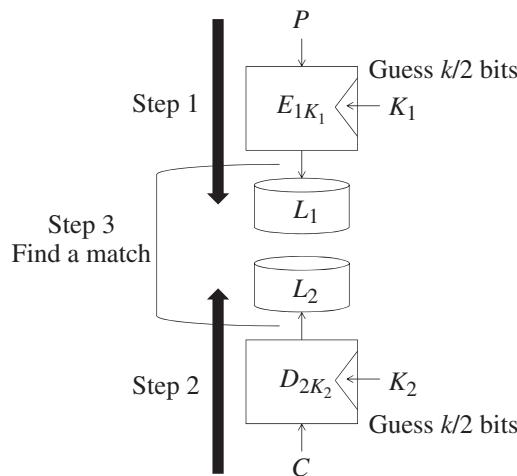
where  $K_1$  and  $K_2$  are two independent subkeys, that is,  $K_1 \cup K_2 = K$  and  $K_1 \cap K_2 = \emptyset$ . An encryption algorithm satisfying this condition is described in Figure 6.17.

#### ***6.4.1.2 Meet-in-the-Middle Attack Procedure***

Suppose that the block size,  $b$ , is bigger than the key size,  $k$ . In addition, suppose that the sizes of  $K_1$  and  $K_2$  are identical, which is  $k/2$  bits. Then, the MitM attack can recover the correct



**Figure 6.17** Target structure of meet-in-the-middle attacks



**Figure 6.18** Key recovery procedure of meet-in-the-middle attacks

$k$ -bit key  $K$  only with one plaintext and ciphertext pair,  $2^{k/2}$  computational cost, and  $2^{k/2}$  memory requirement. This is significantly faster than the exhaustive search for the  $k$ -bit key.

The attack idea is simple. The attacker first obtains a pair of plaintext and ciphertext  $(P, C)$  from the oracle in the known plaintext setting. Then, the analysis is processed as follows, which is also illustrated in Figure 6.18.

1. Exhaustively guess  $2^{k/2}$  values of  $K_1$ . For each guess, compute  $E_{1K_1}(P)$  and store the result along with  $K_1$  in a list  $L_1$ . Sort the list  $L_1$  with respect to the value of  $E_{1K_1}(P)$ .
2. Exhaustively guess  $2^{k/2}$  values of  $K_2$ . For each guess, compute  $D_{2K_2}(C)$  and store the result along with  $K_2$  in a list  $L_2$ , where  $D_{2(\cdot)}(\cdot)$  is the decryption algorithm. Sort the list  $L_2$  with respect to the value of  $D_{2K_2}(C)$ .
3. Find a match between  $2^{k/2}$  elements in  $L_1$  and  $2^{k/2}$  elements in  $L_2$ . For a matched pair, the corresponding  $K_1$  and  $K_2$  are the correct subkey values. Thus,  $K = K_1 \cup K_2$  is recovered.

### 6.4.1.3 Mechanism of Key Recovery

For the right guesses of  $K_1$  and  $K_2$ , the  $b$ -bit intermediate state values will match with probability 1. Thus, the false negative never occurs in the MitM attack. For wrong pairs of  $K_1$  and  $K_2$ , the  $b$ -bit intermediate state values in  $L_1$  and  $L_2$  will match only probabilistically, that is,  $2^{-b}$ . The number of elements in both of  $L_1$  and  $L_2$  is  $2^{k/2}$ , and thus the number of matched pairs is  $2^k$ . When  $b > k$ , an event with a probability of  $2^{-b}$  is unlikely to occur only with  $2^k$  trials.

Note that the essence of the MitM attack is the independence of  $K_1$  and  $K_2$ , which allows the attacker to compute  $E_1$  and  $D_2$  completely independently.

If the assumption  $b > k$  does not hold, the MitM attack cannot recover the correct key only with one pair of plaintext and ciphertext. Each pair provides the  $b$ -bit filtering, and thus the key space is reduced by  $b$  bits per pair. Therefore, to reduce the key space to 1, the number of necessary pairs is written as

$$\left\lceil \frac{k}{b} \right\rceil. \quad (6.33)$$

The attack procedure is shown in Algorithm 6.9.

### 6.4.1.4 Complexity Evaluation

Let us first evaluate the attack complexity of the simple case with  $b > k$  in Figure 6.18. The data complexity is one known plaintext. The time complexity of the first step is  $2^{k/2}$  evaluations of  $E_1$ . The time complexity of the second step is  $2^{k/2}$  evaluations of  $D_2$ . In total,  $2^{k/2}$  evaluations of  $E_1$  and  $E_2$ , which is equivalent to  $2^{k/2}$  evaluation of  $E$ . Note that the cost of sorting  $2^{k/2}$

---

#### Algorithm 6.9 Meet-in-the-Middle Attack for $k > b$

---

**Input:**  $(P_1, C_1), (P_2, C_2), \dots, (P_r, C_r)$ , where  $r = \lceil \frac{k}{b} \rceil$   
**Output:**  $K$

- 1: **for**  $K_1 \leftarrow 0, 1, \dots, 2^{k/2} - 1$  **do**
- 2:   **for**  $j = 1, 2, \dots, r$  **do**
- 3:     Compute  $E_{1K_1}(P_j)$ ;
- 4:   **end for**
- 5:   Store  $E_{1K_1}(P_1) \| E_{1K_1}(P_2) \| \dots \| E_{1K_1}(P_r)$  along with  $K_1$  in a list  $L_1$ ;
- 6: **end for**
- 7: Sort the list  $L_1$  with respect to the value of  $E_{1K_1}(P_1) \| E_{1K_1}(P_2) \| \dots \| E_{1K_1}(P_r)$ ;
- 8: **for**  $K_2 \leftarrow 0, 1, \dots, 2^{k/2} - 1$  **do**
- 9:   **for**  $j = 1, 2, \dots, r$  **do**
- 10:     Compute  $D_{2K_2}(C_j)$ ;
- 11:   **end for**
- 12:   Store  $D_{2K_2}(C_1) \| D_{2K_2}(C_2) \| \dots \| D_{2K_2}(C_r)$  along with  $K_2$  in a list  $L_2$ ;
- 13: **end for**
- 14: Sort the list  $L_2$  with respect to the value of  $D_{2K_2}(C_1) \| D_{2K_2}(C_2) \| \dots \| D_{2K_2}(C_r)$ ;
- 15: Find a match between  $2^{k/2}$  elements in  $L_1$  and  $L_2$ ;
- 16: Let  $K'_1$  and  $K'_2$  be the corresponding  $K_1$  and  $K_2$  for the matched pair, respectively;
- 17: **return**  $K'_1 \cup K'_2$ ;

---

data is usually assumed to be negligible compared to  $2^{k/2}$  computations of the encryption algorithm. The memory complexity is  $2^{k/2} b$ -bit internal state values for  $L_1$  and  $L_2$ , in total  $2^{k/2+1}$  state values. Here, the memory complexity for  $L_2$  can be omitted by performing the match as soon as each element for  $L_2$  is obtained. If the match is not found, that element can be discarded immediately without being stored in  $L_2$ . This reduces the memory complexity to  $2^{k/2}$  state values.

In summary, the complexity of the MitM attack for the simple case with  $b > k$  is

$$(Data, Time, Memory) = (1, 2^{k/2}, 2^{k/2}). \quad (6.34)$$

For the case with  $b < k$ , the data complexity increases to  $\lceil \frac{k}{b} \rceil$  known plaintexts. The time and data complexities also increase by a factor of  $\lceil \frac{k}{b} \rceil$  for handling multiple pairs. In summary, the complexity of the MitM attack with  $b < k$  is

$$(Data, Time, Memory) = (r, r \cdot 2^{k/2}, r \cdot 2^{k/2}), \quad (6.35)$$

where  $r = \lceil \frac{k}{b} \rceil$ .

**Exercise 6.5** Suppose that a block cipher  $E_K$  takes  $k$ -bit key,  $K$ , as input but a user wants higher security by extending the key size to  $i \times k$  bits denoted by  $K_1 \| K_2 \| \dots \| K_i$ . Multiple encryption is a well-known approach to achieve this goal, which computes ciphertext  $C$  from plaintext  $P$  as  $C \leftarrow E_{K_i} \circ \dots \circ E_{K_2} \circ E_{K_1}(P)$ . For example, triple-DES is the triple-encryption of block cipher DES. Double encryption, that is, computing  $C$  as  $E_{K_2} \circ E_{K_1}(P)$  with  $2k$ -bit key  $K_1 \| K_2$ , does not have higher security than the ordinary single encryption, that is, computing  $C$  as  $E_{K_1}(P)$  with  $k$ -bit key  $K_1$ . Explain the reason why double encryption does not have higher security.

#### 6.4.2 Meet-in-the-Middle Attack for Differential Fault Analysis

Remember that the essence of the MitM attack is the independence of  $K_1$  and  $K_2$ , which allows the attacker to compute  $E_1$  and  $D_2$  completely independently. Unfortunately, such a strong independence of subkeys is unusual for standard block ciphers. For example, any three versions of AES generate  $sk_{i+1}$  from  $sk_i$ . Thus, there is no independent subkey. However, the idea of the MitM attack can be useful when a part of the subkeys is guessed during the attack, or an attack against a small number of rounds is considered. Indeed, fault attacks meet those conditions, and thus the mechanism of the MitM attack can be exploited.

MitM DFA is essentially the same as DFA. The efficient key recovery mechanism of the MitM attack is exploited when subkeys are guessed during DFA. The straightforward differential attack requires to guess 10 subkey bytes ( $2^{80}$  guesses), which is infeasible. MitM DFA separates the guess of 10 subkey bytes to two independent procedures with five subkey-byte guess. This enables to recover the last subkey  $sk_{10}$  with about 10 plaintexts,  $2^{40}$  computational cost,  $2^{40}$  amount of memory, and 10 fault injections.

### 6.4.2.1 Fault Model

The assumed fault model in MitM DFA is exactly the same as standard DFA but for the fault injection round. The simple attack assumes as follows:

*1 byte of fault is injected at the beginning of the 7th round of AES-128. The faulty byte position is known to attackers, while the faulty value is not known to attackers.*

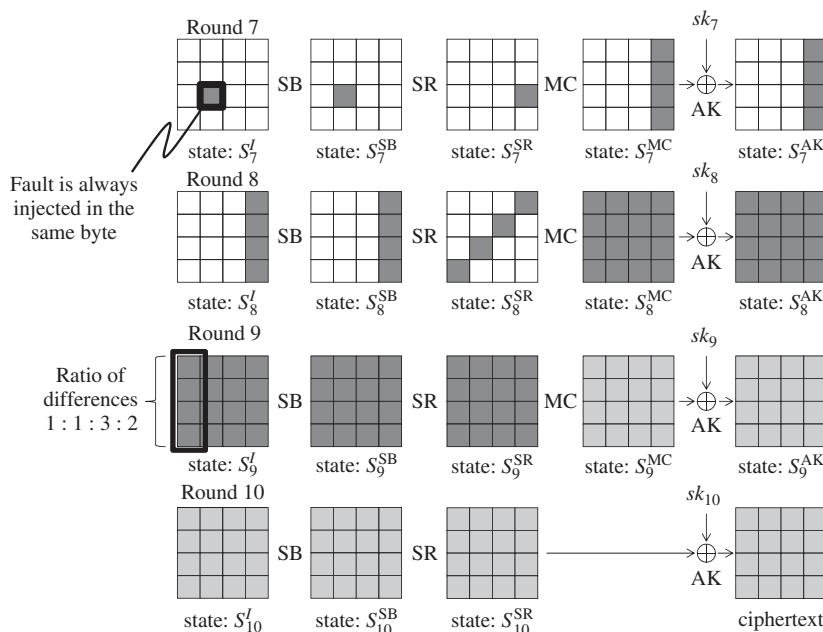
The assumption of the knowledge of the faulty byte position can be relaxed as follows:

*1 byte of fault is injected at the beginning of the 7th round of AES-128. The faulty byte position is **unknown** to attackers. The faulty value, either.*

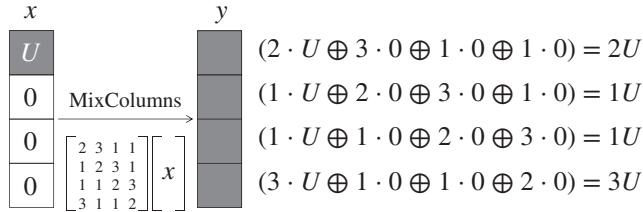
### 6.4.2.2 Differential Characteristic

For simplicity, the attack is first explained in the fault model with the knowledge of the faulty byte position. Let the faulty byte position at the beginning of the seventh round be 6. The attacker obtains a pair of plaintext and ciphertext denoted by  $(P_1, C_1)$ . Then, the attacker makes a query of  $P_1$  to the encryption oracle in the chosen plaintext model, and injects a fault in the known and fixed byte position at the beginning of round 7. The differential propagation in the subsequent rounds is described in Figure 6.19.

The fault injecting timing and the fault model are exactly the same as the ones in impossible DFA discussed at Section 6.2. Thus, the resulting differential propagation is also the same

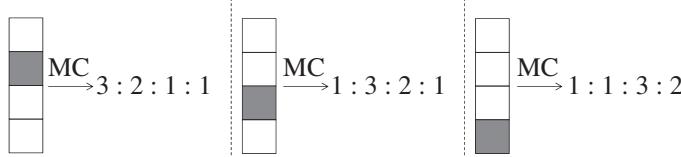


**Figure 6.19** Differential propagation in MitM DFA



The difference value  $U$  is unknown, but the ratio of 4-byte differences is fixed

4-byte differences ratio  $\rightarrow 2 : 1 : 1 : 3$



**Figure 6.20** Ratio of 4-byte differences in a column

as the one in Figure 6.9. However, MitM DFA exploits a different feature from impossible DFA. Namely, the property of having 16 active bytes at state  $S_9^{\text{SR}}$  is no longer used. The new property exploited is the ratio of the byte differences in each column at state  $S_9^I$ , in which  $\Delta S_9^I = \Delta S_8^{\text{MC}}$ . Remember that the input difference to the MixColumns operation in round 8 is unknown to the attacker, while the attacker still can compute the ratio of 4 output-byte differences.

How to compute the ratio of 4 output-byte differences is depicted in Figure 6.20. Suppose that the input column to the MixColumns operation,  $x[0], x[1], x[2], x[3]$ , only has 1 active byte in the top byte as shown in Figure 6.20. Let  $U$  be a nonzero difference in  $\Delta x[0]$ , which is unknown to the attacker. According to the specification of the MixColumns operation, the difference of the output column,  $y[0], y[1], y[2], y[3]$ , is computed as

$$\begin{bmatrix} \Delta y[0] \\ \Delta y[1] \\ \Delta y[2] \\ \Delta y[3] \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} U \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 2U \\ U \\ U \\ 3U \end{bmatrix}. \quad (6.36)$$

Although the value of  $U$  is unknown, the attacker knows that

$$\Delta y[0] : \Delta y[1] : \Delta y[2] : \Delta y[3] = 2 : 1 : 1 : 3. \quad (6.37)$$

In particular, the relation  $\Delta y[1] = \Delta y[2]$  allows the direct application of the MitM attack. That is, the attacker independently computes  $\Delta y[1]$  and  $\Delta y[2]$  with guessing subkeys, and stores those values. The correct guesses of subkeys always yield the match of  $\Delta y[1]$  and  $\Delta y[2]$ , and this much can be found efficiently in the MitM attack.

The same property can be obtained for different input active byte positions.

$$(\Delta x[0], \Delta x[1], \Delta x[2], \Delta x[3]) = (0, U, 0, 0) \xrightarrow{\text{MC}} \Delta y[0] : \Delta y[1] : \Delta y[2] : \Delta y[3] = 3 : 2 : 1 : 1, \quad (6.38)$$

$$(\Delta x[0], \Delta x[1], \Delta x[2], \Delta x[3]) = (0, 0, U, 0) \xrightarrow{\text{MC}} \Delta y[0] : \Delta y[1] : \Delta y[2] : \Delta y[3] = 1 : 3 : 2 : 1, \quad (6.39)$$

$$(\Delta x[0], \Delta x[1], \Delta x[2], \Delta x[3]) = (0, 0, 0, U) \xrightarrow{\text{MC}} \Delta y[0] : \Delta y[1] : \Delta y[2] : \Delta y[3] = 1 : 1 : 3 : 2. \quad (6.40)$$

**Exercise 6.6** Verify the ratio of  $\Delta y[0] : \Delta y[1] : \Delta y[2] : \Delta y[3]$  for the cases of  $(\Delta x[0], \Delta x[1], \Delta x[2], \Delta x[3])$  equal to  $(0, U, 0, 0)$ ,  $(0, 0, U, 0)$  and  $(0, 0, 0, U)$ .

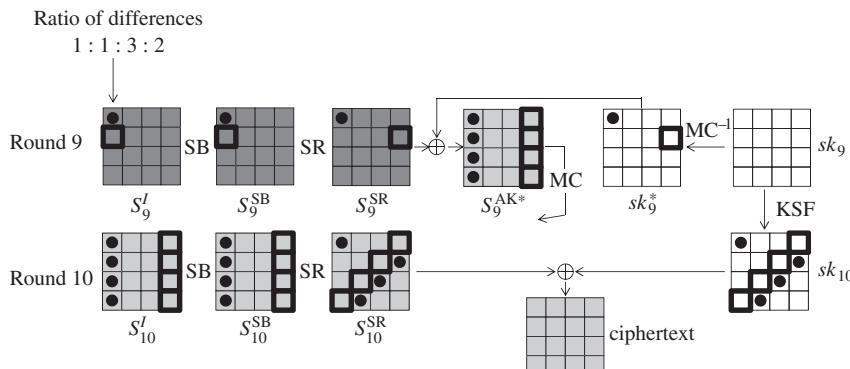
#### 6.4.2.3 Collecting Pairs

The attacker obtains a pair of plaintext and ciphertext denoted by  $(P_1, C_1)$ . While the same plaintext  $P_1$  is processed, the attacker injects a fault in the known and fixed byte position at the beginning of round 7 to obtain the faulty ciphertext  $C'_1$ . In Figure 6.19, this position is fixed to the byte position 6. The same procedure is iterated 10 times to collect  $(P_1, C_1, C'_1), (P_2, C_2, C'_2), \dots, (P_{10}, C_{10}, C'_{10})$ . In the end, the attacker obtains 10 pairs of correct and faulty ciphertexts that follow the differential propagation in Figure 6.19.

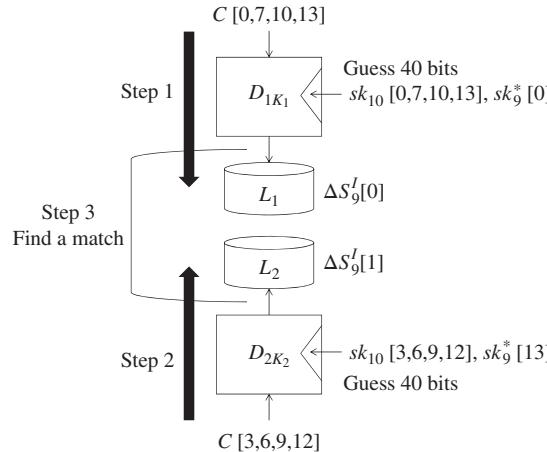
#### 6.4.2.4 Key Recovery Procedure

The attacker guesses 4 bytes of the last subkey  $sk_{10}$  and 1 byte of the converted subkey  $sk_9^*$  to perform the partial decryption from ciphertext to any 1 byte of  $\Delta S_9^I$ . Figure 6.21 describes the partial decryption from ciphertext to 1 byte of  $\Delta S_9^I[0]$  and 1 byte of  $\Delta S_9^I[1]$ . Note that the order of MixColumns and AddRoundKey operations is exchanged in round 9.

From the analysis of the differential ratio,  $\Delta S_9^I[0] = \Delta S_9^I[1]$  holds as long as the fault is injected at  $S_7^I[6]$  or the other 3 bytes in the same inverse diagonal, that is,  $S_7^I[3], S_7^I[9]$



**Figure 6.21** Independent partial decryption with 5-byte guess



**Figure 6.22** Key recovery procedure of MitM DFA

and  $S_7^I[12]$ . Hereafter, the attacker aims to compute  $\Delta S_9^I[0]$  and  $\Delta S_9^I[1]$  independently. As shown in Figure 6.21, subkeys guessed in the partial decryption for  $\Delta S_9^I[0]$  and for  $\Delta S_9^I[1]$  do not have any overlap. Thus, subkey guesses and partial decryptions can be performed independently.

1. The computation for  $\Delta S_9^I[0]$  requires 5-byte subkey guess,  $sk_{10}[0, 7, 10, 13]$  and  $sk_9^*[0]$ . The guessed bytes are marked by filled circles in Figure 6.21. For each  $2^{40}$  possibilities of the 5 subkey bytes, compute  $\Delta S_9^I[0]$  for all the 10 pairs of  $(C_j, C'_j)$ , where  $j = 1, 2, \dots, 10$ . Store the sequence of 10 bytes ( $\Delta S_9^I[0]$  for 10 pairs) in a list  $L_1$ . Then, sort the list  $L_1$ .
2. The computation for  $\Delta S_9^I[1]$  requires 5-byte subkey guess,  $sk_{10}[3, 6, 9, 12]$  and  $sk_9^*[13]$ . The guessed bytes are marked by bold line in Figure 6.21. For each  $2^{40}$  possibilities of the 5 subkey bytes, compute  $\Delta S_9^I[1]$  for all the 10 pairs of  $(C_j, C'_j)$ , where  $j = 1, 2, \dots, 10$ . Store the sequence of 10 bytes ( $\Delta S_9^I[1]$  for 10 pairs) in a list  $L_2$ . Then, sort the list  $L_2$ .
3. Find the match of 10-byte elements in  $L_1$  and  $L_2$ .

The attack procedure is also illustrated in Figure 6.22.

#### 6.4.2.5 Evaluation

Step 1 generates 8-bit value of  $\Delta S_9^I[0]$  for 10 pairs, which yields 80-bit sequences to match. After the exhaustive guess of 5 subkey bytes,  $2^{40}$  80-bit sequences are stored in  $L_1$ . Similarly after step 2,  $2^{40}$  80-bit sequences are stored in  $L_2$ . In step 3,  $2^{40} \times 2^{40} = 2^{80}$  pairs are examined for matching 80-bit values. Only one value is expected to match. Thus, the correct 10 subkey bytes are recovered.

By iterating the same attack procedure for different columns, the remaining 8 subkey bytes of  $sk_{10}$  can be recovered. Thus, all the subkey bytes in  $sk_{10}$  are recovered.

The data complexity of this attack is 20 chosen plaintexts. It requires 10 fault injections. In step 1, the computational cost is  $2^{40} \cdot 10 \cdot 2 \approx 2^{44.3}$  AES round functions, which is equivalent to  $2^{40} \cdot 10 \cdot 2 / 10 = 2^{41}$  AES computations. The memory requirement is storing  $2^{40}$  80-bit

sequence along with 5 subkey bytes, which is less than  $2^{40}$  16-byte values, thus less than  $2^{40}$  AES states. In summary, the attack complexity of the MitM DFA in this fault model is as follows:

$$(Data, Time, Memory, \#Faults) = (20, 2^{41}, 2^{40}, 10). \quad (6.41)$$

**Exercise 6.7** Write the attack procedure of MitM DFA, which recovers  $K (= sk_0)$  in the known faulty byte position, with an algorithmic form.

#### 6.4.2.6 MitM DFA with Unknown Faulty Byte Position

The feasibility of the attack with unknown faulty byte position depends on the assumption whether the unknown faulty byte position at the beginning of round 7 can be fixed or not.

If the faulty byte position can be fixed (but unknown), what the attacker needs is guessing the diagonal position with the faulty byte. The guess is chosen from at most four patterns. Thus, the computational cost of the attack becomes  $2^{43}$  rather than  $2^{41}$ , but it is still feasible.

If the faulty byte position cannot be fixed, there is no method to efficiently match the 80-bit sequences. Thus, the attack can no longer recover the key.

### Further Reading

- Derbez P, Fouque P and Leresteux D 2011 Meet-in-the-middle and impossible differential fault analysis on AES In *Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings* (ed. Preneel B and Takagi T), vol. **6917** of *Lecture Notes in Computer Science*, pp. 274–291. Springer-Verlag.
- Kim CH 2011 Efficient methods for exploiting faults induced at AES middle rounds Cryptology ePrint Archive, Report 2011/349. International Association for Cryptologic Research.
- Mukhopadhyay D 2009 An improved fault based attack of the advanced encryption standard In *Progress in Cryptology - AFRICACRYPT 2009, Second International Conference on Cryptology in Africa, Gammarth, Tunisia, June 21–25, 2009. Proceedings* (ed. Preneel B), vol. **5580** of *Lecture Notes in Computer Science*, pp. 421–434. Springer-Verlag.
- Phan RC and Yen S 2006 Amplifying side-channel attacks with techniques from block cipher cryptanalysis In *Smart Card Research and Advanced Applications, 7th IFIP WG 8.8/11.2 International Conference, CARDIS 2006, Tarragona, Spain, April 19-21, 2006, Proceedings* (ed. Domingo-Ferrer J, Posegga J and Schreckling D), vol. **3928** of *Lecture Notes in Computer Science*, pp. 135–150. Springer-Verlag.
- Piret G and Quisquater J 2003 A differential fault attack technique against SPN structures, with application to the AES and KHAZAD In *Cryptographic Hardware and Embedded Systems - CHES 2003, 5th International Workshop, Cologne, Germany, September 8-10, 2003, Proceedings* (ed. Walter CD, Koç ÇK and Paar C), vol. **2779** of *Lecture Notes in Computer Science*, pp. 77–88. Springer-Verlag.
- Sasaki Y, Li Y, Sakamoto H and Sakiyama K 2013 Coupon collector's problem for fault analysis against AES - high tolerance for noisy fault injections In *Financial Cryptography and Data Security - 17th International Conference, FC 2013, Okinawa, Japan, April 1-5, 2013, Revised Selected Papers* (ed. Sadeghi A), vol. **7859** of *Lecture Notes in Computer Science*, pp. 213–220. Springer-Verlag.
- Tunstall M, Mukhopadhyay D and Ali S 2011 Differential fault analysis of the advanced encryption standard using a single fault In *Information Security Theory and Practice. Security and Privacy of Mobile Devices in Wireless Communication - 5th IFIP WG 11.2 International Workshop, WISTP 2011, Heraklion, Crete, Greece, June 1-3, 2011. Proceedings* (ed. Ardagna CA and Zhou J), vol. **6633** of *Lecture Notes in Computer Science*, pp. 224–233. Springer-Verlag.

# 7

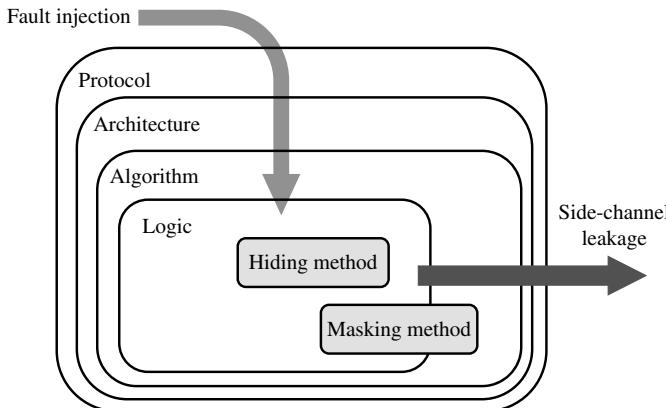
# Countermeasures against Side-Channel Analysis and Fault Analysis

In order to overcome the side-channel attacks and the fault attacks introduced in Chapters 5 and 6, this chapter explains several **countermeasures**. There exist many possible countermeasures depending on the **abstraction level** in cryptosystems. Figure 7.1 describes the hierarchy of the cryptosystem, and the position of logic-level countermeasures for hardware implementations.

The hardware device can counteract the side-channel attack if the countermeasure is appropriately implemented in the lowest abstraction level, that is, in the logic level. Such a logic-level countermeasure not only protects the logical gates but also helps to enhance the security of cryptographic algorithm and protocol against side-channel attacks, which is one of the most advantageous points of applying the countermeasure to the logic level. In general, the side-channel countermeasure tends to be more ineffective if implementing it in a higher level because higher level countermeasures cannot take care of all leakage sources that exist all over the circuits consisting of logical gates. Therefore, this chapter mainly explains the logic-level countermeasure using hiding logics and masking logics. The concept of higher level countermeasures is also explained especially focusing on the fault attack countermeasures since faults can be detected at each abstraction level, several countermeasures for the fault detection can be implemented.

## 7.1 Logic-Level Hiding Countermeasures

There are two major techniques to implement a logic-level countermeasure; **hiding logics** and **masking logics**. Here, several representative countermeasures are introduced as a case study in order to understand their resistance against the side-channel attacks.



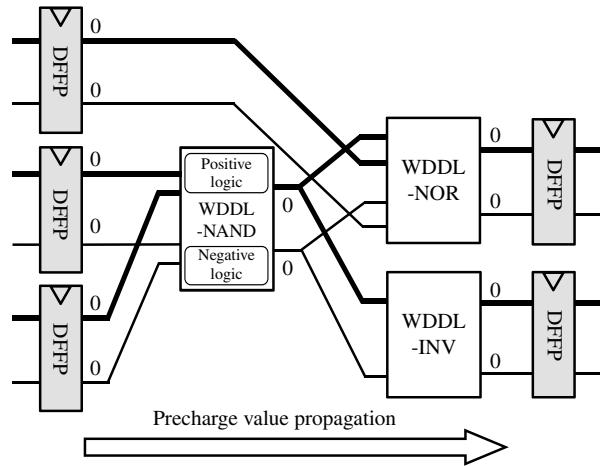
**Figure 7.1** Countermeasures for side-channel analysis and fault analysis

### 7.1.1 Overview of Hiding Countermeasure with WDDL Technique

A countermeasure using the hiding logics aims to make the side-channel information uniform. More specifically, in the case that the power consumption is used as side-channel information, the hiding logics are expected to consume constant power for any input values. This makes the attacker difficult to derive the sensitive information, for example, intermediate values during the cryptographic operations via side-channel analysis.

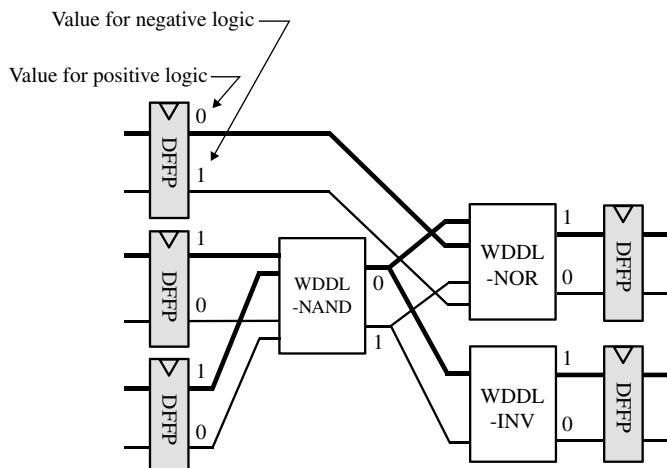
One possible solution to realize such hiding logics is to implement logical gates that consume constant power regardless of the inputs. By doing so, any combinatorial logics, which use those constant-power gates as building blocks, will consume constant power. **Wave dynamic differential logic (WDDL)** is a representative technique that follows the above concept, which was proposed by Tiri and Verbauwhede (2004).

Figure 7.2 illustrates an example circuit with the WDDL technique that is one of the dual-rail logics. The signals propagating on the bold line deal with values operated with **positive logic**, and the signals on the normal line transfers values performed with **negative logic**. One-bit signal,  $a$ , is represented with two one-bit signals as  $(a, \neg a)$  for the input and output of the WDDL gates. Since the Hamming weight of the pair,  $(a, \neg a)$ , is always 1, all the WDDL gates are balanced in terms of the number of ones and zeros in the signals. Moreover, the number of input and output signals for the WDDL gate is twice as many as the normal (single-rail) gate. As can be seen from Figure 7.2, the WDDL-NAND and WDDL-NOR gates have four inputs and two outputs, whereas the WDDL-INV gate has two inputs and two outputs. The sequential logic in the circuit named DFFP is a DFF with precharge logics. The details of DFFP are explained later in this chapter. For the purpose of initializing the circuit with the WDDL technique, DFFP outputs a special paired value,  $(0, 0)$ , called **precharge value**. The precharge value of  $(0, 0)$  propagates through the combinatorial logics since any WDDL gates output  $(0, 0)$  when the precharge values are provided as its inputs. As a result, all the signals in the combinatorial circuit are initialized to zeros. Such a circuit state is denoted by state 0 here. The state 0 will be kept during the time period called **precharge phase**. Figure 7.2 shows the signal values in the combinatorial logics in the precharge phase.

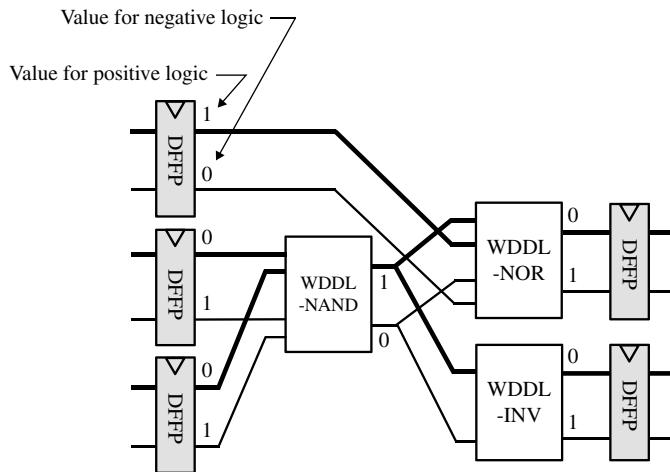


**Figure 7.2** Example circuit with WDDL technique (state 0, precharge phase)

After the precharge phase, **evaluation phase** starts. In the evaluation phase, DFFP provides the pair of  $(a, \neg a)$  that is either  $(1, 0)$  or  $(0, 1)$ . For instance, three left-side DFFPs provide  $(0, 1)$ ,  $(1, 0)$ , and  $(1, 0)$  beginning at the top as shown in Figure 7.3. In this case, the WDDL-NAND operation in the center of the figure outputs  $(0, 1)$  based on the NAND operation on two inputs  $(1, 0)$  and  $(1, 0)$ . In this way, all the signals are determined in the combinatorial logics. Let us call this state as state A. Figure 7.4 shows another result of the evaluation phase. Based on the evaluation for  $(1, 0)$ ,  $(0, 1)$ , and  $(0, 1)$  for the left-side DFFPs, the example circuit goes to state B by a state transition. Note that the number of possible states is eight in total since the left-side DFFPs can provide eight different paired values.



**Figure 7.3** Example circuit with WDDL technique (state A, evaluation phase)



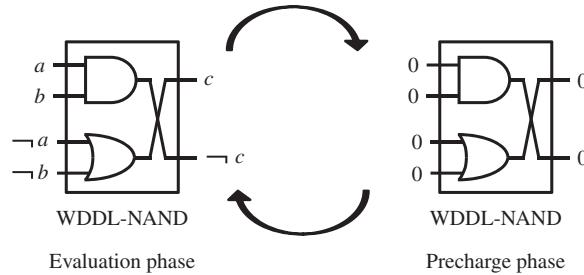
**Figure 7.4** Example circuit with WDDL technique (state B, evaluation phase)

In order to see the role of the precharge phase, let us simulate the power consumption consumed by the state transition. In the case that the state transition from state A to state B occurs in a consecutive clock cycle, the amount of signal toggles from 0 to 1 and from 1 to 0 both become six. Therefore, we would observe the power consumption corresponding to the number of signal toggles. On the other hand, when the state transition from state A to state A happen to occur in consecutive clock cycles, the values are not changed at all in the combinatorial logics, which leads to no power consumption. This observation infers that the power consumption is dependent on the state transitions. Therefore, this could be the vulnerability against the side-channel attacks. Suppose that state 0 in the precharge phase is processed between states A and B, that is, transitions of states A, 0, and B in sequence. From state A to state 0, the amount of signal toggles from 1 to 0 is six, and 0-to-1 signal toggles never happen. From state 0 to state B, the amount of signal toggles from 0 to 1 is six, and 1-to-0 signal toggles never happen. The same results will be obtained even for the case of the transitions of states A, 0, and A. Furthermore, this result holds even if different values are provided from the DFFPs. That is to say, state 0 in the precharge phase plays an important role to make power consumption constant.

**Exercise 7.1** Confirm that the example circuit shown in Figure 7.2–7.4 consumes constant power as far as the precharge phase is processed between the evaluation phases.

### 7.1.2 WDDL-NAND Gate

Based on the explanations of the circuit with the WDDL technique, the details of the WDDL gates are explained here. It is found that there exist several constraints for the WDDL gates to



**Figure 7.5** The WDDL-NAND gate for zero-precharge case

guarantee the constant power consumption. The WDDL-NAND gate illustrated in Figure 7.5 satisfies all the constraints mentioned in the previous section. The positive-logic operation is performed with the AND gate and the negative-logic operation is done with the OR gate in this case. Then, the output wires of the AND and OR gates are crossed so that the positive and negative values are switched. The operation of the WDDL-NAND gate during the evaluation phase is defined as

$$\begin{cases} \neg c = a \wedge b, \\ c = \neg a \vee \neg b. \end{cases} \quad (7.1)$$

The straightforward implementation for the WDDL-NAND would be to use the NAND gate. Nevertheless, the reason why the AND gate is used instead of the NAND gate in implementing the WDDL-NAND gate shown in Figure 7.5 is to make all the signals as zeros in the precharge phase.

As shown in Figure 7.5, signals behave specially in the precharge phase. Therefore, Equation (7.1) holds only in the evaluation phase. In the precharge phase, all inputs for the WDDL-NAND are zeros, and hence the outputs also become zero. Alternation of the precharge and evaluation phases is the key to the solution for constant power consumption.

### 7.1.3 WDDL-NOR and WDDL-INV Gates

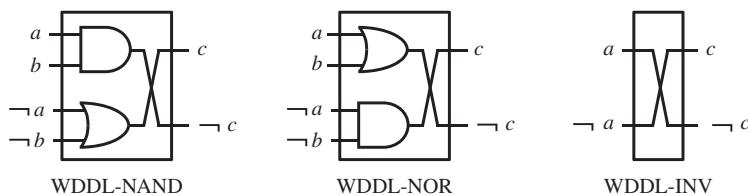
In a similar manner, the operation of the WDDL-NOR gate in the evaluation phase is defined as

$$\begin{cases} \neg c = a \vee b, \\ c = \neg a \wedge \neg b. \end{cases} \quad (7.2)$$

Note that the WDDL-INV gate does not require any operational logic gate. It is realized with the crossed wiring as shown in Figure 7.6. Figure 7.7 shows the pseudo-Verilog code for WDDL-NAND, WDDL-NOR, and WDDL-INV gates.

### 7.1.4 Precharge Logic for WDDL Technique

Precharge is essential for circuits with the WDDL technique in order to guarantee the constant Hamming weight of the signals. Figure 7.8 shows a precharge logic that outputs zeros when



**Figure 7.6** The WDDL-AND, WDDL-NOR, and WDDL-INV gates

```
// Data signals are represented with two bits as
// signal = {positive bit, negative bit}

module wddl_nand (a, b, c);

    input [1:0] a, b;
    output [1:0] c;

    assign c[0] = a[1] & b[1];
    assign c[1] = a[0] | b[0];

endmodule

module wddl_nor (a, b, c);

    input [1:0] a, b;
    output [1:0] c;

    assign c[0] = a[1] | b[1];
    assign c[1] = a[0] & b[0];

endmodule

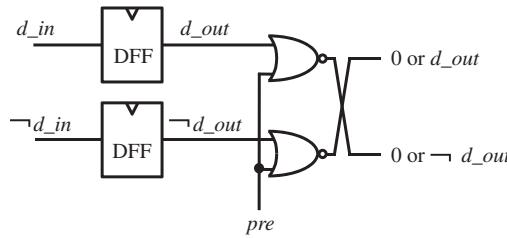
module wddl_inv (a, c);

    input [1:0] a;
    output [1:0] c;

    assign c[0] = a[1];
    assign c[1] = a[0];

endmodule
```

**Figure 7.7** Pseudo-Verilog code for WDDL NAND, NOR, and INV gates



**Figure 7.8** Precharge logics for circuits with WDDL gates

a precharge signal, *pre*, is high. When *pre* is low, it provides the values of DFFPs. Figure 7.9 shows a pseudo-Verilog code for DFFP.

The precharge effects spread to the entire combinatorial logics consisting of the WDDL gates, so that all of the input and output signals of the WDDL gates become zeros. From the electrical point of view, this physical action corresponds to discharge. After precharging, *pre* becomes low and the combinatorial circuits start being evaluated based on the values provided from DFFs. In this phase, exactly a half of the signals become ones, whereas the rest of the signals stay zeros, which contributes to constant power consumption. Figure 7.10 explains the details.

When *pre* is low, Hamming weight of signals for the WDDL-NAND and WDDL-NOR gates is both 3, and when *pre* is high, that is, Hamming weight is 0 as all the signals become zeros. Therefore, regardless of the input values, we know that the WDDL gate consumes constant

```
module wddl_preamble (clk, rst_n, pre, d_in, d_preamble);

    input pre; // Precharge signal
    input [1:0] d_in; // Input data for DFF
    output [1:0] d_preamble; // Output data with precharge

    reg [1:0] d_out; // Output data of DFF

    always @ (posedge clk or negedge rst_n) begin
        if (rst_n == 0)
            d_out <= 0;
        else
            d_out <= d_in;
    end

    assign d_preamble[0] = (d_out[1] | pre);
    assign d_preamble[1] = (d_out[0] | pre);

endmodule
```

**Figure 7.9** Pseudo-Verilog code for DFF with precharge logics (DFFP)

WDDL-NAND							WDDL-NOR						
pre	a	b	$\neg a$	$\neg b$	c	$\neg c$	pre	a	b	$\neg a$	$\neg b$	c	$\neg c$
0	1	0	0	0	0	0	1	0	0	0	0	0	0
	0	0	1	1	1	0	0	0	1	1	1	1	0
	0	1	1	0	1	0	0	1	1	0	0	0	1
	1	0	0	1	1	0	1	0	0	1	0	0	1
	1	1	0	0	0	1	1	1	0	0	0	0	1

(a)    (b)

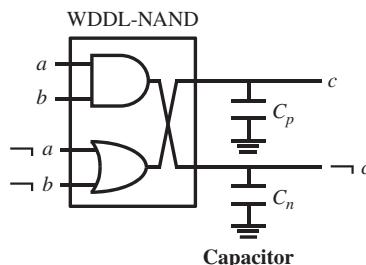
**Figure 7.10** Signal toggles for (a) charge and (b) discharge of WDDL gates

power corresponding to Hamming weight of 3. Furthermore, constant current is expected to flow to the ground when discharging.

However, in the real world, constant power is realized with an equivalent power consumption of the signal wires in the WDDL circuits. The significant challenge in the wire routing is to equalize the wire load capacitance of wires for  $c$  and  $\neg c$ , that is,  $C_p = C_n$ , as illustrated in Figure 7.11. Therefore, the WDDL circuit layout should be carefully taken care.

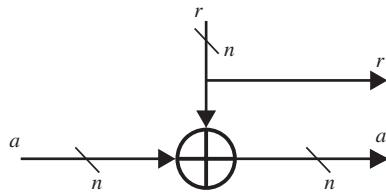
### 7.1.5 Intrinsic Fault Detection Mechanism of WDDL

The WDDL gates can easily detect a fault due to the nature of dual-rail logic. The WDDL gate operates one bit-wise operation twice in the positive and negative values at the same time. If one of the positive and negative logics has error, the output,  $(c, \neg c)$  becomes  $(1, 1)$  or  $(0, 0)$  that never happens in a proper computation.<sup>1</sup> For instance, by checking the XOR of  $c$  and  $\neg c$ , one can provide information whether or not an error occurs. Upon the detection of the error, a cryptographic hardware can be stopped so that any useful information to the attacker does not leak. By doing so, differential fault analysis cannot work since it requires a ciphertext generated in the erroneous computation.



**Figure 7.11** Wire load capacitance in WDDL gate

<sup>1</sup> If both of the positive and negative logics have errors  $(c, \neg c)$  seems correct and one cannot detect the errors. However, this type of multi-fault injection is much more difficult than the case of one-bit fault injection.



**Figure 7.12** Boolean masking of  $n$ -bit signal

**Exercise 7.2** Implement constant power 8-bit ripple-carry adder shown in Figure 2.5 using the WDDL technique, and discuss the impact on performance and cost.

**Exercise 7.3** For the WDDL-NAND gate and WDDL-NOR gate shown in Figure 7.6, denote the path delay of signal  $x$  as  $T_x$ . Assume  $T_a = T_{\neg a} < T_b = T_{\neg b}$ , discuss the signal transitions of  $(c, \neg c)$  and their delay times  $(T_c, T_{\neg c})$  following the style in Figure 5.78 and Table 5.2.

## 7.2 Logic-Level Masking Countermeasures

### 7.2.1 Overview of Masking Countermeasure

**Masking countermeasures** mask an intermediate variable,  $a$ , with a random number,  $r$ , which is called **mask**. The main purpose of the masking countermeasure is to make the intermediate values uncorrelated with side-channel information. That is, as far as using a fresh random number, the masked intermediate value is represented differently, which makes the side-channel attack difficult. In general, the freshness of the random number in the masking countermeasure has a significant impact on side-channel resistance. Therefore, reuse of random numbers should be avoided, and change of the random number should be performed frequently enough.

There are several kinds of masking operations such as **Boolean masking** and **arithmetic masking**. Boolean masking uses bitwise XOR operation, that is, addition over  $GF(2)$  as

$$a_r = a \oplus r, \quad (7.3)$$

where  $a_r$  is a masked value of  $a$  and  $r$  is a random number (mask).

On the contrary, arithmetic masking for a value,  $a$  is obtained with

$$(a + r) \bmod 2^n, \quad (7.4)$$

where  $n$  is the bit length of  $a$  and  $r$ . This type of masking can be used for block ciphers that employ modular additions in the algorithm.

Both types of masking scheme can decrease the correlation between the side-channel information from a device and intermediate values on wires in a hardware implementation. This makes the side-channel attack difficult. Hereafter, we focus on Boolean masking since it is a fundamental technique for many block ciphers including AES.

### 7.2.2 Operations on Values with Boolean Masking

Operations on values with Boolean masking are explained here. Suppose that a masked value,  $a_r$ , is operated with a Boolean function,  $f$ . That is,

$$f(a_r) = f(a \oplus r). \quad (7.5)$$

This equation indicates that the mask,  $r$ , cannot be easily removed or changed after operating the function,  $f$ , since  $r$  cannot be got rid of from  $f(a \oplus r)$  with a simple operation.

If  $f$  is a linear function, the Equation (7.5) becomes as

$$f(a_r) = f(a) \oplus f(r). \quad (7.6)$$

It can be seen that operations on masked values using linear functions can be implemented with performing  $f$  twice. Accordingly, the mask can be easily removed or changed simply with the XOR operation. Therefore, another random bit  $r'$  can be replaced with  $r$  easily since

$$f(a_{r'}) = f(a_r) \oplus f(r) \oplus f(r'). \quad (7.7)$$

In this form, further operations can be continued on the masked values.

From the observation, we know that the challenge in the masking countermeasures is to implement nonlinear functions efficiently.

### 7.2.3 Re-masking and Unmasking

The necessity of changing the mask and removing the mask is explained by using the example case of the AddRoundKey transformation of AES-128. More specifically, we consider the case when  $f$  is the addition in  $GF(2^{128})$ , that is, 128-bit XOR operation.

Suppose that the AddRoundKey transformation takes masked values,  $a_r$ , and a subkey,  $sk$ , as its input data, and perform the XOR operation, that is,  $a_r \oplus sk$ .

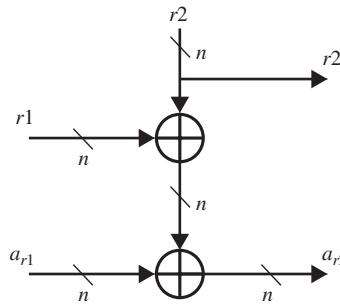
The output value of AddRoundKey transformation is also masked with the same random bits as used for the input value because

$$a_r \oplus sk = a \oplus r \oplus sk \quad (7.8)$$

$$= (a \oplus sk) \oplus r, \quad (7.9)$$

where  $(a \oplus sk)$  is the result of the AddRoundKey transformation. Therefore, neither hardware overhead nor additional computation is required.

However, from the view point of side-channel leakage, the random bits should be refreshed frequently in order to reduce the risk of leaking the information of the random bits. Refreshing the mask is easily realized with XOR operation as illustrated in Figure 7.13 that shows an



**Figure 7.13** Re-masking of  $n$ -bit signal

example for the re-masking. The masked value,  $a_{r1}$ , is re-masked with a fresh random bit,  $r2$ , and  $a_{r2}$  is output.

$$(a_{r1} \oplus (r1 \oplus r2)) \oplus sk = (a \oplus r1 \oplus r2 \oplus r1) \oplus sk \quad (7.10)$$

$$= (a \oplus r2) \oplus sk \quad (7.11)$$

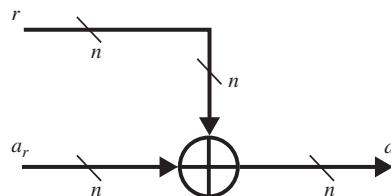
$$= a_{r2} \oplus sk. \quad (7.12)$$

The operation order must be taken care so that the unmasked value,  $a$ , never appears on any operations.<sup>2</sup> In other words, all of the operations have to use the masked values.

The masked values are also easily unmasked simply with XOR operations as shown in Figure 7.14. Note that unmasking should appear only in the last stage of a cipher operation such as after the AddRoundKey transformation of the last round of AES. Again, if unmasked value appears on a wire and if it contains sensitive information, the attacker can recover the sensitive information via side-channel analysis.

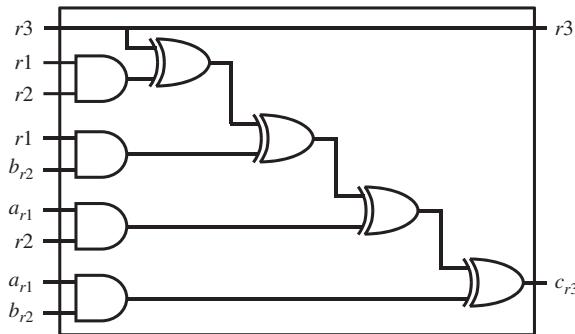
#### 7.2.4 Masked AND Gate

One possible implementation for the AND operations on values with Boolean masking is called **Masked AND** operation, which was proposed by Trichina *et al.* (2005). This gate-level masking can facilitate the difficulty in building up nonlinear functions that deals with Boolean



**Figure 7.14** Unmasking of  $n$ -bit signal

<sup>2</sup> Equation (7.10) uses  $a$  as input, however this is for confirming the correctness the AddRoundKey transformation.



**Figure 7.15** Masked AND gate

masking values. Figure 7.15 illustrates the masked AND gates. The masked AND performs the AND operation on masked inputs,  $a_{r1}$  and  $b_{r2}$ , with refreshing the mask using  $r3$  as

$$c_{r3} = (((r1 \wedge r2) \oplus r3) \oplus (r1 \wedge b_{r2})) \oplus (a_{r1} \wedge r2) \oplus (a_{r1} \wedge b_{r2}) \quad (7.13)$$

$$= (a \wedge b) \oplus r3. \quad (7.14)$$

*Proof.*

$$\begin{aligned} c_{r3} &= (r1 \wedge r2) \oplus r3 \oplus (r1 \wedge b_{r2}) \oplus (a_{r1} \wedge r2) \oplus (a_{r1} \wedge b_{r2}) \\ &= ((r1 \wedge r2) \oplus (r1 \wedge b_{r2})) \oplus ((a_{r1} \wedge r2) \oplus (a_{r1} \wedge b_{r2})) \oplus r3 \\ &= r1 \wedge (r2 \oplus b_{r2}) \oplus a_{r1} \wedge (r2 \oplus b_{r2}) \oplus r3 \\ &= r1 \wedge (r2 \oplus b \oplus r2) \oplus a_{r1} \wedge (r2 \oplus b \oplus r2) \oplus r3 \\ &= (r1 \wedge b) \oplus (a_{r1} \wedge b) \oplus r3 \\ &= ((r1 \oplus a_{r1}) \wedge b) \oplus r3 \\ &= ((r1 \oplus a \oplus r1) \wedge b) \oplus r3 \\ &= (a \wedge b) \oplus r3. \end{aligned}$$

□

There are five inputs,  $a_{r1}$ ,  $b_{r2}$ ,  $r1$ ,  $r2$ , and  $r3$ , that are operated to generate outputs,  $c_{r3}$  and  $r3$ . The order of the XOR operations has special importance for reducing the threat against the side-channel attacks. More specifically, one has to be careful in choosing the order of operations so that unmasked values are not observed on any wire between gates. Figure 7.16 describes the pseudo-Verilog code for the masked AND operation.

**Exercise 7.4** Confirm that all the intermediate values (the wires in Figure 7.15) in the masked AND are masked.

```

module Masked_AND (a_r1, b_r2, r1, r2, r3, c_r3);

    input a_r1, b_r2; // Masked inputs
    input r1, r2, r3; // Random bits for mask
    output r3, c_r3; // Masked output with fresh mask

    wire w1, w2, ..., w7;

    assign w1 = r1 & r2;
    assign w2 = r1 & b_r2;
    assign w3 = a_r1 & r2;
    assign w4 = a_r1 & b_r2;

    assign w5 = r3 ^ w1;
    assign w6 = w5 ^ w2;
    assign w7 = w6 ^ w3;
    assign c_r3 = w7 ^ w4;

endmodule

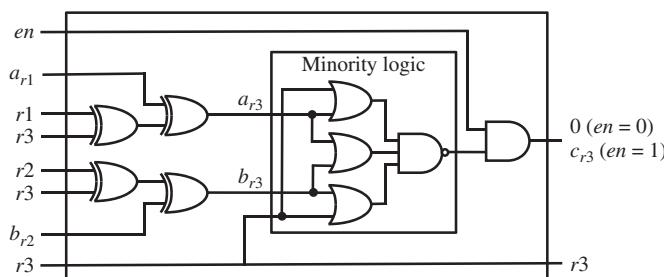
```

**Figure 7.16** Pseudo-Verilog code for masked AND gate

### 7.2.5 Random Switching Logic

The random switching logic (RSL) technique is another famous masking countermeasure, which was proposed by Suzuki *et al.* (2004) and Saeki *et al.* (2009). RSL can be implemented with a standard-cell-based library. Figure 7.17 illustrates a NAND gate realized with the standard-cell-based RSL technique.

The RSL-NAND gate has six inputs and two outputs. The inputs,  $a_{r1}$  and  $b_{r2}$ , are data signals for  $a$  and  $b$  that are masked with  $r1$  and  $r2$ , respectively. The masked signals are re-masked with a fresh random bit,  $r3$ , and the result of the NAND operation for  $a$  and  $b$  are performed. Another input signal,  $en$ , is used for suppressing the propagation of the glitch signals generated in the RSL-NAND gate. In other words,  $en$  should be low until all the evaluation of the signals



**Figure 7.17** Standard-cell-based RSL-NAND

```

module RSL_NAND (a_r1, b_r2, r1, r2, r3, c_r3);

    input a_r1, b_r2; // Masked inputs
    input r1, r2, r3; // Random bits for mask
    input en; // For gating the output
    output r3, c_r3; // Masked output with fresh masking

    wire w1, w2, ..., w6;
    wire a_r3, b_r3;

    assign w1 = r1 ^ r3; // Re-masking
    assign a_r3 = w1 ^ a_r1;
    assign w2 = r2 ^ r3;
    assign b_r3 = w2 ^ b_r2;

    assign w3 = a_r3 | b_r3;
    assign w4 = b_r3 | r3;
    assign w5 = r3 | a_r3;
    assign w6 = (w3 ^ w4 ^ w5); // Minority logic

    assign c_r3 = en ^ w6; // Output gating

endmodule

```

**Figure 7.18** Pseudo-Verilog code for RSL-NAND gate

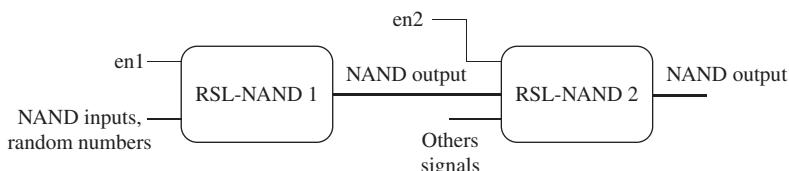
in the RSL-NAND operations completes. The pseudo-Verilog code for the RSL-NAND gate is described in Figure 7.18.

$$c_{r3} = en \wedge \text{Minority}(a_{r1} \oplus (r1 \oplus r3), b_{r2} \oplus (r2 \oplus r3), r3) \quad (7.15)$$

$$= en \wedge (\neg(a \wedge b) \oplus r3). \quad (7.16)$$

*Proof.*

$$\begin{aligned} c_{r3} &= en \wedge \text{Minority}(a \oplus r1 \oplus r1 \oplus r3, b \oplus r2 \oplus r2 \oplus r3, r3) \\ &= en \wedge \text{Minority}(a \oplus r3, b \oplus r3, r3) \\ &= en \wedge \text{Minority}(a, b, 0) \oplus r3 \\ &= en \wedge (\neg(a \wedge b) \oplus r3). \quad \square \end{aligned}$$



**Figure 7.19** Two RSL NAND connected in sequence

**Exercise 7.5** Consider a circuit with the RSL technique as shown in Figure 7.19, two RSL-NAND gates are connected in sequence. All the wires between RSL-NAND are zeros in the precharge phase by setting the enable signal low, that is,  $en = 0$ . In the evaluation phase, the enable signal is set to high. Discuss what kind of timing constraints are needed for the enable signals  $en1$  and  $en2$  in order to escape the glitch propagation through the circuit.

### 7.2.6 Threshold Implementation

**Threshold implementation** or TI is a masking method based on secret sharing, which is proven to be resistant against the first-order side-channel attacks even in the presence of signal glitches. TI was proposed by Nikova *et al.* (2006). Notice that TI is one of the first countermeasures that overcome the vulnerability caused by the glitch signals fundamentally. TI has a characteristic that the intermediate value can be masked not only by one random value but by two or more of them, depending on the number of shares.

For the same reason as other masking schemes, TI can be easily applied to the linear transformation. Therefore, the focus of TI is on how to mask the nonlinear transformation. The basic principle of TI can be summarized as follows. Each input variable of a nonlinear transformation is separated into several **shares** such that the addition over  $GF(2)$  of the shares equals to the input data. That is, the nonlinear transformation is separated into several functions.<sup>3</sup> Each function uses the shares of input to perform the calculation, and the addition over  $GF(2)$  for the outputs of all functions is the expected output. In order to make TI resistant against the first-order side-channel attacks, one has to assure that for every input variable of each function, all of the shares are not used. By doing so, one can ensure that the calculation of each function is independent from the original input variables, and hence the first-order side-channel resistance is achieved.

Let us take the AND gate,  $x \wedge y$ , as an example. One can separate each input variable into three shares, that is,  $x = x_1 \oplus x_2 \oplus x_3$  and  $y = y_1 \oplus y_2 \oplus y_3$ . The calculation of the TI-AND gate can be realized with three functions as

$$s_1 = (x_2 \wedge y_2) \oplus (x_2 \wedge y_3) \oplus (x_3 \wedge y_2), \quad (7.17)$$

$$s_2 = (x_3 \wedge y_3) \oplus (x_1 \wedge y_3) \oplus (x_3 \wedge y_1), \quad (7.18)$$

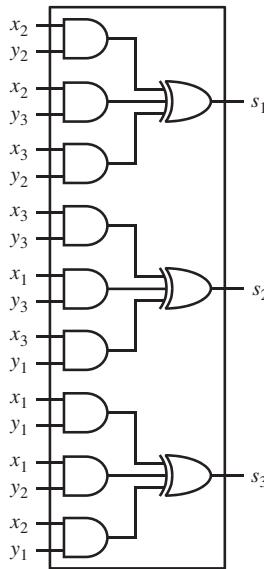
$$s_3 = (x_1 \wedge y_1) \oplus (x_1 \wedge y_2) \oplus (x_2 \wedge y_1). \quad (7.19)$$

One can see that  $s_1$  does not use shares,  $x_3$  and  $y_3$ , so  $s_1$  is independent from  $x$  and  $y$ . Similarly, it can be found that  $s_2$  and  $s_3$  are also independent from  $x$  and  $y$ . The addition of shares over  $GF(2)$ ,  $s_1$ ,  $s_2$  and  $s_3$ , is

$$s_1 \oplus s_2 \oplus s_3 = x \wedge y. \quad (7.20)$$

---

<sup>3</sup>The separated functions are not necessarily the same.



**Figure 7.20** Shared AND gate with TI technique

*Proof.*

$$\begin{aligned}
 s_1 \oplus s_2 \oplus s_3 &= (x_2 \wedge y_2) \oplus (x_2 \wedge y_3) \oplus (x_3 \wedge y_2) \oplus (x_3 \wedge y_3) \oplus (x_1 \wedge y_3) \oplus (x_3 \wedge y_1) \\
 &\quad \oplus (x_1 \wedge y_1) \oplus (x_1 \wedge y_2) \oplus (x_2 \wedge y_1) \\
 &= x_1 \wedge (y_1 \oplus y_2 \oplus y_3) \oplus x_2 \wedge (y_1 \oplus y_2 \oplus y_3) \oplus x_3 \wedge (y_1 \oplus y_2 \oplus y_3) \\
 &= (x_1 \oplus x_2 \oplus x_3) \wedge (y_1 \oplus y_2 \oplus y_3) \\
 &= x \wedge y.
 \end{aligned}$$

□

Therefore, operations corresponding to Equations (7.17)–(7.19) can be regarded as a split computation of the AND operation based on the secret sharing scheme with three shares. Figure 7.20 shows the block diagram for the TI-AND gate, and Figure 7.21 describes the corresponding pseudo-Verilog code.

**Exercise 7.6** Compare the penalty on speed performance and area cost of each gate-level countermeasure (e.g., Masked-AND, RSL, TI).

**Exercise 7.7** Discuss whether or not the DFA attack can be applied to the AES that has its S-box protected by masking countermeasures.

- (a) Consider whether or not the fault used in DFA can be injected to the S-boxes with masking countermeasure.
- (b) Consider whether or not the propagation of active bytes is the same for the S-boxes with masking countermeasure.

### 7.3 Higher Level Countermeasures

The gate-level countermeasures require a significant degradation of the speed performance and hardware cost. This motivates us to consider algorithm-level countermeasures that are normally cost-effective compared to the gate-level countermeasures. In the algorithm-level countermeasures, a range of protection is in units of a composite operation that consists of multiple logical gates in hardware. An algorithm-oriented optimization is often possible, and a better trade-off between performance and cost is likely to be improved compared to the gate-level countermeasures.

Differences between architecture- and algorithm-level countermeasures are not so clear since they are tightly related to the type of computation and its grain size. However, in one

```

module Shared_AND (x_1, x_2, x_3, y_1, y_2, y_3, s_1, s_2, s_3);
// x and y are split into three shares as
// x = x_3, x_2, x_1 and y = y_3, y_2, y_1
  input [3:1] x, y;
  output [3:1] s;

  wire w1, w2, ..., w9;

  assign w1 = x[2] & y[2];
  assign w2 = x[2] & y[3];
  assign w3 = x[3] & y[2];
  assign s[1] = w1 ^ w2 ^ w3;

  assign w4 = x[3] & y[3];
  assign w5 = x[1] & y[3];
  assign w6 = x[3] & y[1];
  assign s[2] = w4 ^ w5 ^ w6;

  assign w7 = x[1] & y[1];
  assign w8 = x[1] & y[2];
  assign w9 = x[2] & y[1];
  assign s[3] = w7 ^ w8 ^ w9;

endmodule

```

**Figure 7.21** Pseudo-Verilog code for shared AND gate with TI technique

perspective, the architecture-level countermeasure can be regarded as a hardware architecture that offers the resistance against the side-channel and/or fault attacks regardless of the performed algorithm. For instance, a general-purpose CPU implemented with the logic-level countermeasure such as the WDDL technique could offer the architecture-level countermeasure to any software implementation. For another example, a current equalizer circuit to isolate the critical encryption/decryption activity can be considered one of the architecture-level countermeasures in terms of preventing the power analysis attacks.<sup>4</sup> One example was shown by Tokunaga and Blaauw (2009).

In this section, countermeasure at each abstraction level is explained together with several examples. Especially, countermeasures for the fault attacks are focused on since the fault can be detected at any abstraction level. On the other hand, higher the abstraction level becomes, more difficult the side-channel countermeasure tends to be. This is because the side-channel attack exploits the gate-level information leakage, and hence a countermeasure is necessary in the lowest abstraction level.

### 7.3.1 Algorithm-Level Countermeasures

The masking countermeasures such as the masked AND and TI are not necessary to be implemented in the gate level. For instance, as for the masked AND technique, Equation (7.10) holds even if the bitwise AND and XOR operations are replaced with  $n$ -bit multiplication and addition in  $GF(2^n)$ , respectively as

$$C_{r3} = (((R1 \times R2) + R3) + (R1 \times B_{R2})) + (A_{R1} \times R2) + (A_{R1} \times B_{R2}) \quad (7.21)$$

$$= (A \times B) + R3, \quad (7.22)$$

where the operators, “ $\times$ ” and “ $+$ ” respectively are multiplication and addition in  $GF(2^n)$  with an irreducible polynomial,  $P(x)$ , whose degree is  $n$ .  $R1$ ,  $R2$ , and  $R3$  are  $n$ -bit random numbers used for masking  $n$ -bit multiplication in  $GF(2^n)$  of  $C = A \times B$ .

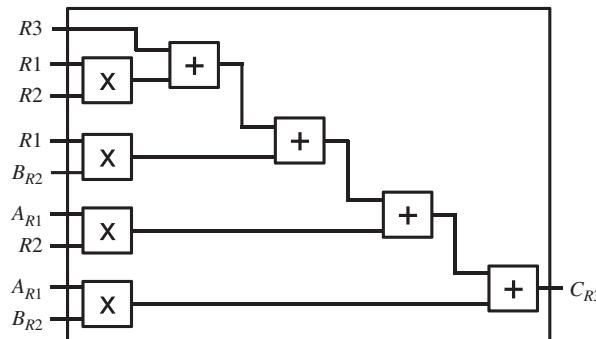
Figure 7.22 illustrates the block diagram for masked multiplication in  $GF(2^n)$ . Notice that it consists of four normal multiplication modules and four normal additions that are simply realized with XOR gates. That is, there is some flexibility in choosing a multiplication module. Figure 7.23 describes a pseudo-Verilog code for masked multiplier in  $GF(2^n)$ .

**Exercise 7.8** Discuss the cost of the masked multiplication module in  $GF(2^n)$  for  $n = 2, 4, 8$ .

TI can also be applied to the architecture-level countermeasure. For example, it has been shown that TI for the finite field inversion in AES S-box can be achieved with five shares, which is shown by Nikova *et al.* (2006). Note that the original TI has been verified to be

---

<sup>4</sup>This countermeasure is not effective against attacks where the attackers can exploit the local information leakage, for example, invasive side-channel attacks and EMA attacks.

**Figure 7.22** Masked modular multiplication in  $GF(2^n)$ 

```

module MM (A, B, C);
    input [n-1:0] A, B;
    output [n-1:0] C;

    //Description for multiplication in  $GF(2^n)$ 
    //Irreducible polynomial, for example,  $x^8 + x^5 + x^3 + x^2 + 1$ 

endmodule

module Masked_MM (A_R1, B_R2, R1, R2, R3, C_R3);

    input [n-1:0] A_R1, B_R2; // Masked inputs
    input [n-1:0] R1, R2, R3; // Random bits for mask
    output [n-1:0] C_R3; // Masked output with fresh masking
    wire [n-1:0] W1, W2, ..., W7;

    MM MM1 (R1, R2, W1);
    MM MM2 (R1, B_R2, W2);
    MM MM3 (A_R1, R2, W3);
    MM MM4 (A_R1, B_R2, W4);

    assign W5 = R3 ^ W1;
    assign W6 = W5 ^ W2;
    assign W7 = W6 ^ W3;
    assign C_R3 = W7 ^ W4;

endmodule

```

**Figure 7.23** Pseudo-Verilog code for masked multiplier in  $GF(2^n)$

resistant against the first-order power analysis in many literatures; however, it is known that the original TI cannot provide the resistance against higher order side-channel attacks.

### 7.3.1.1 Fault Detection in Algorithm Level

A fault detection mechanism can be embedded in a block cipher module by utilizing the feature of AES encryption and decryption. One of the representative fault-detection techniques utilizing the feature of the AES encryption/decryption algorithm is shown in Algorithm 7.1, where  $\perp$  denotes the reject symbol.

For a plaintext,  $P$ , and a secret key,  $K$ , encryption is performed firstly. Secondly, the encrypted result,  $C_1$ , is decrypted and stored as  $P_1$ . Obviously, this is a redundant operation; however, it can be used to detect a fault during the operation of the encryption by checking whether or not  $P$  and  $P_1$  are the same value. Namely, if a fault happens in the encryption of  $E_K(P)$ , a wrong ciphertext,  $C'_1$ , is generated. Accordingly, the decryption for  $C'_1$  generates a plaintext different from  $P$ , and the fault is detected correctly with high probability.<sup>5</sup>

The drawback is its long latency to perform all the steps in Algorithm 7.1 sequentially. More precisely, it requires the operation time for the AES encryption and decryption at least since the decryption can be performed only after the encryption result is ready. That is, a parallel architecture introduced in Section 3.1 cannot be employed in this case. Moreover, even when the encryption is correctly performed, the algorithm might go to Step 4 upon the failure of the AES decryption, which means a false-negative fault detection occurs.

Another example for the fault detection in the AES encryption is shown in Algorithm 7.2. This algorithm checks every round operation, RF, by checking whether or not the input of RF is correctly recovered with the inverse operation,  $RF^{-1}$ . The latency to detect the fault is significantly improved compared to Algorithm 7.1. Although the algorithm continues regardless of a fault, it can be changed so that some appropriate action can be taken immediately after detecting a fault. The speed performance can also be improved by exploiting the parallelism in steps after unrolling the for loop. However, there is still a possibility to have a false-negative detection.

---

#### Algorithm 7.1 AES Encryption with Fault Detection by Decrypting Encryption Result

---

**Input:** Plaintext  $P$  and secret key  $K$ ;

**Output:** Ciphertext  $C = E_K(P)$

```

1:  $C_1 \leftarrow E_K(P);$ 
2:  $P_1 \leftarrow D_K(C_1);$ 
3: if  $P \neq P_1$  then
4:   return  $C \leftarrow \perp$ ; //fault is detected
5: else
6:   return  $C \leftarrow C_1;$ 
7: end if
```

---

<sup>5</sup> It is assumed that the probability of having another fault such that  $P = D_K(C')$  is low.

**Algorithm 7.2** AES Encryption with Fault Detection in Round Operations

---

**Input:** Plaintext  $P$  and secret key  $K$  (subkeys:  $sk_0, sk_1, \dots, sk_{10}$ )  
**Output:** Ciphertext  $C = E_K(P)$

```

1:  $d \leftarrow 0$ ;
2:  $state_0 \leftarrow P \oplus sk_0$ ;
3: for  $i = 1$  to  $9$  do
4:    $state_i \leftarrow RF(state_{i-1}, sk_i)$ ; // 1st to 9th round operation for AES encryption
5:    $state'_{i-1} \leftarrow RF^{-1}(state_i, sk_i)$ ; // Inverse round operation
6:   if  $state_{i-1} \neq state'_{i-1}$  then
7:      $d \leftarrow 1$ ; //fault is detected
8:   end if
9: end for
10:  $state_{10} \leftarrow RF_{last}(state_9, sk_{10})$ ; //10th round operation for AES encryption
11:  $state'_9 \leftarrow RF_{last}^{-1}(state_{10}, sk_{10})$ ; // Inverse round operation
12: if  $state_9 \neq state'_9$  then
13:    $d \leftarrow 1$ ; //fault is detected
14: end if
15: if  $d = 0$  then
16:   return  $C \leftarrow state_{10}$ ;
17: else
18:   return  $C \leftarrow \perp$ ;
19: end if
```

---

**Exercise 7.9** Draw a block diagram for parallelized hardware implementation of Algorithm 7.2 and discuss the improvement in the speed performance compared to Algorithm 7.1.

### 7.3.2 Architecture-Level Countermeasures

For instance, consider a hardware implementation of a side-channel-resistant CPU in which entire circuits are protected with the gate-level countermeasure technique against the side-channel attacks. This solution significantly reduces leaked information from any software implementation not only for cryptographic operations but also for any other functional operations. Therefore, such a CPU would be over-engineered since it often performs non-cryptographic operations that do not require a side-channel resistance. As for the fault resistance, the same observation can be seen, that is, a CPU implemented with the WDDL technique may be able to detect any faults in the gate level as previously mentioned; however, the cost efficiency is not satisfactory depending on the application.

Instead, a dedicated hardware accelerator module can be implemented for cryptographic algorithms considering the speed-cost trade-offs. The whole or a part of the cryptographic modules, which should be protected from side-channel and fault attacks, are implemented separately from CPU so that they are resistant against those attacks. This hardware/software separation enables us to apply a countermeasure to a limited region of the implementation. For example, an accelerator module for block cipher. The communication between CPU and the accelerator must be carefully implemented so that sensitive information does not leak via the data bus.

### 7.3.2.1 Fault Detection in Architecture Level

Detecting a fault is also possible in the architecture level. There are two major techniques; temporal duplication and spatial duplication of a functional operation. Algorithms 7.3 and 7.4 show the examples using AES encryption, that is,  $C = E_K(P)$ . Note that the functional operation does not have to be AES encryption, but any functional operation can be applied in both the algorithms. Namely, they can be regarded as general architecture-level countermeasures.

Algorithm 7.3 performs AES encryption twice sequentially, and compares the results at Step 3. Therefore, it takes twice as much time as AES encryption (**temporal duplication**). However, only one AES encryption module is needed, which leads to a cost-efficient implementation. It is worth noting that the number of repetitions of AES encryption can be more than twice for the purpose of a strict test of faults or an avoidance of false-negative detections.<sup>6</sup>

On the contrary, in Algorithm 7.4 the same AES encryption is performed in parallel. Therefore, it requires two AES encryption modules in hardware, which means that the area cost will be doubled (**spatial duplication**). One of the merits is obviously in its speed performance. More specifically, its hardware implementation can be performed at the same speed as one AES encryption. Note that more than two modules can be used as well.

### 7.3.3 Protocol-Level Countermeasure

There is no perfect countermeasure to protect any side-channel and fault attacks. Only what we can do to protect cryptographically sensitive data from those attacks is to refresh the secret key before being retrieved by the attacker. Therefore, the so-called **key lifetime** should be

---

#### Algorithm 7.3 AES Encryption with Fault Detection Using Temporal Duplication

---

**Input:** Plaintext  $P$  and secret key  $K$   
**Output:** Ciphertext  $C = E_K(P)$

```

1:  $C_1 \leftarrow E_K(P);$ 
2:  $C_2 \leftarrow E_K(P);$ 
3: if  $C_1 \neq C_2$  then
4:   return  $C \leftarrow \perp$ ; // fault is detected
5: else
6:   return  $C \leftarrow C_1;$ 
7: end if
```

---

<sup>6</sup> It depends on how to set the condition in Step 3 in Algorithm 7.3.

**Algorithm 7.4** AES Encryption with Fault Detection Using Spatial Duplication

---

**Input:** Plaintext  $P$  and secret key  $K$   
**Output:** Ciphertext  $C = E_K(P)$

```

1:  $C_1 \leftarrow E_K(P)$  and  $C_2 \leftarrow E_K(P)$ ; // operated in parallel
2: if  $C_1 \neq C_2$  then
3:   return  $C \leftarrow \perp$ ; // fault is detected
4: else
5:   return  $C \leftarrow C_1$ ;
6: end if
```

---

considered in the protocol level. In order to determine the key lifetime, countermeasures for the side-channel and fault attacks have to go beyond the attacker's ability. There are still a lot of open questions about the countermeasure, which suggests the necessity of further research.

**Exercise 7.10** Suppose that laser equipment is available to inject arbitrary faults both in time and in space for a very powerful attacker. In Algorithms 7.3, if the attacker injects the same fault at Step 1 and Step 2, the countermeasure in Algorithms 7.3 can be bypassed and the faulty ciphertext become available. What kind of fault injections are required to bypass the countermeasure in Algorithms 7.3?

**Exercise 7.11** In the case that the number of repetitions of AES encryption is three in Algorithm 7.3, discuss the possible conditions in Step 3 and their effects on the resistance against fault attacks and on the robustness of the hardware implementation.

**Exercise 7.12** In the case that three AES encryption modules are used in Algorithm 7.4, discuss the possible conditions in Step 2 and their effects on the resistance against fault attacks and on the robustness of the hardware implementation.

## Bibliography

- (ed. Joye M and Tunstall M) 2012 *Fault Analysis in Cryptography*. Springer-Verlag.  
 Nikova S, Rechberger C and Rijmen V 2006 Threshold implementations against side-channel attacks and glitches  
*Information and Communications Security, 8th International Conference, ICICS 2006, Raleigh, NC, USA, December 4-7, 2006, Proceedings*, pp. 529–545.

- Saeki M, Suzuki D, Shimizu K and Satoh A 2009 A design methodology for a DPA-resistant cryptographic LSI with RSL techniques *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*, pp. 189–204.
- Satoh A, Sugawara T, Homma N and Aoki T 2008 High-performance concurrent error detection scheme for AES hardware In *Cryptographic Hardware and Embedded Systems? CHES 2008* (ed. Oswald E and Rohatgi P), vol. **5154** of Lecture Notes in Computer Science, pp. 100–112. Springer-Verlag Berlin and Heidelberg.
- Suzuki D, Saeki M and Ichikawa T 2004 Random switching logic: a countermeasure against DPA based on transition probability. *IACR Cryptology ePrint Archive* **2004**, 346.
- Tiri K and Verbauwheide I 2004 A logic level design methodology for a secure DPA resistant ASIC or FPGA implementation *DATE*, pp. 246–251.
- Tokunaga C and Blaauw D 2009 Secure AES engine with a local switched-capacitor current equalizer *IEEE International Solid-State Circuits Conference, ISSCC 2009, Digest of Technical Papers, San Francisco, CA, USA, 8-12 February, 2009*, pp. 64–65.
- Trichina E, Korkishko T and Lee K 2005 Small size, low power, side channel-immune aes coprocessor: design and synthesis results In *Advanced Encryption Standard ? AES* (ed. Dobbertin H, Rijmen V and Sowa A), vol. **3373** of Lecture Notes in Computer Science, pp. 113–127. Springer-Verlag, Berlin and Heidelberg.

# Index

- abstraction level, 269
- active, 112
- active byte, 91
- active byte with respect to
  - the difference, 91
- addition chain, 8
- additive inverse, 4
- AddRoundKey, 17
- AES, 12
  - AES-128, 12
  - AES-192, 12
  - AES-256, 12
  - AES-comp, 169
  - AES-pprm1, 169
- algorithmic noise, 170
- all property, 132
- AND, 3
- arithmetic logic unit, 29
- asynchronous-style design flow, 27
- attack complexity, 74
- attack model, 71
  - balanced property, 137
  - basic impossible characteristic, 123
  - binary field, 4
  - block, 2
  - block cipher, 2
  - Boolean domain, 3
  - Boolean functions, 3
- Boolean masking, 277
- burst access mode, 41
- carry-select adder, 50
- ciphertext, 1
- clock, 29
- clock edge, 28
- clock jitter, 30
- clock period, 40
- clock signal, 27
- clock skew, 30
- clockwise collision, 196
- clockwise collision analysis, 196
- codebook, 77
- combinatorial logics, 31
- complementary metal-oxide-semiconductor (CMOS), 27
- constant property, 132
- controller, 29
- correlation power analysis (CPA), 192
- correlation-enhanced power analysis
  - collision attack, 199
- counter mode, 65
- countermeasures, 269
- critical fault injection intensity, 215
- critical path delay, 43
- cryptology, 1
- cryptosystems, 1
- CTR mode, 65

- data, 73  
 data signals, 29  
 datapath, 29, 39  
 decryption oracle, 72  
 delay flip flop (DFF), 32  
 design automation (DA), 27  
 determining bit, 94  
 diagonal, 25  
 dictionary attack, 77  
 difference, 78  
 difference of means (DoM), 181  
 differential characteristic, 91  
 differential distribution table (DDT), 87  
 differential fault analysis (DFA), 208  
 differential power analysis (DPA), 163  
 distinguishing attack, 93  
 divide-and-conquer, 154  
 dynamic timing analysis (DTA), 46  
  
 encryption, 1  
 encryption oracle, 72  
 equivalent transformation of the subkey  
     addition, 127  
 evaluation function, 165  
 evaluation phase, 271  
 exhaustive search, 74  
 extended binary field, 4  
  
 false path, 46  
 fault attack (FA), 151  
 fault model, 209  
 fault sensitivity (FS), 215  
 fault sensitivity analysis (FSA), 215  
 filtering, 98  
 filtering power, 98  
 finite field, 3  
 finite state machine (FSM), 36  
 full adder (FA), 31  
  
 Galois field, 3  
 gate equivalent, 47  
  
 Hamming distance (HD) model, 169  
 Hamming weight (HW) model, 169  
 hiding logics, 269  
 higher-order integral cryptanalysis, 141  
  
 hold buffer, 44  
 hold time, 43  
  
 implementation attacks, 149  
 impossible differential cryptanalysis, 111  
 indistinguishability, 70  
 input difference, 81  
 INV, 3  
 inverse diagonal, 25  
 inversion, 3  
 involution, 19  
 irreducible polynomial, 5  
  
 key lifetime, 290  
 key recovery resistance, 70  
 key schedule function (KSF), 12  
 key space, 118  
  
 latency, 47  
 layout, 28  
 leakage model, 165  
 least significant bit (LSB), 32  
 linear functions, 7  
 logic synthesis, 28  
 logical gates, 28  
 loop architecture, 51  
 loop-unrolled, 51  
  
 mask, 277  
 masked AND, 279  
 masking countermeasures, 277  
 masking logics, 269  
 maximum distance separable, 12  
 memory, 29, 73  
 message, 1  
 MixColumns, 17  
 mode of operation, 65  
 module, 29  
 most significant bit (MSB), 32  
 multiple impossible differential  
     characteristics, 124  
 multiplicative inverse, 4  
  
 negative edge, 29  
 negative logic, 270  
 netlist, 28

- non-profiling analysis, 156
- nonlinear functions, 7
- normal basis, 5
- OR, 3
- oracle, 72
- output difference, 81
- parallel architecture, 49
- path delay, 32, 39, 205
- physical attacks, 149
- pipeline architecture, 55
- pipeline stall, 55
- plaintext, 1
- plaintext recovery resistance, 70
- polynomial basis, 4
- positive edge, 29
- positive logic, 270
- precharge phase, 270
- precharge value, 270
- profiling analysis, 156
- pseudo-Random Permutation, 71
- queries, 72
- random switching logic (RSL), 281
- ranking test, 96
- reduced instruction set computer (RISC), 55
- register file, 41
- register transfer level (RTL), 27
- reset, 29
- reset signal, 30
- right pairs, 95
- ripple-carry adder, 32
- round function, 2
- round operation, 37
- S-box, 9
- scalability, 55
- selection function, 165
- sequential logics, 32
- setup time, 43
- shares, 283
- ShiftRows, 17
- side-channel attack (SCA), 151
- side-channel information, 152
- signal toggles, 40
- signal-to-noise ratio, 98
- simple power analysis (SPA), 163
- spatial duplication, 290
- state, 15
- static random access memory (SRAM), 40
- static timing analysis (STA), 45
- structure, 122
- SubBytes, 17
- subkey space, 118
- subkeys, 12
- substitution table, 9
- substitution-permutation network (SPN), 2
- synchronous design, 27
- tamper-proofed device, 154
- temporal duplication, 290
- threshold implementation (TI), 283
- throughput, 47
- time, 73
- traces, 160
- transfer gate (TG), 33
- true paths, 46
- truth table, 3
- Verilog HDL, 28
- Vernam cipher, 2
- wave dynamic differential logic (WDDL), 270
- whitening, 13
- wide trail strategy, 108
- wires, 28
- write enable signal, 41
- wrong pairs, 95
- XOR, 2
- zero-value analysis, 190

# **WILEY END USER LICENSE AGREEMENT**

Go to [www.wiley.com/go/eula](http://www.wiley.com/go/eula) to access Wiley's ebook EULA.