SIDE CHANNEL ATTACKS ON SYMMETRIC KEY PRIMITIVES

YASER ESMAEILI SALEHANI

A THESIS

IN

THE DEPARTMENT

OF

ELECTRICAL AND COMPUTER ENGINEERING

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF MASTER OF APPLIED SCIENCE IN ELECTRICAL AND

COMPUTER ENGINEERING

CONCORDIA UNIVERSITY

MONTRÉAL, QUÉBEC, CANADA

JULY 2011

CONCORDIA UNIVERSITY
SCHOOL OF GRADUATE STUDIES

This is to certify that the thesis prepared

By: Yaser Esmaeili Salehani

Entitled: "Side Channel Attacks on Symmetric Key Primitives"

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science

Complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Chair
Dr. R. Raut

_____ Examiner, External
Dr. A. Bagchi (BCEE) To the Program

_____ Examiner
Dr. A. Agarwal

_____ Supervisor
Dr. A. Youssef

Approved by: _____
Dr. W. E. Lynch, Chair
Department of Electrical and Computer Engineering

_____20_____ _____
Dr. Robin A. L. Drew
Dean, Faculty of Engineering and
Computer Science

Abstract

Side Channel Attacks on Symmetric Key Primitives

Yaser Esmaeili Salehani

Cryptographic primitives, including symmetric key encryption algorithms, are the basic building blocks of security systems. Cryptanalytic attacks against these algorithms can be divided into two classes: pure mathematical attacks and side channel attacks. Pure mathematical attacks are traditional cryptanalytic techniques that rely only on known or chosen input-output pairs of the encryption function, and exploit the inner structure of the cipher to reveal secret key information. In side channel attacks, the physical implementation of the cryptographic algorithms is considered. In particular, in this class of attacks, it is assumed that the attacker has some access to the cryptographic device and is able to make measurements with respect to time or power consumption, or is able to induce errors in the memory or operation of the device. The additional information gained by utilizing such a side channel are then combined with methods that exploit the inner structure of the cipher to reveal the secret key. The wide spread of unprotected software or hardware cryptographic implementations can offer various possibilities for these side channel attacks. Throughout this thesis, we present side channel cryptanalysis against three symmetric key ciphers.

First, we present a differential fault analysis of SOSEMANUK which is a software-based stream cipher that supports a variable key length between 128 and 256 bits and a 128-bit

initial value. SOSEMANUK has passed all three stages of the ECRYPT stream cipher project and is a member of the eSTREAM software portfolio. We analyze the cipher utilizing the fault model in which the attacker is assumed to be able to fault a random inner state word but cannot control the exact injected fault locations. Our attack, which recovers the secret inner state of the cipher, requires around 6144 faults, work equivalent to around 2^{48} SOSEMANUK iterations and a storage of around $2^{38.17}$ bytes.

Next, we present a differential fault analysis against Hummingbird. Hummingbird is a lightweight encryption algorithm that has a hybrid structure of block cipher and stream cipher with 16-bit block size, 256-bit key size, and 80-bit internal state. We analyze the cipher utilizing the fault model in which the attacker is assumed to be able to fault a random word before the linear transform, after the s-boxes, of the four block ciphers which are used in the Hummingbird encryption process but cannot control the exact location of injected faults. Our attack, which recovers the 256-bit key, requires around 50 faults and 2^{66} steps.

ZUC is a new stream cipher that was proposed for the 4G mobile standard by the Data Assurance and Communication Security Research Center of the Chinese Academy of Sciences. Our third contribution is a scan based cryptanalysis of ZUC. A scan path connects registers in a hardware circuit serially so that a tester can observe the register values inside the circuit. Scan-based attacks exploit the scan chains that are inserted into the devices for the purpose of testing. Under reasonable assumptions, our scan-based cryptanalysis allows the attacker to ascertain the whole location of internal registers including the LFSR and the memory cells of the cipher. The attack, which utilizes the key loading procedure and the working mode of the cipher execution procedure, allows the cryptanalyst to recover the

secret internal state of the cipher in a relatively small number of clock cycles.

Acknowledgments

I would like to appreciate everyone who made this dissertation possible.

First of all, I would like to express gratitude to my supervisor, Dr. Amr Youssef, whose encouragement and support in all academical stages paved the way for me. I thank him for sharing his knowledge, enthusiasm, time and ideas with me from the start to the end.

I would like to say a special *thank you* to my wife, Mona, for her patience. She helped me to concentrate on completing this dissertation and supported me faithfully during my endeavors. Nothing I can say can do justice to how I feel about your support, Mona.

I also wish to express my appreciation for my parents. Without their belief in my goals, their kindness and affection, I would not be able to enter the field of scientific research.

Finally, I wish to thank my colleagues at Concordia University. My keen appreciation goes to both Aleksandar Kircanski and Hossein Khonsari for their help and their productive research discussions.

Contents

List of Figures

List of Tables

List of Acronyms

AES	Advanced Encryption Standard
CRT	Chinese Remainder Theorem
DES	Data Encryption Standard
DFA	Differential Fault Analysis
DFT	Design For Testability
DPA	Differential Power Analysis
ECC	Elliptic Curve Cryptosystems
ECRYPT	European Network of Excellence for Cryptology
EDC	Error Detecting Codes
eSTREAM	ECRYPT Stream Cipher Project
FSM	Finite State Machine
GE	Gate Equivalent
LFSR	Linear Feedback Shift Register
LTE	Long Term Evolution
MAC	Message Authentication Code
NESSIE	New European Schemes for Signatures, Integrity and Encryption

NIST	National Institute of Standards and Technology
NLCG	Non-Linear Combiner Generator
NLFF	Non-Linear Filter Function
RBT	Redundancy Based Techniques
RFID	Radio Frequency Identification
SHA	Secure Hash Algorithm
SPA	Simple Power Analysis
SPN	Substitution Permutation Network
TEA	Tiny Encryption Algorithm

Chapter 1

Introduction

Protection and hiding of valuable information has a very old history [1] where cryptology has developed over the centuries from an art, in which only few were ingenious, into a science with well established foundations. There are several goals which security professionals desire to achieve through the use of cryptography. These goals include confidentiality, data integrity, entity authentication, non-repudiation and data origin authentication [2]. Cryptology encompasses two related fields: cryptography and cryptanalysis. Cryptography can be defined as the study of mathematical techniques to ensure various aspects of information security as those mentioned. On the other hand, cryptanalysis can be defined as the study of techniques to analyze, and break, information security services by targeting, specifically, the underlying cryptographic algorithms.

Cryptographic primitives, which can be seen as the basic building blocks of any cryptosystem, are extremely important in the view of cryptographers and attackers. While cryptographers design cryptosystems using these low-level building blocks, attackers attempt

to evaluate such blocks in order to compromise the overall security of deployed systems. Block ciphers, stream ciphers, hash functions, message authentication codes and public key cryptosystems are among the most fundamental primitives of cryptography.

Cryptographic algorithms can be divided into two basic classes: symmetric key (or secret key) systems and asymmetric key (or public key) systems. Both of these systems are used to provide a variety of security functions for networks and information systems. Whereas symmetric key cryptography uses the same key for encrypting and decrypting information, public key algorithms do not require a secure initial exchange of one or more secret keys between the sender and receiver. In symmetric key cryptography, confidentiality is provided by using stream ciphers or block ciphers. While stream ciphers encrypt the bits of the message one at a time, block ciphers take a number of bits and encrypt them as block units under a private key. The focus of this work is the cryptanalysis of symmetric key primitives.

1.1 Motivation

Generally, cryptanalysis of a cipher, or simply cipher breaking, does not mean finding an efficient algorithm for an adversary to recover the plaintext from the ciphertext. In particular, in the academic cryptanalysis literature, breaking a cipher mainly means finding a weakness in the cipher that can be exploited with a complexity strictly less than brute force. In other words, the cryptosystem is said to be broken if the effort required by the attacker to gain the secret information (e.g., plaintext or key) is less than the effort needed

by naive exhaustive key search. Based on this definition, which can be controversial, one may need unrealistic amounts of time, memory, or known/chosen plaintext-ciphertext pairs. In fact, most of the published pure mathematical attacks against modern ciphers belong to this category, i.e., these attacks require an unrealistic amount of resources.

However, in many practical scenarios, the adversary may have access to the cryptographic device. In such a case, side channel attacks allow the attackers to practically break these cryptosystems using relatively small amount of computational resources and a small number of known/chosen plaintext-ciphertext pairs. In particular, side channel attacks make use of the physical implementation of the cryptosystem and cover different models which increase the capabilities of the attackers. These capabilities include gaining side channel information about the encryption or decryption process such as timing analysis [3] and power analysis [4]. It may also include the ability to apply some kind of influence on the internal state of the cryptographic devices by using unsupported supply voltage or excessively overclocking the device. Strong electric or magnetic fields, or even ionizing radiation flipping random bit(s) in the internal registers of the hardware implementation may also be used to gain access to some faulty computations of the cryptographic devices which allow the recovery of secret internal information [5].

Currently, the wide spread of unprotected software or hardware cryptographic implementations offer various possibilities for these side channel attacks. Such attacks are practical and do not require expensive equipments. One class of side channel attacks that we will focus on throughout this work is differential fault analysis (DFA) [6]. DFA is a powerful

side channel attack which can be applied to various kinds of cryptographic devices including public key systems, block ciphers and stream ciphers. The basic idea behind DFA is to force the cryptographic device to produce some controlled incorrect output results which allow the attacker to deduce information about the secret internal state of the cryptographic device. This technique was first applied to RSA [6] and then generalized to other public key ciphers [6] and block ciphers such as DES [7] and AES [8]. Later on, DFA was also applied to stream ciphers [9], particularly against the finalist list of the eStream project including HC-128 [10] and Rabbit [11, 12].

In the first part of this work, we present a differential fault analysis on SOSEMANUK, which is a software-based stream cipher that has passed all three stages of the ECRYPT stream cipher project and is a member of the eSTREAM software portfolio.

Recently, dedicated ultra-lightweight symmetric key algorithms have been proposed for applications within low-cost resource constrained devices such as RFID tags, smart cards, and wireless sensor nodes. Obviously, security and privacy challenges should be considered in these devices as well. In particular, DFA technique is still one of the main cryptanalysis techniques that can be utilized to break theses cryptographic devices. Consequently, in the second part of this work, we apply DFA to cryptanalyze a relatively new lightweight encryption algorithm, Hummingbird, proposed by Engels *et al.* at FC'10 [13].

Design for testability (DFT) is a technique which has drastically improved the manufacturing testing with an efficient method of yielding a high fault coverage. Although the scan test causes DFT to be more successful, it can also be applied to assist attackers launch non-invasive attacks, which exploit information that is unintentionally leaked externally,

to recover important secret information from the cryptographic device. In general, this method can be applied to various kinds of cryptographic algorithms that are implemented in hardware. In the last part of this thesis, we present a scan-based cryptanalysis against a hardware implementation of the new stream cipher ZUC which was recently proposed for inclusion in the 4G Long Term Evolution (LTE) mobile standard [14, 15].

1.2 Contributions of the thesis

Our contributions can be summarized as follows:

- **Differential Fault Analysis of SOSEMANUK:** SOSEMANUK [16] is a software-based stream cipher which supports a variable key length between 128 and 256 bits and a 128-bit initial value. It has passed all three stages of the ECRYPT stream cipher project and is a member of the eSTREAM software portfolio. Our first contribution is a fault analysis attack on SOSEMANUK. The fault model in which we analyze the cipher is the one in which the attacker is assumed to be able to fault a random inner state word but cannot control the exact location of injected faults. Our attack, which recovers the secret inner state of the cipher, requires around 6144 faults, work equivalent to around 2^{48} SOSEMANUK iterations and a storage of around $2^{38.17}$ bytes.

- **Differential Fault Analysis of Hummingbird:** Hummingbird [13,17] is a lightweight encryption algorithm proposed by Engels *et al.* at FC'10. Unlike other lightweight cryptographic primitives, which can be classified as either block ciphers or stream ciphers, Hummingbird has a hybrid structure of block cipher and stream cipher with

16-bit block size, 256-bit key size, and 80-bit internal state. Preliminary analysis conducted by the cipher's designers shows that it is resistant to most common attacks against block ciphers and stream ciphers. Our second contribution is a differential fault analysis attack on Hummingbird. The fault model in which we analyze the cipher, is the one where the attacker is assumed to be able to fault a random word before the linear transform, after the s-boxes, of the four block ciphers which are used in the Hummingbird encryption process but cannot control the exact location of injected faults. Our attack, which recovers the 256-bit key, requires around 50 faults and 2^{66} steps.

- **Scan Based Side Channel Attack of ZUC:** ZUC [14, 15] is a relatively new stream cipher that was proposed for the 4G mobile standard by the Data Assurance and Communication Security Research Center of the Chinese Academy of Sciences. Our third contribution is a scan based cryptanalysis against ZUC. Under reasonable assumptions, our cryptanalysis allows the attacker to ascertain the whole location of internal registers of the LFSR and the memory cells.

The above results are partially published in [18] and [19].

1.3 Thesis Outline

The rest of the thesis is organized as follows. The next chapter introduces the required background and the literature review of the side channel attacks. Our differential fault analysis of SOSEMANUK is presented in chapter 3. Chapter 4 includes our differential fault

analysis of Hummingbird. The scan based cryptanalysis of ZUC is presented in chapter 5. Finally, chapter 6 provides the conclusions and future work.

Chapter 2

Cryptanalysis of Symmetric Key Encryption Algorithms

Symmetric key encryption algorithms can be classified as either block ciphers or stream ciphers. In this chapter, we first provide a brief introduction to block ciphers and stream ciphers. We then present an overview of different cryptanalytic attacks against these ciphers.

2.1 Block ciphers

Symmetric key block ciphers are among the most prominent elements in modern cryptography. As a fundamental building block, their versatility allows the construction of pseudo random number generators, stream ciphers, MACs, and hash functions [2]. Individually, they provide confidentiality under a secret parameter called private key or secret key.

Block ciphers operate on fixed length groups of bits, called blocks, typically of sizes

ranging between 64 to 256 bits. The Data Encryption Standard (DES) [20], which was developed in the 1970s by IBM, is an example for a symmetric key block cipher which encrypts 64-bit data blocks under the control of a 56-bit key. The DES decryption is the inverse of DES encryption and uses the same key. In 2001, the Advanced Encryption Standard (AES) [21], with a 128-bit block length and 128-256 bit key length, was approved by NIST as a replacement for DES.

Block ciphers are constructed by combining basic building blocks which consist mainly of linear transformations or permutations, non-linear functions such as s-boxes, and modular addition. Usually, these building blocks are combined in units called rounds. The Substitution Permutation Network (SPN) structures and Feistel (also referred to as DES-like) structures are the two most commonly used designs for block cipher constructions [2].

2.1.1 Resource constrained block ciphers

Due to the tight cost and constrained resources of high volume consumer devices such as RFID tags, smart cards and wireless sensor networks, it is desirable to employ lightweight and specialized cryptographic primitives for many security applications. Several resource constrained devices have limited capabilities in every aspect of computation, communication and storage. To secure these devices, lightweight cryptographic primitives are needed.

The gate constraints for security of low-cost tags are about 200-2000 gates which is less than what is usually required by standard cryptographic primitives. Thus existing cryptographic algorithms can be hardly implemented under such resource constraints. The

resources required for AES are around 3600 Gate Equivalent (GE). While the exact implementation requirements for the primarily constrained resource algorithms, Tiny Encryption Algorithm (TEA) [22], are not known, a crude estimate is that TEA needs at least 2100 GE and XTEA [23] needs at least 2000 GE. Some of the most extensive proposals for low-cost implementation are mCrypton [24], HIGHT [25], SEA [26], CGEN [27], PRESENT [28], MIBS [29], and Hummingbird [13] (CGEN and Hummingbird are not classified as pure block ciphers). The required gates for the hardware implementation of some of these encryption algorithms are 2949 GE for mCrypton , 3048 GE for HIGHT, 2280 GE for SEA, 1570 GE for PRESENT and 1396 for MIBS.

2.2 Stream ciphers

Stream ciphers provide another alternative for block ciphers in applications requiring symmetric key encryption. Generally, compared to block ciphers, stream ciphers are preferred in software applications with very high throughput requirements, and in hardware applications with restricted resources such as limited storage, gate count, or power consumption.

Stream ciphers can be though of as pseudorandom number generators that are initialed by secret keys and, usually known, Initial Values (IVs). The encryption process is performed by combining the plaintext with the produced key stream, usually through an XOR operation.

Unlike the case for block ciphers, currently there is no specific standard for stream

ciphers. However, two main projects are worth mentioning. The first one, the New European Schemes for Signatures, Integrity and Encryption (NESSIE), is a European research project funded from 2000 to 2003 [30]. At the end of this project, none of the submitted stream ciphers were selected because all of them were attacked. In 2004, a call for another competition of stream ciphers proposals was issued by the Network of Excellence within the Information Societies Technology (IST) Programme of the European Commission [31]. Thirty five stream cipher algorithms were submitted to the project, known as the eSTREAM [32], at three profiles: Profile I (Software), Profile II (Hardware) and Profile I+II (Software+Hardware). In 2008 and after three phases, four algorithms were selected as Profile I and three stream ciphers were announced as Profile II as follows:

- Profile I (SW): HC-128 [33], Rabbit [34], Salsa20/12 [35] and SOSEMANUK [16].
- Profile II (HW): Grain [36], MICKEY [37] and Trivium [38].

Stream ciphers can be classified as synchronous and self-synchronizing or asynchronous. A synchronous stream cipher is one in which the keystream is generated independent of the plaintext message and of the ciphertext. Synchronization requirements, and no error propagation are properties of synchronous stream ciphers. An asynchronous stream cipher is one in which the previous ciphertext digits participate in computing the next keystream word. Consequently, the keystream generated by an asynchronous stream cipher algorithm is a function of the key and a fixed number of previous ciphertext digits. Some properties of asynchronous stream ciphers are self-synchronization, limited error propagation, active attacks and diffusion of plaintext statistics.

Stream ciphers can also be classified with respect to their design components into shift register based and non shift register based stream ciphers. RC4 [39], HC-128 [33] and HC-256 [40] are examples for non shift register based steam ciphers. Shift register based ciphers can be further divided into Linear Feedback Shift Register (LFSR)-based and Non-Linear Feedback Shift Register (NLFSR)-based structures. Due to inherent linearity of the output of LFSRs sequences, its direct application in cryptography is restricted although its produced sequences may have several good properties such as long period, balancedness and good correlation properties. To eliminate this inherent linearity in LFSRs based stream ciphers, one can use more than one LFSR and utilize a Non-Linear Combiner Generator (NLCG) to remove the linearity of the produced sequence in a regularly clocked LFSR. The nonlinear combining functions are required to have cryptographic properties such as balance, high nonlinearity, correlation immunity, high algebraic degree and high algebraic immunity degree [42] to ensure that the output of the stream cipher is secure. Another method to improve the nonlinearity of the output sequence is through the use of Non-Linear Filter Function (NLFF) which operates on a subset of the bits of the LFSR. Irregularly clocked LFSRs were also proposed to improve the nonlinearity of stream ciphers [41]. In this case, the underlying structure has more than one LFSR where each of them is clocked at a different rate, independent of the others, at each step of the cipher iterations.

2.3 Cryptanalysis of symmetric ciphers

Cryptanalysis is the study of techniques which attempt to defeat information security services by compromising the underlying cryptographic schemes. Cryptanalysis of symmetric key ciphers typically involves looking for attacks against block ciphers, stream ciphers and MACs. There is a wide variety of cryptanalytic attacks and they can be classified by several ways.

Based on the nature of the adversary, cryptanalytic techniques can be classified into either passive attacks or active attacks. A passive attack is one where the adversary only monitors the communication channel which only threatens the data confidentiality. On the other hand, in an active attack, the attacker may attempt to delete, add, or change the transmission on the channel. Data integrity, authentication and confidentiality are threatened by this type of adversaries.

Another classification is based on the information available to the adversary. In this case, cryptanalytic attacks can be classified as ciphertext-only attacks, known-plaintext attacks, chosen-plaintext attacks, chosen-ciphertext attack, adaptive chosen-plaintext attacks and adaptive chosen-ciphertext attacks [2]. In a ciphertext-only attack, the cryptanalyst has access only to the ciphertext and tries to recover the key or plaintext whereas in a known-plaintext attack, the cryptanalyst has access to a ciphertext and its corresponding plaintext. A chosen-plaintext attack is a cryptanalysis form in which the adversary may

choose a plaintext and learn its corresponding ciphertext while in a chosen-ciphertext attack, the attacker can choose ciphertexts and learn their corresponding plaintexts. In adaptive chosen-plaintext attack, the cryptanalyst makes a series of interactive queries, choosing subsequent plaintexts based on the information from the previous encryptions. Finally, an adaptive chosen-ciphertext attack is an interactive model of chosen-ciphertext attacks in which an attacker sends a number of ciphertexts to be decrypted, then uses the results of these decryptions to select subsequent ciphertexts.

There is another classification of attacks with respect to whether the attacker has some sort of physical access to the encrypting device or not. Pure mathematical attacks are traditional cryptanalytic techniques that rely only on known or chosen input-output pairs of the encryption function, and exploit the inner structure of the cipher to reveal secret key information. On the contrary, in side channel attacks, it is assumed that the attacker has some access to the encryption device, either by being able to make measurements with respect to time or power consumption, or by being able to induce errors in the memory of the device (fault analysis). The additional information gained by utilizing such a side channel is then combined with methods that exploit the inner structure of the cipher to reveal the secret key.

The success of a cryptanalytic attack is typically measured by the following complexities:

- *Data complexity*: The amount of plaintext/ciphertext information necessary to perform the attack.

- *Time complexity*: The amount of necessary computations required to execute the attack. For example, in the case of a brute force attack in which every key is trivially examined, the number of operations is $2^{|K|-1}$ on average, where $|K|$ denotes the size of the key space in bits.

- *Memory complexity*: The amount of storage required by the algorithm that executes the attack.

- *Number of necessary physical actions on the encrypting device*: This measure is relevant only to side channel attacks and can include the number of necessary measurements in case of side channel analysis (such as power analysis attacks and timing attacks) or number of induced faults in the memory of the cipher, in case of fault analysis.

2.3.1 Pure mathematical attacks of symmetric ciphers

In pure mathematical attacks, the adversary regards the problem as how to recover the secret key given input/output pairs of the encryption algorithm from a purely mathematical perspective without considering the physical implementation of the cipher.

There are various types of pure mathematical cryptanalysis models against symmetric key ciphers. However, a set of generic cryptanalytic methods exists regardless whether the cryptographic algorithm is a stream cipher or a block cipher. Brute force or exhaustive key

search is the most trivial generic cryptanalytic method which can be applied to such algorithms, independent of the design details. Besides, many other cryptanalytic methods investigate weaknesses of symmetric algorithms. In what follows, the intuition behind some of the well-known cryptanalysis methods which are applicable to symmetric primitives is given.

Linear cryptanalysis [43] is a known plaintext attack that utilizes the existence of any linear relation, between some plaintext and ciphertext bits, that holds with probability different than $\frac{1}{2}$. Linear cryptanalysis was successfully applied by Matsui against DES in 1993. Later on, the attack was widely applied to many other block ciphers with different degrees of success. The first step of linear cryptanalysis is to obtain a linear approximation for the nonlinear blocks (e.g., s-boxes or non-linear combining functions). In this step, the attacker utilizes approximations that hold with a large bias. The second step is to propagate the achieved approximation throughout the other component of the cipher in order to achieve an overall probabilistic linear relation that involves the plaintext, ciphrtext and the key bits (as the only unknowns).

Differential cryptanalysis [44] is a general cryptanalytic method applicable primarily to block ciphers, but was also applied recently to stream ciphers. In a very broad sense, differential cryptanalysis studies how specific differences in the input of a particular transformation affect the resulting output differences. As in linear cryptanalysis, the first step of differential cryptanalysis is to find differential characteristics that hold with relatively good probability for the different building blocks of the cipher. Then, these characteristics are concatenated to form a differential for the overall cipher. Consequently, possible key

values can be recovered from the desired output difference between two chosen or known plaintext inputs.

Truncated differential cryptanalysis [45], impossible differential attacks [46], higher-order differential cryptanalysis [47], and Boomerang attacks [48] are some well known extensions of differential cryptanalysis.

Other attacks on block ciphers, independent of linear and differential cryptanalysis, include the interpolation attacks [49], related key attacks [50], square attacks, integral attacks, and multiset attacks [51,52]. Examples of dedicated cryptanalysis techniques against stream ciphers include the correlation attacks [53] and the guess and determine attacks [54].

For modern stream and block ciphers, the above examples of pure mathematical attacks are interesting, mainly, from a theoretical perspective but they typically require an overwhelming computation and/or data complexity.

2.3.2 Side channel attacks

Side channel attacks concentrate on how to utilize the information leaked from physical implementations of cryptographic modules during execution of the algorithm. Implementation dependent attacks, i.e., side channel attacks, present a serious threat for many applications of symmetric key primitives which are widely deployed in many devices such as TV set-top boxes, prepaid cards and smart cards.

In the side channel attacks, the cryptanalyst is assumed to have some physical access to the particular device that performs the encryption. Certain parameters such as the instantaneous power consumption of the cryptographic device or the time used to perform

the encryption operation can be measured. In some of these attacks, the attacker is also assumed to be able to induce errors in the memory of the device or at a particular step of the computation process of the device (fault analysis). In other words, the leakage of information can be extracted by analyzing timing measurements, power consumption or electromagnetic radiations. Besides, other forms of side channel information can be available as a result of hardware or software failures which can be cleverly introduced into the cryptographic device by changing the operating frequency or temperature beyond the allowed limits or by other dedicated methods of fault injection. By utilizing this side channel information, the attacker might be able to deduce some information about the encrypting process which leads to recovering the key.

Anderson *et al.* [55] categorize side channel attacks into the following four classes:

- *Invasive Attacks*: These attacks require a direct physical access to the internal elements of the cryptographic modules. For instance, the attacker may reach the layer of the cryptographic module and put a microprobing needle on a data bus to record, and later analyze, the data transfer. Several defensive measures are usually implemented in hardware to efficiently limit invasive attacks. For example, if tampering is detected, some cryptographic modules with higher security level reset all their memories [56].

- *Semi-invasive Attacks*: In these attacks, the adversary can access the device but without inducing a physical damage to the chip or making unauthorized electrical interface connection. For example, in fault analysis attacks, the attacker may utilize a

laser beam to ionize the device to change some of its memory data and finally alter the device output [5].

- *Local non-invasive Attacks*: In this class of attacks, the cryptanalyst needs close investigation or manipulation of the device's operation. For example, from the power analysis point of view, the attacker can observe the current drawn by the processor precisely. Then, the cryptographic keys can be recovered by means of measuring the correlation of the mentioned parameter with the computations being performed by the device.

- *Remote attacks*: These attacks require only observation or manipulation of the device's normal input and output. There are various types of attack that are independent of the distance between the attacker and the cryptographic device. Timing analysis, protocol analysis and attacks on application programming interfaces belong to this group.

Each of these types of attacks may also be classified as passive or active depending on assumptions regarding the control executed by the attacker over the computation process. In passive attacks, the normal operations of the device are not affected, i.e., the adversary can collect information about the operation of the target system without disturbing its normal task. In active attacks, the attacker can interfere with the device inputs or environment to change the normal operation while the target system may or may not be able to detect such an influence.

Among various kinds of side channel attacks, timing attacks [3], power analysis attacks

[4] and fault attacks [6] are the three most well-known and widely studied attacks. Other types of side channel attacks include electromagnetic attacks [57,58], acoustic attacks [59] and cold boot attacks [60,61].

Timing attacks, introduced in 1996 by Kocher [3] against RSA, are the first type of modern side channel attacks presented in the open academic literature. The basic idea of the timing attack comes from the fact that typical implementations of cryptographic algorithms execute the computations in a non-fixed time. Whenever these operations involve secret parameters, they should be considered as a potential risk because these timing variations can leak some useful information about the secret parameters. By careful study of the obtained timing statistics, one can recover these secret parameters. A common method to prevent timing attacks is that all operations should be designed to take the same time duration. In case this is not achievable, some rough timing disturbance can be applied by introducing random timing shifts and wait states or by adding dummy instructions. To increase the number of ciphertexts required by the adversary, random delays can be added to the processing time. Generally, the number of required samples increases approximately as the square of the timing noise [3].

Power analysis attacks are another powerful form of side channel attacks which utilize the correlation between the power consumption of cryptographic devices and the secret parameters used in the cryptographic computations performed by these devices. These attacks, which were proposed in 1998 by Kocher *et al.* [4], can non-invasively extract secret information, such as cryptographic keys, from the device by measuring the instantaneous

power consumption from the running cryptographic operations that involve the desired secret parameters. Unlike the timing analysis, power analysis attacks are mainly applicable to hardware implementations. There are two general types of power analysis attack: Simple Power Analysis (SPA) and differential power analysis (DPA). In SPA attacks, the attacker tries to guess which particular instruction is being carried out at a specific time from the measured power traces as well as the input and output values of this instruction. In contrast to SPA, DPA is a more advanced form of power analysis that needs no knowledge of implementation details. In DPA attacks, the adversary computes the intermediate values within cryptographic computations by statistically analyzing data collected from multiple cryptographic operations. To alleviate power analysis attacks, one can modify the design of the hardware device to randomize its power consumption or to equalize the power consumption of all operations to make it independent of the processed secret values. Another commonly used countermeasure method against power analysis attacks is the data masking technique, which can be applied at the software or hardware level [62, 63].

In fault analysis attacks, the cryptanalyst applies some kinds of physical influence, such as ionizing radiation, on the internal state of the cryptosystem which influence the cryptographic primitive execution or memory. By carefully studying the results of computations performed under such faults, the attacker can retrieve information about the secret key. In 1996, Boneh *et al.* [6] introduced fault analysis by describing an attack that targets the RSA public key cryptosystem and exploits a faulty Chinese Remainder Theorem computation to factor the modulus n. Subsequently, fault analysis attacks were extended to symmetric

systems such as DES [7] and later to AES [8]. Fault analysis attacks became a more serious threat after cheap and low-tech methods of applying faults were presented [5]. Fault attacks against stream ciphers were introduced by Hoch *et al* [9], where attacks against LILI-128 and SOBER-t32 and RC4 were described. Other stream ciphers that were analyzed in the fault analysis model include SNOW 3G [64], Trivium [65], HC-128 [10] and Rabbit [11, 12].

The number of required faults in the above fault attacks varies depending on the assumed fault analysis model. In general, all models follow the one given in Armknecht *et al.* [66], which assumes that the attacker has access to the physical device, and that the attacker is able to reset the device to the same unknown initial settings as often as needed. However, different assumptions with respect to the amount of control the attacker has over the induced faults are utilized. For example, the attacker may have control over the location of the faulted memory register, or may be able to restrict the Hamming weight of the induced faults. For instance, Biham *et al.* [67] assumed a model in which the attacker can choose the exact location (register) of the fault which causes RC4 to enter a special inner state and makes its recovery a trivial task. Similarly, Armknecht *et al.* [66] described a fault analysis attack against SNOW 2.0 where they assumed that the fault occurs exactly in a particular register of the cipher. On the other hand, in the fault analysis of Trivium [65], it is assumed that the attacker has no control or knowledge over the fault position. Different assumptions also exist regarding the Hamming weight of induced faults. For instance, in [10], it is assumed that the fault causes a 1-bit flip in the inner state of the cipher, whereas in [67], it is assumed that the fault is localized in one byte of the inner state.

Fault analysis attacks have also been applied to several block ciphers [7, 68–70]. The basic idea of the differential fault attack against SPN-based block ciphers is to use the diffusion property of the last linear transformation layer in order to determine whether the difference before the last nonlinear layer possibly originates in a fault or not. In particular, the adversary induces a fault as a differential input of the last linear transform and looks at the corresponding differential output. This provides the attacker with a distinguishing criteria for the last round key. More details on this class of attacks can be found in [69].

To secure cryptographic devices against fault analysis, proper countermeasures have to be applied. Generally, these countermeasures try to detect any temporal or permanent faults which happen in the cryptosystem, and then, immediately, disable the device output or rest all the output bits to 0s. As a result, the attacker will be prevented from observing the output of the faulty cryptographic computations and hence the vulnerability of the cryptosystem to these attacks can be alleviated. Several approaches of fault detection techniques have been investigated. These techniques include error detecting codes (EDCs) and redundancy-based techniques (RBT) [71, 72].

In addition to the above mentioned side channel attacks and models, another technique, called scan based side channel attack, was recently introduced to recover secret keys from hardware implementation of cryptographic devices that are designed with some built-in testability features. In the proposed model [73], the attacker first locates all the scan elements of the scan chain by scanning out the internal state in the test mode after loading pairs of known plaintexts with one-bit difference in the normal mode. In [73], the secret key was determined by using the structure of the DES s-boxes and three additional plaintexts. This

attack was also applied to stream ciphers [74, 75].

The flipped scan technique [76] was proposed to mitigate scan based side channel attacks. In this technique, inverters are introduced at random points in the scan chain. The authors in [76] claimed that the required security is reached when the inverter positions cannot be guessed with a probability significantly greater than $\frac{1}{2}$. However, Agrawal *et al.* showed that this technique was vulnerable to the reset attack [74]. To prevent this attack, they also proposed another protection mechanism, called the XOR-chain, in which XOR gates are inserted at random points in the scan chain [74]. In this scheme, each XOR gate acts as a data-dependent inverter which conditionally changes the current input of the flip-flop based on the preceding one. As a result, the new method using XOR chains has been shown to provide a good level of resistance against known scan based attacks without compromising testability [74].

Chapter 3

Differential Fault Analysis of SOSEMANUK

3.1 Introduction

SOSEMANUK [16] is a fast software-oriented stream cipher that has passed all the three phases of the ECRYPT eSTREAM competition and is currently a member of the eSTREAM Profile 1 (software portfolio). It uses a 128-bit initialization vector and allows keys of either 128-bit or 256-bits, whereas the claimed security is always 128-bits. The design of SOSEMANUK (See Figure 2) is based on the SNOW2.0 stream cipher [77] and utilizes elements of the Serpent block cipher [78]. SOSEMANUK aims to fix weaknesses of the SNOW 2.0 design and achieves better performance, notably in the ciphers initialization phase. Also, the secret inner state of SOSEMANUK is reduced when compared to SNOW 2.0 and amounts to 384 bits.

The preliminary analysis [16], conducted during the SOSEMANUK design process, includes the assessment of the cipher with respect to different cryptanalytic attacks such as correlation attacks, distinguishing attacks and algebraic attacks. Public analysis followed and SOSEMANUK was assessed in [79] by Ahmadi *et al.* where a guess-and-determine attack requiring 2^{226} operations and 2^4 keystream words was provided. Another improved guess-and-determine attack was presented by Tsunoo *et al.* in [80]. A correlation attack on SOSEMANUK was presented by Jung-Keun Lee *et al.* [81] with a computational complexity of $2^{147.88}$ and success probability 99% to recover the initial secret inner state. The data requirement for the attack was relaxed by Cho *et al.* [82]. In 2009, Lin *et al.* [83] improved the guess-and-determine attack, achieving complexity of 2^4 word keystream using 2^{192} steps. Another guess-and-determine attack with time complexity 2^{176} was recently presented by Feng *et al.* in Asiacrypt 2010 [84].

In this chapter, we present a fault analysis attack on SOSEMANUK. The fault analysis model adopted in the chapter is the one in which the attacker is assumed to be able corrupt a random inner state register in between the iterations of the cipher but the attacker has no control or knowledge over which inner state register has been corrupted. Also, the attacker is assumed to be able to reinitialize the cipher with the same key and IV arbitrary number of times. The attack recovers the secret inner state without recovering the key and requires about 6144 faults, 2^{48} operations each equivalent to one SOSEMANUK iteration and the storage of about $2^{38.17}$ bytes.

The rest of the chapter is organized as follows. In the next section, we provide a brief overview of fault analysis attacks. In section 3.2, relevant details of SOSEMANUK are

reviewed. An overview of the proposed attack is provided in section 3.3. Details of the attack are described in section 3.4 and section 3.5. Finally, the conclusion is given in section 5.5.

3.2 The SOSEMANUK specifications

The following notation will be utilized throughout the rest of the chapter:

- x^i: i-th bit of an n-bit word x
- $\boxplus, \times$: addition and multiplication modulo 2^{32}, respectively
- $\oplus$: bit-wise XOR
- $<<<$: left rotation defined on 32 bit values
- $|$:concatenation
- $X_i = f^i_{t+3}|f^i_{t+2}|f^i_{t+1}|f^i_t$: input value for i-th s-box applied in the Serpent1 function at some step t (the t value will be clear from the context). The Serpent1 function, shown in Figure 1, is defined by 32 applications of S in the bit-slice mode, where

$$S = [8, 6, 7, 9, 3, 12, 10, 15, 13, 1, 14, 4, 0, 11, 5, 2]$$

  is the s-box used in the third s-box layer of the Serpent block cipher [78].

- ´: Sign for denoting faulty cipher registers or output. For example s'_0 will denote the LFSR register s_0 in the faulty instance of the cipher.

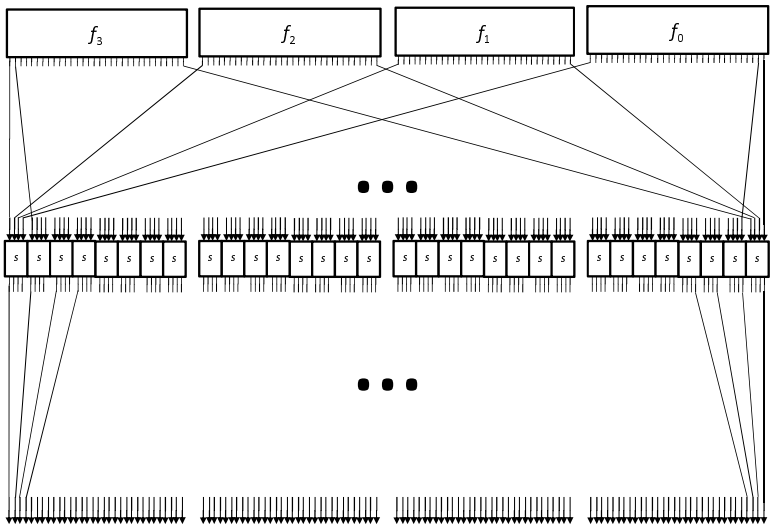


Figure 1: The Serpent1 function

While the claimed security level of SOSEMANUK is 128 bits, it supports a variable key length of 128 or 256 bits and 128 bit initialization value. As depicted in Figure 2, the secret inner state of SOSEMANUK consists of 12 32-bit words $(s_0, \ldots, s_9, R1, R2)$ and utilizes three main components to generate the keystream output: a linear feedback shift register (LFSR), a finite state machine (FSM) and an s-box-like function, Serpent1. To update the LFSR, the following recurrent relation is applied:

$$s_{t+10} = s_{t+9} \oplus \alpha^{-1} s_{t+3} \oplus \alpha s_t \tag{1}$$

where α is a root of the primitive polynomial $P(X) = X^4 + \beta^{23} X^3 + \beta^{245} X^2 + \beta^{48} X + \beta^{239}$ over GF(2^8) and β ia a root of the primitive polynomial $Q(X) = X^8 + X^7 + X^5 + X^3 + 1$ over GF(2).

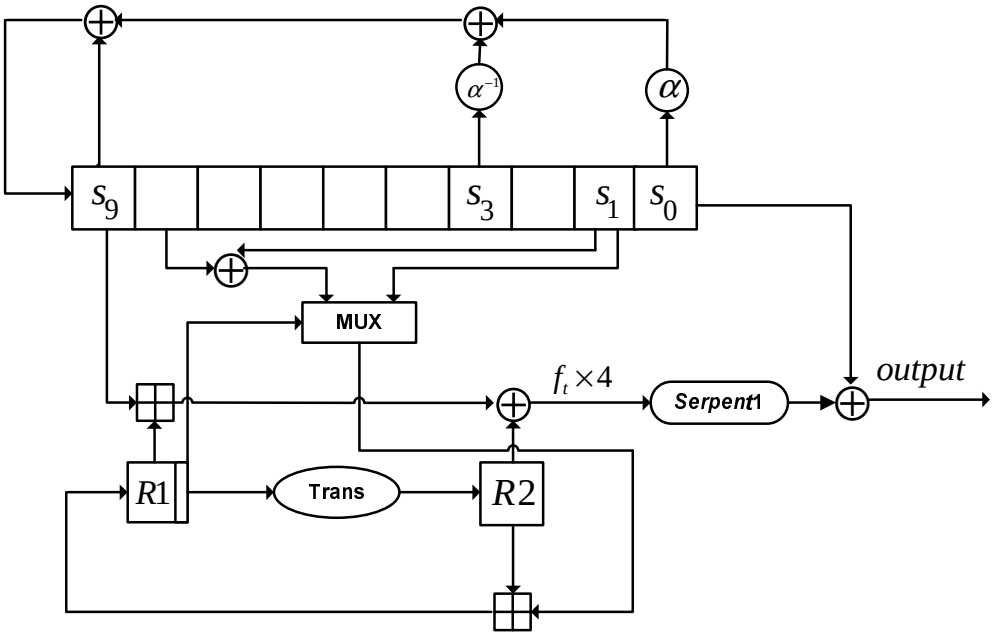


Figure 2: Overview of the SOSEMANUK stream cipher

The FSM update procedure is defined as follows:

$$R1_{t+1} = (R2_t \boxplus mux(lsb(R1_t), s_{t+1}, s_{t+1} \oplus s_{t+8})) \quad (2)$$

$$R2_{t+1} = (Trans(R1_t)) \quad (3)$$

where $mux(c, x, y) = \begin{cases} x & if\ c = 0 \\ y & if\ c = 1 \end{cases}$, *Trans*(*x*)=(M $\times$ *x*) <<< 7 and $M = 0x54655307$.

The FSM output at each step is defined by

$$f_t = (s_{t+9} \boxplus R1_{t+1}) \oplus R2_{t+1} \quad (4)$$

The inner state right after the initialization is denoted by $(s_0, \ldots, s_9, R1_0, R2_0)$. At each step, first the FSM is updated and the f_t and s_t values are preserved in the internal buffer, then the LFSR is updated. Once every four steps, a 128-bit word is generated by

$$z_t|z_{t+1}|z_{t+2}|z_{t+3} = Serpent1(f_t|f_{t+1}|f_{t+2}|f_{t+3}) \oplus s_t|s_{t+1}|s_{t+2}|s_{t+3}. \quad (5)$$

For a more detailed description of SOSEMANUK, the reader is referred to [16].

3.3 The attack overview

In this section, we provide a high level overview of the proposed attack. According to our fault analysis model, the attacker is assumed to be able to re-initialize the cipher an arbitrary number of times. Furthermore, while we assume that each induced fault corrupts only one of the 12 inner state registers, the attacker does not know, and cannot control the position or the new value of the faulted register.

3.3.1 The main idea

The main idea of the attack can be explained as follows. In every SOSEMANUK iteration, 32 s-boxes are applied in the bit-slice mode as a part of the Serpent1 function. The first part of the attack restricts the input for each of the s-boxes by considering faults that occur at s_5 and s_4. Consider the case where the fault has been injected right after the SOSEMANUK initialization step and that it occurred in the register s_5. During the next cipher iteration

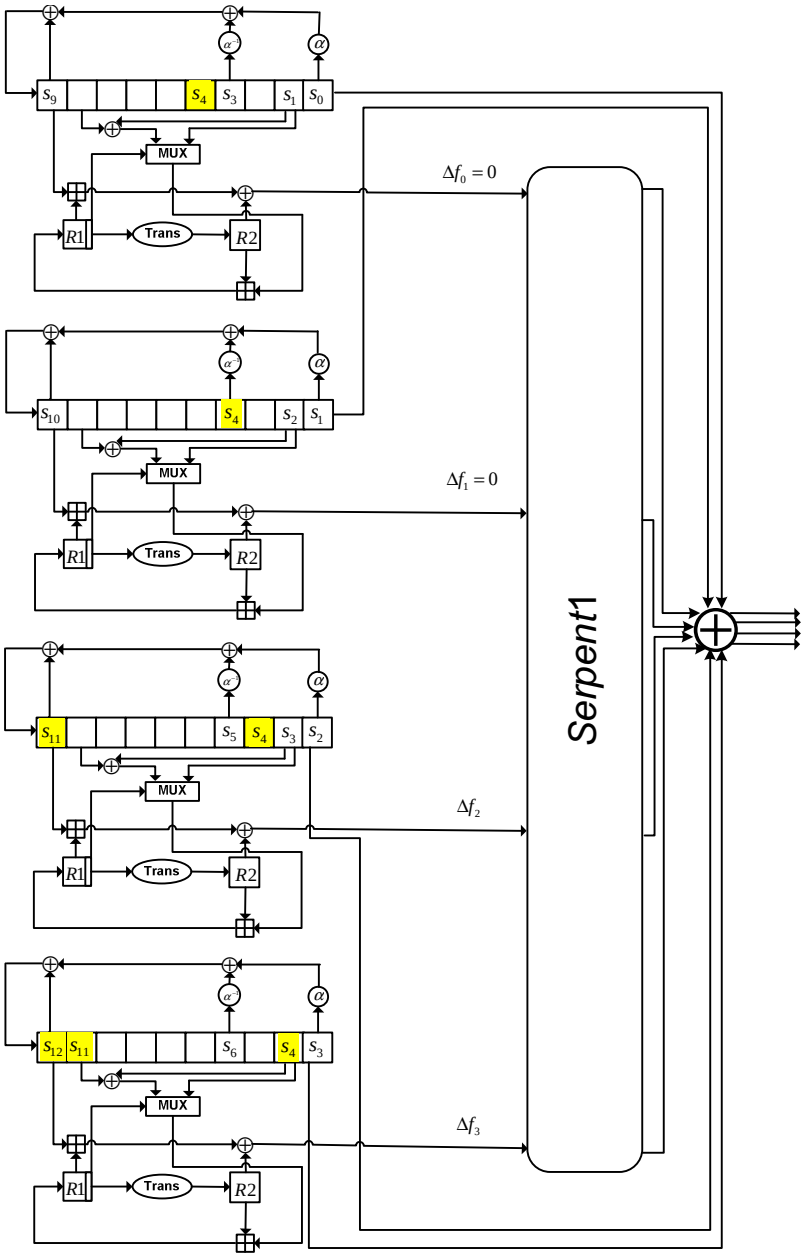


Figure 3: The Δf values corresponding to the case where s_4 is faulted

in which the $z_0|z_1|z_2|z_3$ 128-bit keystream word is produced, the fault moves in the right-hand direction as the LFSR is clocked for 4 times. In particular, no faulty values participate in generation of f_0. Furthermore, since in every step, first the FSM is updated and then the f_t value is computed and finally the LFSR is clocked, f_1 and f_2 are computed without error and the fault affects only f_3. Now the non-faulty f_0, f_1, f_2 and the faulty f_3 enter the Serpent1 function. In the bit-slice mode, the Serpent1 function applies 32 s-boxes 4-bit inputs, where i-th bit comes from register f_i, $i = 0, \ldots 3$ (See Figure 1). Thus, the input difference of all activated s-boxes will be equal to $0x8$ (1000 in binary). The attacker can then retrieve the corresponding s-box output difference and restrict the set of candidates for the s-box input-output values. When the fault occurs at register s_5, each s-box output will be faulted with probability $\frac{1}{2}$, which allows us to establish a criterion to recognize faults in register s_5. Similarly, in the case where the fault occurs at s_4, it propagates as shown in Figure 3 potentially affecting only f_2 and f_3. In other words, only the two most significant bits of every s-box input might be affected. Since a criterion for recognizing faults at s_4 can also be established, observing the output s-box differences for such faults also reduces the set of candidates for the s-box input-output values.

After the candidates for the s-box input-output values have been restricted, equation (5) is used to provide a restriction on the LFSR registers. From (1), it follows that the LFSR registers are not independent and restrictions on the LFSR registers can be coupled with the dependence of the LFSR registers to further prune the candidates for the s_t values. Finally, a guess and determine attack is used to find the rest of the inner state.

3.3.2 The steps of the attack

The attack can be divided into two phases. The first phase collects faulty output in four different steps of the cipher execution and can be summarized as follows:

- For $l \in \{0, 1, 2, 4\}$
    - Repeat the steps below for m times
        - Reinitialize the cipher
        - Iterate for l times
        - Induce a fault, corrupting a random inner state register
        - Collect and store the keystream output word $z'_{4l}|z'_{4l+1}|z'_{4l+2}|z'_{4l+3}$

The second phase, which uses the collected information to uniquely determine the secret inner state, can be summarized as follows:

(1) Use the faulty outputs gathered in the first phase of the attack for $l \in \{0, 2, 4\}$ to reduce the number of candidates for (s_0, s_1, s_2, s_3), $(s_8, s_9, s_{10}, s_{11})$ and $(s_{16}, s_{17}, s_{18}, s_{19})$ to 2^{32} each. Then, use dependencies between the three fourplets imposed by relation (1) to further reduce the corresponding numbers of candidates (details are explained in section 3.4)

(2) Similar to the previous step, using the information collected in the first phase of the attack for $l = 1$, reduce the number of candidates for (s_4, s_5, s_6, s_7) to 2^{32} (details are explained in section 3.4)

(3) Apply the guess-and-determine strategy through the space reduced sets of candidates obtained by previous two steps to recover the complete inner state (details are explained in section 3.5)

In the first phase of attack, data is collected for $l = 4$ and not for $l = 3$ since the LFSR registers candidate sets due to $l = 0$, $l = 2$ and $l = 4$ are correlated and allow further reduction. The reduction due to $l = 1$ is used later in the guess-and-determine attack.

3.4 Reducing the number of candidates for LFSR registers (s_0, s_1, s_2, s_3) and $(s_8, s_9, s_{10}, s_{11})$

The starting number of candidates for the LFSR registers (s_0, s_1, s_2, s_3) and $(s_8, s_9, s_{10}, s_{11})$ is 2^{128} each. In this section, first we show how to reduce this number to 2^{32} and then, by exploiting the fact that the two register components are linked by relation (1), reduce it further to 2^{16}, each.

3.4.1 Recovering the s-box differences

Let SOSEMANUK be in state $t = 0$. From (5) and since $z_0|z_1|z_2|z_3$ is accessible to the attacker, it is evident that reducing the uncertainty for $f_0|f_1|f_2|f_3$ leads to reducing the uncertainty of $s_0|s_1|s_2|s_3$. In this subsection, the $f_0|f_1|f_2|f_3$ value is constrained by calculating the s-box input-output differences using the faulty information. Since the algorithms below are also applied to constraint $f_4|f_5|f_6|f_7$, $f_8|f_9|f_{10}|f_{11}$ and $f_{16}|f_{17}|f_{18}|f_{19}$, these algorithms are specified for general time t and will be used for $t \in \{0, 4, 8, 16\}$.

Define δ_i and Δ_i by

$$\delta_i = S(X_i \oplus \text{0x8}) \oplus S(X_i),$$

$$\Delta_i = \{S(X_i \oplus \text{0x4}) \oplus S(X_i), S(X_i \oplus \text{0x}c) \oplus S(X_i)\}$$

for every $i = 0, \ldots 31$. Algorithm 1 and Algorithm 2, described below, are used to recover δ_i and Δ_i, respectively, for each $i = 0, \ldots 31$.

In what follows, the probability distribution of the number of non-activated s-boxes in the SOSEMANUK output is analyzed. In particular, probabilities of the event that there will be more than 16 non-activated s-boxes are estimated under different assumptions about the location of the fault. For that purpose, let $0 \leq n \leq 32$ be a random variable which denotes the number of s-boxes that are *not* active in the application of the 32 s-boxes of Serpent1 in some steps of a faulty SOSEMANUK instance. Consider for example the probability that a particular s-box will not be activated given that the fault has occurred at s_0. In that case, only the 3 most significant bits of the s-box input may be corrupted. Note that, due to (5) by which the corrupted s_0 is XOR-ed to the least significant bits of each s-box, it may also happen that the difference in the s-box output caused by the s-box input cancels out. However, such a possibility has been ruled out by exhaustively checking that for each s-box input value it is not possible to cause a difference only in the least significant bit of the s-box output by any of the differences in the 3 most significant bits of the input. Thus, the probability that the particular s-box has not been activated is 2^{-3}. Now, it is clear that variable $n \sim B(2^{-3}, 32)$, i.e., n follows binomial distribution with parameters $p = 2^{-3}$ and

$n = 32$. According to the binomial distribution, $P[16 \leq n \leq 31] = \sum_{i=16}^{31} \binom{32}{i} p^i (1 - p)^{32-i} \approx 2^{-21}$. More generally, the distribution of n in terms of the fault position is given follows:

- $\{s_0\}$: $P[16 \leq n \leq 31] \approx 2^{-21}$ as explained above.

- $\{s_1, s_9, R1, R2\}$: all four s-box input bits may be corrupted. Hence, $n \sim B(2^{-4}, 32)$. For the fault position s_1, the possibility of cancelling out the s-box output difference has been ruled out the same way as in the case of s_0. Using the binomial distribution, it follows that $P[16 \leq n \leq 31]$ is negligible.

- $\{s_8\}$: if $R1_0^0 = 0$, then, $n = 0$ with probability 1. Otherwise, all four s-box input bits may be corrupted and $n \sim B(2^{-4}, 32)$ and as for the previous case, $P[16 \leq n \leq 31]$ is negligible.

- $\{s_2, s_3\}$: only the least significant bit will certainly not be corrupted. For s_3, the cancellation of the s-box output difference is ruled out as in the case of s_0. In case of s_2, there exists one s-box input such that the s-box output difference can be cancelled out by inverting the second most significant bit ($S(1111) = S(1111 \oplus 1110) \oplus 0100$). Approximating $n \sim B(2^{-3}, 32)$ gives $P[16 \leq n \leq 31] \approx 2^{-21}$.

- $\{s_4\}$: the most significant two bits may be corrupted, from which it follows that $n \sim B(2^{-2}, 32)$. So, $P[16 \leq n \leq 31] \approx 0.002$.

- $\{s_6, s_7\}$: no s-box input bits can be corrupted and thus $n = 32$ with probability 1

- $\{s_5\}$: Only the most significant bit of every s-box input may be corrupted. Thus $n \sim B(\frac{1}{2}, 32)$ and $P[16 \leq n \leq 31] = \sum_{i=16}^{31} \binom{32}{i} \frac{1}{2^i} \frac{1}{2^{(32-i)}} = 0.569$.

From the above reasoning, it follows that when the fault does not occur at s_5, $P[16 \leq n \leq 31] \approx \frac{1}{11} \times 0.02 \approx 0.0018$, where $\frac{1}{11}$ is the probability that the fault occurred at s_4, given that it did not occur at s_5. On the other hand, if the fault occurred at s_5, the probability of event $16 \leq n \leq 31$ is equal to 0.569. This analysis indicates that one can decide whether the fault occurred at s_5 or not by verifying whether $16 \leq n \leq 31$, or not, respectively.

In Algorithm 1, keystream words for which $16 \leq n \leq 31$ are considered. Namely, once such a keystream word have been found, the values of *activated* s-boxes are used to learn about the corresponding δ_i values. According to the discussion above, if the fault indeed occurred at s_5, such differences necessarily represent the s-box output difference for the input difference equal to $0x8$. To diminish the possibility of false positives (event $16 \leq n \leq 31$ takes place, but the fault does not occur at s_5), the final output difference value is taken as the most frequent difference candidate taken over different faulty keystream words at the (fixed) SOSEMANUK step in question, for which $16 \leq n \leq 31$ holds.

Algorithm 1

- Initialize 32 multisets: $Cand_1(k) = \emptyset$, $k = 0, \ldots, 31$.

- For each faulty keystream word $z'_t|z'_{t+1}|z'_{t+2}|z'_{t+3}$, such that

$$16 \leq \#\{z'^i_t|z'^i_{t+1}|z'^i_{t+2}|z'^i_{t+3} = z^i_t|z^i_{t+1}|z^i_{t+2}|z^i_{t+3} : i = 0, \ldots 31\} \leq 31 \qquad (6)$$

do:

- For each $0 \leq k \leq 31$, if $d = z_t'^k|z_{t+1}'^k|z_{t+2}'^k|z_{t+3}'^k \oplus z_t^k|z_{t+1}^k|z_{t+2}^k|z_{t+3}^k$ is different than 0, add d to $Cand_1(k)$.

- Return the most frequent element in the multiset $Cand_1(i)$ as $\delta_i = S(X_i \oplus 0\text{x}8) \oplus S(X_i)$, for each $0 \leq i \leq 31$.

The overall number of required fault injections $m = 1536$ has been determined by incrementing m in steps of 128 and experimentally verifying that Algorithm 1 always recovers the correct $\delta_i = S(X_i \oplus 0x8) \oplus S(X_i)$, $i = 0, \ldots 31$ for 1000 randomly initialized instants of SOSEMANUK.

Algorithm 2 uses δ_i recovered by Algorithm 1 to find the sets Δ_i, $i = 0, \ldots 31$. In particular, the algorithm recognizes faulty keystream words that correspond to an error in register s_4 and then uses the s-box output differences in such keystream words to deduce Δ_i for $i = 0, \ldots 31$.

The criterion for recognizing faults in register s_4 is similar to the previously stated criterion for recognizing faults in s_5. However, instead of asking for 16 or more unactivated s-boxes, we expect to have more than 16 s-boxes which are either unactivated or with output difference equals to δ_i. Namely, let v be the number of s-boxes in one step of SOSEMANUK which are either not activated, or activated by an input difference of $0x8$. The probability of the event that one s-box is either not activated, or activated by an input difference of $0x8$ depends on the location where the fault occurred. In case the error is in register s_4, the probability in question will be $\frac{1}{2}$ since in that case only the 2 most significant bits of the

s-box input may be faulted and the input difference has to among $0x0$, $0x8$, $0xc$ and $0x4$ values. Thus, if the fault is in s_4, $v \sim B(\frac{1}{2}, 32)$, and $P[16 \leq v \leq 31] = 0.569$. On the other hand, if the fault occurs at some other register, say at $R1$, all four s-box input bits may be corrupted and the probability that the input difference will be either $0x8$ or $0x0$ is significantly smaller. Again, this gives a methodology to decided whether the fault occurred at s_4 or not by counting the number of s-boxes which reacted with difference of either δ_i (using the corresponding i) or 0. Once the faults due to an error in register s_4 are recognized, finding the sets Δ_i proceeds with the following logic. When a keystream word for which the event $16 \leq v \leq 31$ took place has been found, the output s-box differences which are not due to input difference of $0x8$ or $0x0$ have to be due to difference $0xc$ or $0x4$. Again, to diminish the possibility of false positives (i.e., $16 \leq v \leq 31$ but the fault does not occur at s_4), the final output set is taken as the set with two most frequent difference candidates for the difference taken over different faulty keystream words at the SOSEMANUK step in question for which $16 \leq v \leq 31$ holds.

Algorithm 2

- Initialize 32 multisets: $Cand_{2,3}(k) = \emptyset$, $k = 0, \ldots, 31$.

- For each faulty keystream output word $z'_t|z'_{t+1}|z'_{t+2}|z'_{t+3}$, such that

$$\begin{aligned} 16 \leq \#\{z'^i_t|z'^i_{t+1}|z'^i_{t+2}|z'^i_{t+3} = z^i_t|z^i_{t+1}|z^i_{t+2}|z^i_{t+3}|i = 0, \ldots 31\}+ \\ \#\{z'^i_t|z'^i_{t+1}|z'^i_{t+2}|z'^i_{t+3} \oplus z^i_t|z^i_{t+1}|z^i_{t+2}|z^i_{t+3} = \delta_i|i = 0, \ldots 31\} \leq 31 \end{aligned} \tag{7}$$

where δ_i, $0 \leq i \leq 31$ has been recovered by Algorithm 1, do:

- For each $0 \leq k \leq 31$, add each $d = z_t'^k|z_{t+1}'^k|z_{t+2}'^k|z_{t+3}'^k \oplus z_t^k|z_{t+1}^k|z_{t+2}^k|z_{t+3}^k$ such that $d \notin \{0, \delta_k\}$ to the multiset $Cand_{2,3}(k)$.

- Return the two highest occurring elements in the multiset $Cand_{2,3}(i)$ as the required two-element set Δ_i, for each i.

For the above choice of total number of faults $m = 1536$, Algorithm 2 always succeeded in recovering the sets Δ_i, $i = 0, \ldots 31$, for 1000 randomly initialized instants of SOSEMANUK.

3.4.2 Restricting the number of candidates for the LFSR registers

In each SOSEMANUK step, in which a 128-bit keystream word is produced, according to (5), 32 4×4 s-boxes are applied. In the previous subsection, it has been shown how to use the faulty information to deduce the s-box output differences for certain input s-box differences. Naturally, these evaluated input-output differences impose a constraint on the actual input-output values. In this subsection, the sets of possible s-box input-output values are deduced and the effect of the deduced input-output s-box values constraints on the number of candidates for the LFSR registers (s_0, s_1, s_2, s_3) is presented.

Having determined the δ_i value and the two-element set Δ_i by Algorithms 1 and 2, for each $0 \leq i \leq 31$, the actual input-output values for the s-box are deduced according to Table 1. As can be noted from the table, in case the s-box input is even, the input-output value can be deduced uniquely. On the other hand, in case when the s-box input value is

δ_i,Δ_i	i-th s-box input	i-th s-box output
5,{8,B}	0	8
9, {2,D}	2	7
3,{B,E}	4	3
F,{4,D}	6	A
5,{D,E}	8	D
9,{4,B}	A	E
3,{8,D}	C	0
F,{2,B}	E	5
7,{A,D}	{1,5,9,D}	{6,C,1,B}
D,{6,B}	{3,7,B,F}	{9,F,4,2}

Table 1: Determining the s-box input-output values based on sets δ_i and Δ_i

odd, there exist four candidates for the s-box input-output.

Assuming a uniform distribution on the s-box input values, it is expected that the attacker will deduce 64 out of 128 output bits. For the remaining 64 bits, it will be composed out of 16 4-bit values, each restricted to 4 candidates. The overall number of candidates for the 128-bit value $Serpent1(f_0|f_1|f_2|f_3)$ is then $4^{16} = 2^{32}$. Since we have

$$z_0|z_1|z_2|z_3 = Serpent1(f_0|f_1|f_2|f_3) \oplus s_0|s_1|s_2|s_3 \tag{8}$$

and $z_0|z_1|z_2|z_3$ is known, it follows that there will be 2^{32} candidates for $s_0|s_1|s_2|s_3$.

The number of candidates for $s_4|s_5|s_6|s_7$, $s_8|s_9|s_{10}|s_{11}$ and $s_{16}|s_{17}|s_{18}|s_{19}$ can be restricted in a similar way. Namely, for that purpose, Algorithms 1 and 2 need to be applied using $z_4|z_5|z_6|z_7$, $z_8|z_9|z_{10}|z_{11}$ and $z_{16}|z_{17}|z_{18}|z_{19}$ and the faulty values obtained by the first phase of the attack described in section 3.3 for $l = 1$, $l = 2$ and $l = 4$, respectively. Then, Table 1 is utilized to restrict the s-box input-output values occurring in steps $t = 1$, $t = 2$ and $t = 4$. Following the procedure explained in this section, it follows that $s_4|s_5|s_6|s_7$,

$s_8|s_9|s_{10}|s_{11}$ and $s_{16}|s_{17}|s_{18}|s_{19}$ are expected to be restricted to 2^{32} candidates each.

3.4.3 Further pruning of the LFSR registers candidates

In the previous subsection, the uncertainty for (s_0, s_1, s_2, s_3), $(s_8, s_9, s_{10}, s_{11})$ and (s_{16},s_{17}, s_{18},s_{19}) values has been reduced. In this subsection, we note that these three four-tuples of 32-bit values are not independent. Namely, according to (1), we have $s_{10} = s_9 \oplus \alpha^{-1} s_3 \oplus \alpha s_0$ and $s_{18} = s_{17} \oplus \alpha^{-1} s_{11} \oplus \alpha s_8$. These two relations are used to further prune candidates for (s_0, s_1, s_2, s_3) and $(s_8, s_9, s_{10}, s_{11})$. More precisely, after the end of the process, the attacker is left with 2^{16} candidates for

$$(f_0, f_1, f_2, f_3, s_0, s_1, s_2, s_3, f_8, f_9, f_{10}, f_{11}, s_8, s_9, s_{10}, s_{11}) \tag{9}$$

The two relations from the previous paragraph can be rewritten as

$$\alpha^{-1} s_3 \oplus \alpha s_0 = s_{10} \oplus s_9 \tag{10}$$

$$\alpha^{-1} s_{11} \oplus \alpha s_8 = s_{18} \oplus s_{17} \tag{11}$$

Before stating the candidate reduction procedure, we note that the candidates for (s_0, s_1, s_2, s_3) are specified in a way which allows listing them in a table efficiently. In particular, the candidate set for (s_0, s_1, s_2, s_3) is specified by sets B_i, $i = 0, \ldots 31$, such that $s_0^i|s_1^i|s_2^i|s_3^i \in B_i$. Then, each element of the set $B_0 \times B_1 \times \ldots \times B_{31}$ specifies one (s_0, s_1, s_2, s_3) value. The sets of candidates for $(s_8, s_9, s_{10}, s_{11})$ and $(s_{16}, s_{17}, s_{18}, s_{19})$ can be transformed to a list in

the same way and this property is used in step (1) and step (5) of the procedure below.

1. List all of the (s_0, s_1, s_2, s_3) and $(s_{16}, s_{17}, s_{18}, s_{19})$ candidates and call the two generated tables T_1 and T_3, respectively. Include also the columns containing (f_0, f_1, f_2, f_3) and $(f_{16}, f_{17}, f_{18}, f_{19})$ in T_1 and T_3, respectively. Create an empty table T.

2. Extend T_1 by adding a column with the left-hand side of equation (10).

3. Extend T_3 by adding a column with the right-hand side of equation (11).

4. Sort T_1 and T_3 by columns added in steps (2) and (3).

5. For each candidate for $(s_8, s_9, s_{10}, s_{11})$

    5.1. Calculate the left-hand side of equation (11). If there does not exists an element in T_3 such that (11) holds, go to the next $(s_8, s_9, s_{10}, s_{11})$ candidate (step (5)).

    5.2. Otherwise, calculate the right-hand side of equation (10) and find rows of T_1 for which (10) holds. For each such row, add the complete row of the form (9) to table T.

To find the expected size of table T, note that it is expected that 16 bits of the T_3 table column containing $s_{18} \oplus s_{17}$ value are constant, due to the fact that 16 out of 32 s-box inputs corresponding to $(s^i_{16}, s^i_{17}, s^i_{18}, s^i_{19})$ have been recovered uniquely by the procedure in the previous subsection. On the other hand, no constant bits are expected to exist in $\alpha^{-1}s_{11} \oplus \alpha s_8$ values due to randomization resulting from multiplying by α and α^{-1}. Thus, about 2^{16} candidates for $(s_8, s_9, s_{10}, s_{11})$, with the corresponding $(f_8, f_9, f_{10}, f_{11})$, will pass the elimination step (5.1).

In step (5.2), the remaining 2^{16} candidates are joined with T_1, which contains 2^{32} rows, according to (10). Since there exists no fixed bits in the $\alpha^{-1}s_3 \oplus \alpha s_0$ column of T_1, it is expected that around 2^{16} will be present in the output of the join step, i.e., in table T. Since both T_1 and T_3 contain 9 32-bit words in each row and table T contains 16 32-bit words in each row, the required memory space for the previous procedure is $2 \times 2^{32} \times 9 \times 4 + 2^{16} \times 16 \times 4 = 2^{38.17}$ bytes. The computational cost is equal to sorting two tables of 2^{32} rows, executing a search in a sorted table of length 2^{32} for 2^{32} times and finally executing a search for 2^{16} times in the sorted table of 2^{32} entries. By noting that sorting tables of length n takes $O(nlog(n))$ steps and that a binary search in the sorted table requires $O(log(n))$ steps, the overall cost is about $2^{32} \times 32 \times 2 + 2^{32} \times 32 + 2^{16} \times 32 = 2^{38.585}$ operations.

3.5 Recovering the rest of the inner state

In the previous subsections, we have reduced the LFSR complexity to 2^{32} candidates for (s_4, s_5, s_6, s_7) and 2^{16} candidates for the registers present in (9). In this subsection, a guess-and-determine like procedure that completes the secret inner state recovery is provided.

Let $R1_t^0$ denote the least significant bit of register $R1_t$. To recover s_4, s_5, $R1_0$ and $R2_0$, the following steps are applied:

- Pick a row from table T as a guess for (9).
- Determine s_4 from $s_4 = \alpha(\alpha s_1) \oplus \alpha(s_{10} \oplus s_{11})$ which holds due to (1) since s_1, s_{10} and s_{11} are known.
- Guess $R1_0$ by fixing the register to one of the 2^{32} possible values.

- Determine:

  - $R2_0$, from $f_0 = (R1_0 \boxplus s_9) \oplus R2_0$

  - $R2_1$, from $R2_1 = Trans(R1_0)$

  - $R1_1$, from $R1_1 = R2_0 \boxplus (s_2 \oplus R1_0^0 \cdot s_9)$, which is another way to formulate (2)

  - $R2_2$, from $R2_2 = Trans(R1_1)$

  - $R1_2$, from $R1_2 = R2_1 \boxplus (s_3 \oplus R1_1^0 \cdot s_{10})$, which follows from (2)

  - $R2_3$, from $R2_3 = Trans(R1_2)$

  - $R1_3$, from $R1_3 = R2_2 \boxplus (s_4 \oplus R1_2^0 \cdot s_{11})$, which follows from (2)

  - s_{12}, from $f_3 = (R1_3 \boxplus s_{12}) \oplus R2_3$

  - s_5, from $s_{12} = s_{11} \oplus \alpha^{-1} s_5 \oplus \alpha s_2$

With a guess for (9) from the first step of the procedure above and having recovered s_4, s_5, $R1_0$ and $R2_0$, the only left unknown inner state registers are s_6 and s_7. To recover the remaining two registers, the table of 2^{32} candidates for (s_4, s_5, s_6, s_7) obtained in section 3.4.2 is matched with newly found value for s_4, s_5, as follows. Consider the s-box input-output in the second iteration of SOSEMANUK, for which the input-output has not been recovered uniquely. For some $0 \leq i \leq 31$, $f_7^i|f_6^i|f_5^i|f_4^i$ and consequently, $S(f_7^i|f_6^i|f_5^i|f_4^i)$ can take 4 values as specified by Table 1. More precisely, rewriting (5) while isolating i-th s-box

$$z_7^i|z_6^i|z_5^i|z_4^i = S(f_7^i|f_6^i|f_5^i|f_4^i) \oplus s_7^i|s_6^i|s_5^i|s_4^i, \tag{12}$$

we have two options regarding the possible candidates. In other words, from the last two

rows of Table 1, we have either

$$S(f_7^i|f_6^i|f_5^i|f_4^i) \in \{0110, 1100, 0001, 1011\} \tag{13}$$

or

$$S(f_7^i|f_6^i|f_5^i|f_4^i) \in \{1001, 1111, 0100, 0010\}. \tag{14}$$

Moreover, according to the procedure given in this subsection, the value of bits s_4^i, s_5^i has been determined uniquely. Since s_4^i and s_5^i are known, according to (12), the two least significant bits of $S(f_7^i|f_6^i|f_5^i|f_4^i)$ can be determined uniquely. Finally, due to the structure of sets (13) or (14), given information on the two least significant bits, all the 4 bits of $S(f_7^i|f_6^i|f_5^i|f_4^i)$ are uniquely determined. Presented reasoning uniquely determines the input-output for every s-box, from which, according to (12), s_7 and s_6 are determined uniquely, which completes the recovery of the whole secret inner state.

Now, the found secret inner state can be verified by comparing the actual SOSEMANUK output with the output produced by the recovered inner state. If a difference registered, the next guess for (9) and $R1_0$ is made and the procedure is repeated.

3.6 Summary and conclusions

In this chapter, a differential fault analysis attack on SOSEMANUK was presented. The overall attack complexity can be summarized as follows:

- The average number of faults required to perform the attack is $4 \times 1536 = 6144$.

These 1536 transient faults are introduced in steps $t = 0$, $t = 1$, $t = 2$ and $t = 4$. This fault injection phase requires the attacker to reinitialize the cipher for 6144 times

- The number of operations required for the attack is dominated by the guess-and-determine part of the analysis. Namely, as concluded in section 3.4.3, table T has 2^{16} rows and thus there exists 2^{16} possible guesses for (9). Since register $R1_0$ is a 32-bit value, the number of guesses that need to be checked is $2^{16} \times 2^{32} = 2^{48}$. Verifying each guess according to the procedure in section 3.5 is equivalent to one SOSEMANUK iteration and thus the attack requires work equivalent to around 2^{48} iterations.

- The storage amount required for the attack is equal to the size of the tables T_1, T_3 and T which amounts to $2^{38.17}$ bytes.

Chapter 4

Differential Fault Analysis of Hummingbird

4.1 Introduction

Hummingbird [13, 17] is an encryption algorithm designed for lightweight software and lightweight hardware implementations on resource-constrained devices such as RFID tags and wireless sensor nodes. Its design was inspired by the Enigma machine which led to a hybrid combination of block cipher and stream cipher structures.

The security of Hummingbird was evaluated by its designers who concluded that the the cipher is resistant to most common attacks against block ciphers and stream ciphers including birthday attacks, differential and linear cryptanalysis, structure attacks, algebraic attacks, and cube attacks. Also a chosen-IV and chosen-message attack was reported by Saarinen [85].

In this chapter, we present a differential fault analysis attack on Hummingbird. The fault model in which we analyze the cipher is the one in which the attacker is assumed to inject a transient fault at a random 4 bit word before the linear transformation, after the 4×4 s-boxes, of the four block ciphers which are used in the Hummingbird encryption process but cannot control the exact location of injected faults. We also assume that the attacker is able to reset the cipher an arbitrary number of times.

The main idea of our attack is inspired by recent differential fault analysis attacks against the AES. In these attacks, the attacker collects few differential pairs relative to the last non-linear step which allows the attacker to reduce and finally guess the values computed in the last rounds and infer the last round key. However, unlike the AES case in which once the round key has been recovered, the key schedule can be inverted to obtain the initial secret key, this is not possible for Hummingbird since it does not have an explicit invertible key schedule. Instead, the above procedure has to be reiterated on each round, starting from the last one, until the whole key material is exposed. Our simulation results showed that our attack, which recovers the 256-bit key, requires around 50 fault injections and 2^{66} steps.

The rest of the chapter is organized as follows. A brief description of the relevant details of Hummingbird is provided in the next section. Our attack and its complexity analysis are provided in section 4.3.

4.2 Description of Hummingbird

The following notation and functions are used throughout the chapter:

$\boxplus$: addition mod 2^{16}.

$\oplus$: bit-wise XOR.

$||$: Concatenation of the words.

$<<<$: left rotation defined on 16-bit value.

S_i: the i^{th} 4-bit s-box where $i = 1, 2, 3, 4$.

$SBOX$: the 16-bit nonlinear function which equals to $S_1||S_2||S_3||S_4$.

$SBOX^{-1}$: the inverse mapping of $SBOX$.

$L(x)$: the 16-bit linear transform equation.

$L^{-1}(x)$: the inverse mapping of $L(x)$.

$\triangle X_i$: the 4-bit differential input of S_i.

$\triangle Y_i$: the 4-bit differential input of S_i.

As mentioned above, Hummingbird has a hybrid structure and consists of two main components: a stream cipher and a block cipher. The input/output block size is 16-bit and the internal state and key size are 80-bit and 256-bit, respectively.

Figure 4 depicts a top level view of the encryption process. The cipher algorithm consists of four 16-bit block ciphers, $E_{k_i}(i = 1, 2, 3, 4)$, four 16-bit internal state registers $RSi(i = 1, 2, 3, 4)$, and a 16-bit linear feedback shift register (LFSR). The original 256-bit key is divided into four 64-bit subkeys $k_i, i = 1, 2, 3, 4$, which are used in the four block

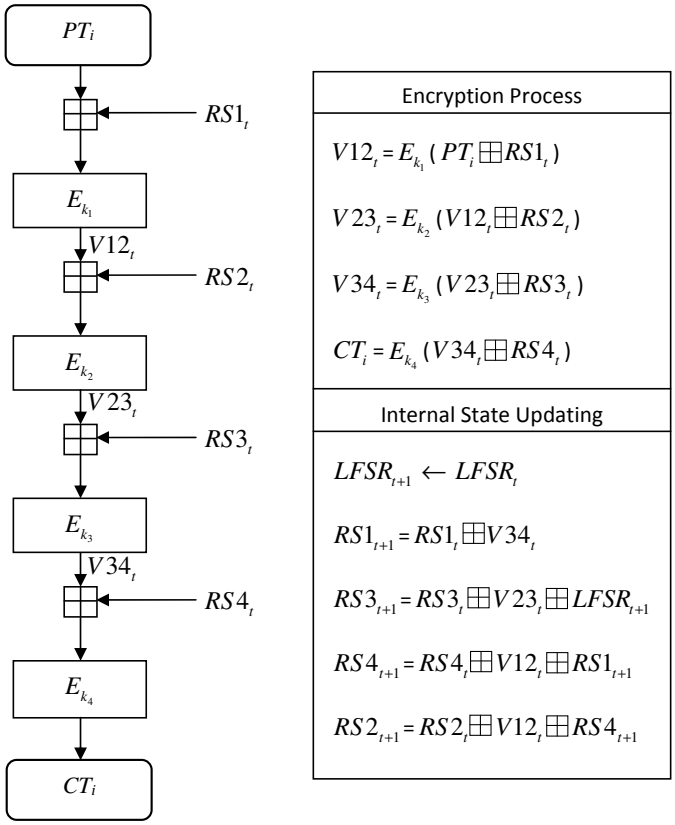


Figure 4: An overview of the Hummingbird encryption process.

ciphers.

A 16-bit plaintext block PT_i is added to the first internal state register $RS1$ modulo 2^{16}. The result of the addition is then encrypted by the first block cipher E_{k_1}. This procedure is repeated in a similar manner for another three times and the output of E_{k_4} is the corresponding ciphertext, CT_i. At the same time, the states of the four internal state registers will also be updated based on their current state values, the outputs of the first three block ciphers, and the value of the LFSR. The decryption process follows a similar pattern as in the encryption mode.

In practice, Hummingbird is initialized with four 16-bit random nonces, $NONCE_i, i = 0, 1, 2, 3$, to construct the four internal state registers $RS_i, i = 1, 2, 3, 4$, respectively, followed by applying the encryption algorithm to the message $RS1 \boxplus RS3$. The final 16-bit ciphertext of the initialization is used to initialize the LFSR where the 13^{th} bit of the LFSR is always set to 1.

The 16-bit block cipher is a typical substitution-permutation (SP) network with 16-bit block size and 64-bit key. It consists of five rounds: four regular rounds and a final round that only includes the key mixing and the s-box substitution steps. Each round comprises of a key mixing step, a substitution layer, and a permutation layer. For the key mixing, a simple xor operation is used. Figure 5 depicts the structure of the 16-bit block cipher and the specification of the four s-boxes and linear transform mapping which are used in E_{k_i}.

The 64-bit subkey k_i is divided into four 16-bit round keys $K_j^{(i)}, j = 1, 2, 3, 4$, which are used in the four regular rounds of SP structure, respectively. Moreover, two keys $K_5^{(i)}$ and $K_6^{(i)}$ which are directly derived from the four round keys are applied before and after the last s-boxes.

Further details about the encryption, decryption and initialization processes can be found in [13].

4.3 The proposed attack

The main idea of using differential fault analysis against Hummingbird is to recover the whole secret keys based on determining round keys in four steps. In particular, we can

retrieve k_4 and peel off the last round function of the encryption process (E_{k_4}) to determine $V34_t \boxplus RS4_t$. Detailed explanation of how to recover the keys of the E_{k_i} step is provided in the next subsection. By guessing all the 2^{16} values of $RS4_t$ and applying the same method for determining the keys of the 16-bit block cipher (E_{k_3}), k_3 can be revealed. Similarly, the values of k_2 and k_1 can be determined by applying the same procedure and guessing $RS3_t$ and $RS2_t$. Finally, by guessing $RS2_t, RS3_t$ and $RS4_t$ we can determine 2^{48} candidates for k_1, k_2, k_3 and k_4. The correct key can then be uniquely determined using additional plaintext/ciphertext pairs.

4.3.1 Key recovery of E_{k_i}

In this section, we describe a differential fault analysis on the E_{k_i} function which are used in Hummingbird algorithm. The injected faults are assumed to occur in one 4-bit word before the linear transform and after the 4×4 s-boxes.

Consider one 4-bit word differences at the input of the linear layer $L(x)$. We have 60 ($= 15 \times 4$) possible such differences corresponding to 4 different possible locations and 15 different possible values. Because of the linearity, the number of corresponding possible differences at its output is also 60 but, while the input difference affected one 4-bit word only, the output difference affects up to 12 bits because of the diffusion linear transformation layer (See Table 4.3.1.) Note that the key addition does not change the set of possible differences.

First we describe how we can recover the last round subkey $K_6^{(i)}$. We follow the same idea of the attack described in [69]. Algorithm 1 shows the details.

	Differential input = ($\triangle X_1 \triangle X_2 \triangle X_3 \triangle X_4$)			
	$\triangle X_1 \neq 0$	$\triangle X_2 \neq 0$	$\triangle X_3 \neq 0$	$\triangle X_4 \neq 0$
The corresponding differential output	1440	0144	4014	4401
	2880	0288	8028	8802
	3CC0	03CC	C03C	CC03
	4011	1401	1140	0114
	5451	1545	5154	4515
	6891	1689	9168	8916
	7CD1	17CD	D17C	CD17
	8022	2802	2280	0228
	9462	2946	6294	4629
	A8A2	2A8A	A2A8	8A2A
	BCE2	2BCE	E2BC	CE2B
	C033	3C03	33C0	033C
	D473	3D47	73D4	473D
	E8B3	3E8B	B3E8	8B3E
	FCF3	3FCF	F3FC	CF3F

Table 2: Possible differential outputs of the linear transformation - The $\mathcal{D}$'s list
(Differential output = ($\triangle Y_1 \triangle Y_2 \triangle Y_3 \triangle Y_4$)
All values are in Hexadecimal)

Algorithm 1:

1. Compute the 60 possible differences at the output of linear transform ($L(x)$), i.e. the 60 values of $L(x)$, where $x = x_1||x_2||x_3||x_4$ and only one of the x_i's has a Hamming weight not equal to zero. Store the obtained values in a list $\mathcal{D}$.

2. Consider a plaintext P, its corresponding ciphertext C and faulty ciphertext C'.

3. Guess the value of the round key $K_6^{(i)}$.

4. Compute the difference $SBOX^{-1}(C \oplus K_6^{(i)}) \oplus SBOX^{-1}(C' \oplus K_6^{(i)})$. Check whether it is in $\mathcal{D}$. If yes, add the round key to the list $\mathcal{L}$ of possible candidates.

Fault location		The recovered data	The recovered keys
E_{k_4}	round 4 round 3 round 2 round 1	$K_2^{(4)} \oplus K_4^{(4)}$ $K_1^{(4)} \oplus K_3^{(4)}$ $K_4^{(4)}$ $K_3^{(4)}$	 $K_2^{(4)}$,$K_4^{(4)}$ $K_1^{(4)}$,$K_3^{(4)}$
Guess all of 2^{16} possibilities of $RS4_t$		$\longmapsto$	Get $V34_t$
E_{k_3}	round 4 round 3 round 2 round 1	$K_2^{(3)} \oplus K_4^{(3)}$ $K_1^{(3)} \oplus K_3^{(3)}$ $K_4^{(3)}$ $K_3^{(3)}$	 $K_2^{(3)}$,$K_4^{(3)}$ $K_1^{(3)}$,$K_3^{(3)}$
Guess all of 2^{16} possibilities of $RS3_t$		$\longmapsto$	Get $V23_t$
E_{k_2}	round 4 round 3 round 2 round 1	$K_2^{(2)} \oplus K_4^{(2)}$ $K_1^{(2)} \oplus K_3^{(2)}$ $K_4^{(2)}$ $K_2^{(4)}$	 $K_2^{(2)}$,$K_4^{(2)}$ $K_1^{(2)}$,$K_3^{(2)}$
Guess all of 2^{16} possibilities of $RS2_t$		$\longmapsto$	Get $V12_t$
E_{k_1}	round 4 round 3 round 2 round 1	$K_2^{(1)} \oplus K_4^{(1)}$ $K_1^{(1)} \oplus K_3^{(1)}$ $K_4^{(1)}$ $K_3^{(1)}$	 $K_2^{(1)}$,$K_4^{(1)}$ $K_1^{(1)}$,$K_3^{(1)}$

Table 3: Summary of the fault attack on Hummingbird

5. Consider a new plaintext P (with corresponding C and C') and go back to step 2. This time, the round key guesses only go through the list $\mathcal{L}$ of possible candidates. If the difference computed at step 4 is not in $\mathcal{D}$, remove the candidate from $\mathcal{L}$). Repeat until there remains only one candidate in $\mathcal{L}$.

The complexity of the Algorithm 1 is around 2^{16} since we have to search all of the target key at the beginning of step 3 of the algorithm. After $K_6^{(i)}$ is uniquely determined, the last round is peeled off, and the attack is repeated on the reduced cipher.

We can find the subkey $K_5^{(i)}$ by repeating the same algorithm which is used to recover $K_6^{(i)}$ but using the difference equation $SBOX^{-1}(L^{-1}(C_{-1} \oplus K_5^{(i)})) \oplus SBOX^{-1}(L^{-1}(C'_{-1} \oplus K_5^{(i)}))$ where $C_{-1} = SBOX^{-1}(C \oplus K_6^{(i)})$ and $C'_{-1} = SBOX^{-1}(C' \oplus K_6^{(i)})$. $K_3^{(i)}$ and $K_4^{(i)}$ are revealed in the same way. Finally, we determine the remaining subkeys of $K_1^{(i)}$ and $K_2^{(i)}$ by computing $K_3^{(i)} \oplus K_5^{(i)}$ and $K_4^{(i)} \oplus K_6^{(i)}$, respectively.

The above attack was simulated for 10,000 times using different random values for the 16-bit input and 64-bit subkey of k_i for E_{k_i}. Based on our simulations, our attack requires an average of 12.51 fault injections to recover the whole subkeys, $k_i, i = 1, 2, 3, 4$. The recorded minimum and the largest number of required faults were 8 and 22, respectively.

Table 4.3.1 summarizes the whole steps of the attack to recover the 256-bit secret key.

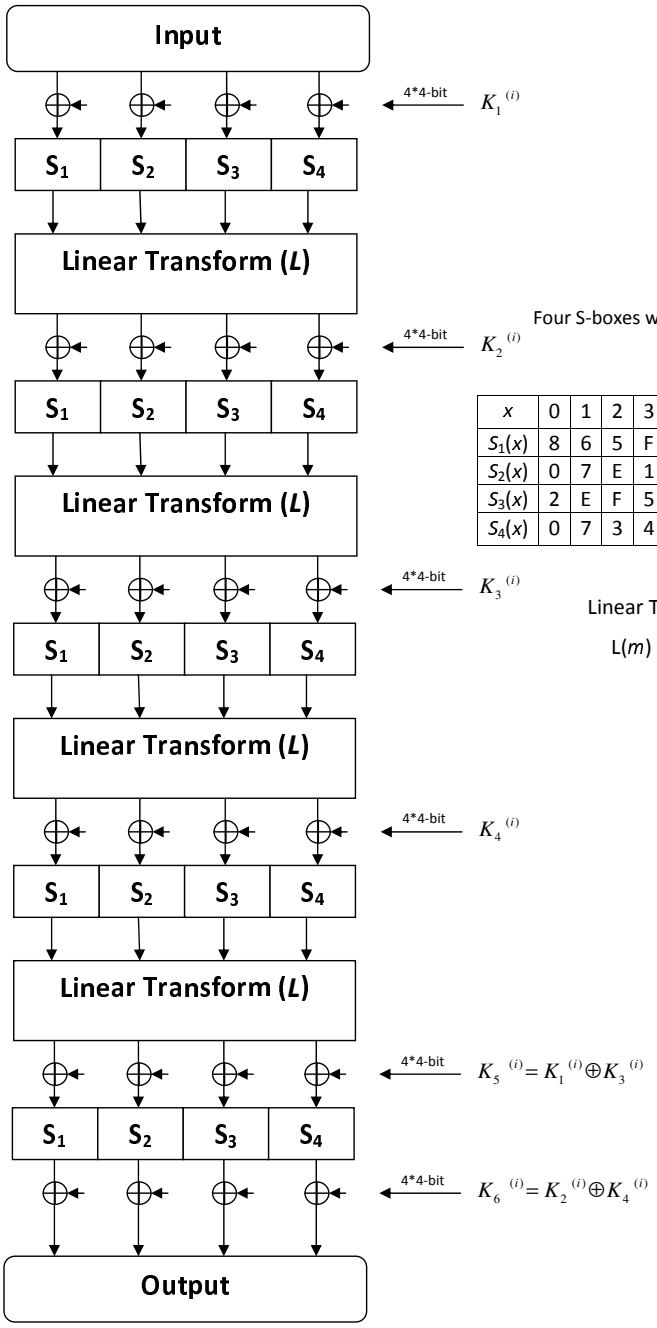
After the keys are recovered, we can then find the other internal registers by applying the initialization process since the values of $NONCE_i (i = 0, 1, 2, 3)$ are public.

4.3.2 The complexity of our attack

To recover all of the 256-bit secret key we have to apply the above fault attack for four times and guess all candidate values of the three internal registers: $RS2_t, RS3_t$ and $RS4_t$. From our experimental results, we need around 12.5 faulty values to uniquely determine each subround key. Thus, our attack requires 50 faulty ciphertext, 2^{48} guessing of 16-bit values and calling Algorithm 1 (with complexity 2^{16}) in each step to reveal the subkeys. Since we have to use four times of the Algorithm 1 in each step of E_{k_i}, the total complexity of our attack is about $4 \times 2^{16} \times 2^{48} = 2^{66}$.

4.4 Summary and conclusions

In this chapter, we presented a fault attack against a newly introduced ultra lightweight encryption algorithm, Hummingbird. Each 64-bit round key can be found, on average, using 12.51 faulty encryptions. If we assume that the 256-bit secret key and 80-bit internal state have random distribution, then the whole cipher can be broken after around 50 faults. To fully recover the key, we have to guess 2^{48} values of three 16-bit internal state registers which brings the whole complexity of our attack to 2^{66}.



Four S-boxes which used in the block cipher in Hummingbird (in hexadecimal notation)

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S_1(x)$	8	6	5	F	1	C	A	9	E	B	2	4	7	0	D	3
$S_2(x)$	0	7	E	1	5	B	8	2	3	A	D	6	F	C	4	9
$S_3(x)$	2	E	F	5	C	1	9	A	B	4	6	8	0	7	3	D
$S_4(x)$	0	7	3	4	C	1	A	F	D	E	6	B	2	8	9	5

Linear Transform (L) : $\{0,1\}^{16} \rightarrow \{0,1\}^{16}$

$L(m) = m \oplus (m \lll 6) \oplus (m \lll 10)$

Figure 5: The structure of E_{k_i} - the 16-bit block cipher of Hummingbird in encryption mode.

Chapter 5

Scan Based Side Channel Attack on ZUC

5.1 Introduction

Recently, a new set of cryptographic algorithms was proposed by the Data Assurance and Communication Security (DACAS) research center of the Chinese Academy of Sciences [14, 15] for inclusion in the 4G Long Term Evolution (LTE) mobile standard. The core of the new LTE cryptographic algorithms consist of an encryption algorithm, called 128-EEA3, and an integrity algorithm, called 128-EIA3.

The ZUC algorithm [86, 87] is the core of the proposed confidentiality and integrity algorithms. ZUC is a word-oriented stream cipher that generates a key stream of 32-bit words based on two stages of execution. It uses a 128-bit key and a 128-bit initial vector (iv) as input. The two execution stages of ZUC are the key initialization stage and

the working stage. In the first stage, a key initialization is performed, i.e., the cipher is clocked without producing any output. In the second stage, a 32-bit word of output is produced with every round. Preliminary evaluation of the ZUC algorithm, conducted by the algorithm standardization group ETSI SAGE, concluded that it is a strong and suitable candidate for the LTE. Some other evaluations were also presented in the first international workshop on ZUC algorithm [88]. Generally, all of these security evaluations considered only the potential theoretical weaknesses of the algorithm. On the other hand, hardware implementations [89] of cryptographic algorithm are used in many applications in order to achieve the growing requirements for high throughout. Thus, hardware implementation related attacks have to be considered.

In this chapter, we present a scan-based attack on ZUC. A scan path connects registers in a circuit serially so that a tester can observe the register values inside the circuit. The scan path is widely used in recent circuit implementations due to its ease of implementation and high test coverage [90]. Scan-based attacks exploit the scan chains that are inserted into the devices for the purpose of testing. Scan based attacks have been demonstrated on block ciphers such as DES [73] and AES [91], stream ciphers [74, 75] and public keys such as elliptic curve cryptosystems [90]. Our scan based attack on ZUC allows us to uniquely determine the corresponding locations of the inner state variables (the LFSR registers and the memory cells) in the output scan chain and consequently recover the secret initial state.

The rest of the chapter is organized as follows. In the next section, we provide a brief overview of scan based analysis attacks. In section 5.3, relevant details of ZUC are reviewed. Details of the attack are described in section 5.4. Finally, the conclusion is given

in section 5.5.

5.2 General description of the attack

Scan chain based attacks can be considered as a class of side channel attacks that targets the design for testability feature of modern hardware circuity. This design for testability is a design technique in which scan chains are kept with the objective to test designs by providing a simple way to set and observe every flip flop in the hardware circuit. A special signal, scan enable, is added to the design. When this scan enable signal is set, every flip flip of the tested circuit is connected as a chain of registers. The data to this chain is provided through one input pin and the scan output is provided through another output pin. An input pattern can be scanned into the registers on each clock event. Then after a normal run of the circuit, the scan chain content can be scanned out for testing.

In 2004, Yang *et al.* [73] introduced the notion of scan based attacks against dedicated hardware implementations of cryptographic algorithms and described the details of these attacks against DES. This class of attacks was extended to AES in [91] and [92]. Besides, a scan-based attack against ECC was proposed in [90]. Scan based attacks against stream ciphers were introduced by Agrawal *et al* [74], where a detailed attack against Trivium was described. Liu *et al* in [75] presented scan based attacks against other LFSR-based stream ciphers where this technique was applied on the six such stream ciphers including DECIM, Pomaranch, A5/1, A5/2, w7 and LILI II.

Scan based attacks have two phases [74]: ascertaining the internal structure of the scan

chain and deciphering the cryptosystem by revealing the secret internal data. Throughout our analysis, we assume that:

- The attacker knows the details of the cipher algorithm.
- The attacker has access to high level timing diagram of the hardware implementation.
- The secret key is stored in a secure memory and is not part of the scan chain.
- The attacker does not know the structure of the scan chain.
- The device under attack can be run for any prespecified number of clock cycles chosen by the attacker.
- The attacker can scan out the states of internal registers of the device after any prespecified number of clock cycles.
- The attacker can scan in chosen vectors and apply chosen inputs to the device under attack.

For our analysis, we assume a straightforward hardware implementation of ZUC that does not imply any pipelining optimization. We consider an implementation in which both stages of ZUC, the initialization stage and the working stage, are performed in hardware. Furthermore, we assume that the key loading procedure (see section 5.3) is performed off line and then its result is loaded into the LFSR. In this scenario, the hardware associated with the internal state of ZUC would consists of 496 bits for the LFSR and 64 bits for R_1 and R_2. In other words, the scan chain for ZUC will have a total of 560 bits.

5.3 The ZUC specifications

ZUC is a word-oriented stream cipher which takes a 128-bit initial key and a 128-bit initial vector (*iv*) as input and generates a 32-bit word as an output in every clock cycle. This key stream can be used to encrypt the plaintext. ZUC utilizes three main components to generate the keystream output: a Linear Feedback Shift Register (LFSR), bit-reorganization (BR) and non-linear function (F).

The following notation, mostly from [14, 15], will be used throughout the rest of this chapter:

- $\boxplus$: The addition modulo 2^{32}.
- $\oplus$: The bit-wise XOR.
- $\vee$: The bit-wise OR.
- $\wedge$: The bit-wise AND.
- $\overline{X}$: The bit-wise complement of an n-bit word X.
- $||$: Concatenation.
- mod : The modulo operation of integers.
- $(a_1, a_2, \ldots, a_n) \longrightarrow (b_1, b_2, \ldots, b_n)$: The assignment of the values of a_i to b_i in parallel.
- a_H : The leftmost 16 bits of integer a.

- a_L : The rightmost 16 bits of integer a.

- $a <<<_n k$: The k-bit cyclic shift of the n-bit register a to the left.

- $a >> 1$: The l-bit right shift of integer a.

- X_i^j: The j-th bit of an n-bit word X_i.

- $X_i^j(t)$: The j-th bit of an n-bit word X_i after t clock cycles.

- $\overset{?}{X_i^j}$: The j-th bit of an n-bit word X_i has been set to 1 but its corresponding location in the scan chain output has not been determined uniquely.

- $\overset{\checkmark}{X_i^j}$: The location, in the scan chain output, of the j-th bit of an n-bit word X_i has been determined uniquely.

Figure 6 shows the three dependent layers of ZUC. The secret inner state of ZUC consists of the LFSR and the F function layers which have sixteen 31-bit words $(s_0, \ldots, s_{15})$ and 2 32-bit words (R_1, R_2), respectively. The LFSR has two modes of operations: the initialization mode and the working mode.

According to the new version of the ZUC specifications [15], the first mode works as follows:

LFSR With Initialization Mode (u) {

1. $v = 2^{15}s_{15} + 2^{17}s_{13} + 2^{21}s_{10} + 2^{20}s_4 + (1 + 2^8)s_0 \bmod (2^{31} - 1)$;

2. $s_{16} = (v + u) \bmod (2^{31} - 1)$;

3. If $s_{16} = 0$, then set $s_{16} = 2^{31} - 1$;

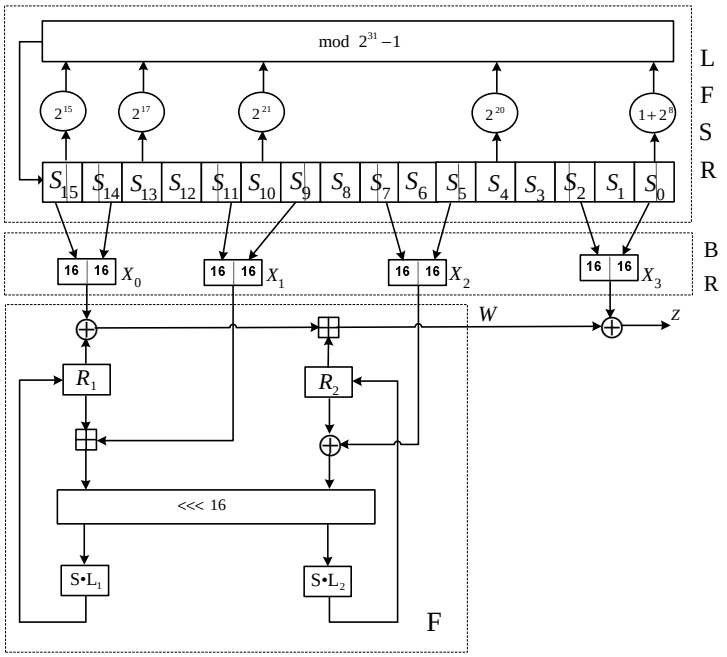


Figure 6: Overview of the ZUC stream cipher

4. $(s_1, s_2, \ldots, s_{15}, s_{16}) \longrightarrow (s_0, s_1, \ldots, s_{14}, s_{15}).\}$

In the working mode, the LFSR does not receive any input, and it updates its current state as follows:

LFSR With Work Mode() {

1. $s_{16} = 2^{15}s_{15} + 2^{17}s_{13} + 2^{21}s_{10} + 2^{20}s_4 + (1 + 2^8)s_0 \bmod (2^{31} - 1)$;

2. If $s_{16} = 0$, then set $s_{16} = 2^{31} - 1$;

3. $(s_1, s_2, \ldots, s_{15}, s_{16}) \longrightarrow (s_0, s_1, \ldots, s_{14}, s_{15}).\}$

The bit-reorganization layer extracts 128 bits from 8 registers of the LFSR and forms 4 32-bit words as follows:

Bit reorganization() {

1. $X_0 = s_{15H}||s_{14L}$;

2. $X_1 = s_{11L}||s_{9H}$;

3. $X_2 = s_{7L}||s_{5H}$;

4. $X_3 = s_{2L}||s_{0H}.\}$

The nonlinear function F consists of two 32-bit words R_1 and R_2 as memory cells. F takes X_0, X_1 and X_2 as it's inputs, which are the outputs of the bit-reorganization layer, then outputs a 32-bit word W as follows:

F(X_0, X_1, X_2) {

1. $W = (X_0 \oplus R_1) \boxplus R_2$;

2. $W_1 = R_1 \boxplus X_1$;

3. $W_2 = R_2 \oplus X_2$;

4. $R_1 = S(L_1(W_{1L}||W_{2H}))$;

5. $R_2 = S(L_2(W_{2L}||W_{1H})).\}$

where S is a 32×32 s-box, L_1 and L_2 are linear transformation. The s-box is composed by 4 juxtaposed 8×8 s-boxes, i.e., $S = (S_0, S_1, S_0, S_1)$. The lookup table definition of these s-boxes can be found in [15]. L_1 and L_2 are linear mapping from 32-bit words to 32-bit words, and are defined as follows:

$$L_1(X) = X \oplus (X <<<_{32} 2) \oplus (X <<<_{32} 10) \oplus (X <<<_{32} 18) \oplus (X <<<_{32} 24)$$

$$L_2(X) = X \oplus (X <<<_{32} 8) \oplus (X <<<_{32} 14) \oplus (X <<<_{32} 22) \oplus (X <<<_{32} 30)$$

The execution of ZUC has two stages: key initialization stage and working stage. The algorithm first calls the key loading procedure to load the key and the iv into the LFSR as the initial state and set R_1 and R_2 to all 0's.

The key loading procedure works as follows:

Key loading {

1. Let D be a 240 bit constant value by

   $D = d_0||d_1||\ldots||d_{15}$

2. For $i = 0 \ldots 15$, let $s_i = k_i||d_i||iv_i$.}

where d_i are 16 15-bit constant values, k_i and $iv_i, i = 0 \ldots 15$ are all bytes of the 128-bit initial key k and the 128-bit initial vector iv respectively. Then the cipher runs the following operation 32 times to finish the key initialization stage.

The initialization stage {

1. Bit reorganization();

2. $w = F(X_0, X_1, X_2)$;

3. LFSR With Initialization Mode($w >> 1$).}

After the initialization stage and the first iteration of the working stage, in which the output W of F is discarded, the algorithm goes into the stage of generation key stream. For each iteration, the following operations are done once and a 32-bit word Z is produced as an output:

The working stage {

1. Bit reorganization();

2. $Z = F(X_0, X_1, X_2) \oplus X_3$;

3. LFSR With Work Mode(). }

5.4 The proposed attack

5.4.1 Overview

Our proposed attack is applicable to scenarios where the attacker has access to the encryption hardware device after the key loading process and the initialization stage have been executed. The proposed scan based attack allows the attacker to recover the internal state of the cipher, but not the key, since we assume that the key bits are stored in a secure memory and cannot be scanned out. According to our scan based analysis model, the attacker is also assumed to be able to re-initialize the encryption device an arbitrary number of times and obtain the values of the states of internal registers of the device after each clock cycle.

The attack can be divided into two phases. The first phase uses the key loading procedure to determine correspondence between the individual bits in the scan chain output and the s_i bits that are loaded with the 128-bit initial key *k* and the 128-bit initial vector *iv* in step 2 of the key initialization stage.

In the second phase (explained in sections 5.4.3 and 5.4.4), the attacker determines the remaining structure of the scan chain, i.e., the attacker determines the exact location of the remaining internal state registers including the remaining bits of the LFSR and the memory cells R_1 and R_2.

5.4.2 Key loading stage

The scan out bits corresponding to the 256 bits of the LFSR register that are loaded with the 128-bit initial key *k* and the 128-bit initial vector *iv* in step 2 of the key initialization stage can be determined by the key loading procedure as follows:

For $l = 0 \ldots 127$:

1 Set *iv*=0 and $k = 2^l$.

2 Load the *k* and the *iv*.

3 Determine the location corresponding to the l^{th} bit in k (which is set to 1) in the $scanned - out$.

4 Use $scanned - in$ with the information of step 2 to determine the correspondence in the $scanned - in$ position.

The above procedure recovers the positions, in the scan out chain, corresponding to the most significant bytes of the LFSR. Recovering the position of the least significant bytes can be achieved by applying the same procedure with *k*=0 and $iv = 2^l$. Thus, after this step, we have

$$\begin{array}{c} \overset{\checkmark}{S_i^{31}}, \overset{\checkmark}{S_i^{30}}, \overset{\checkmark}{S_i^{29}}, \overset{\checkmark}{S_i^{28}}, \overset{\checkmark}{S_i^{27}}, \overset{\checkmark}{S_i^{26}}, \overset{\checkmark}{S_i^{25}}, \overset{\checkmark}{S_i^{24}}, \\ \overset{\checkmark}{S_i^{8}}, \overset{\checkmark}{S_i^{7}}, \overset{\checkmark}{S_i^{6}}, \overset{\checkmark}{S_i^{5}}, \overset{\checkmark}{S_i^{4}}, \overset{\checkmark}{S_i^{3}}, \overset{\checkmark}{S_i^{2}}, \overset{\checkmark}{S_i^{1}} \end{array} \tag{15}$$

for *i*=0...15.

5.4.3 Determining the locations of the remaining LFSR bits

In what follows, we show how to determine the remaining $16 \times 15 = 240$ bit position of the LFSR (which are loaded by the d_i values in the key loading procedure). We also show how to determine four bits of R_1 and R_2 ($R_1^{15}, R_1^{18}, R_1^{22}, R_2^{32}$).

First, we initialize the LFSR register, R_1 and R_2 with all 0's and then run the system for 1 clock cycle to determine some specific positions. Then, we reset the circuit and scan in a specific input pattern with a Hamming weight equals "1". Finally we run the system for a prespecified number of clock cycles to load the above pattern and then perform a scan out operation. As shown below, examining the output pattern in the scan chain output allows us to determine the required bit positions.

1. **Determining the bit locations of S_{15}:**

   (a) Set $S_0(0) = \ldots = S_{15}(0)$=0 and $R_1(0) = R_2(0) = 0$. Then after 1 clock cycle we have:

   - $X_0 = X_1 = X_2 = X_3 = 0$.
   - $R_1(1)$=0x3E553E55 and $R_2(1)$=0x3E553E55.
   - $S_{16} = 2^{31} - 1$.
   - $(S_0, S_1, \ldots, S_{15}) = (0, 0, \ldots$,0,0x7FFFFFFF).
   - (R_1, R_2)=(0x3E553E55,0x3E553E55).

   Thus all the activated bits, i.e., the bits that are set to "1" in the above process, belong to S_{15}, R_1 and R_2.

   (b) For $i = 0 \ldots 7$: set $S_0(0) = 2^i$, $S_1(0) = \ldots = S_{15}(0)$=0 and $R_1(0) = R_2(0) = 0$. Then after 1 clock cycle we have:

   - $X_0 = X_1 = X_2 = X_3 = 0$.
   - $R_1(1)$=0x3E553E55 and $R_2(1)$=0x3E553E55.
   - $S_{16} = 2^i + 2^{i+8}$.
   - $(S_0, S_1, \ldots, S_{15}) = (0, 0, \ldots, 0, 2^i + 2^{i+8})$.
   - (R_1, R_2)=(0x3E553E55,0x3E553E55).

   The values of R_1 and R_2 remain the same as in step 1a. On the other hand, two bits of S_{15} will change which allows us to determine the position of the $(i + 9)^{th}, i = 0 \ldots 7$, bit in S_{15} which corresponds to 2^{i+8} in the equations

above. Consequently, at the end of this step we have

$$\overset{\checkmark}{S_{15}^{9}}\ \overset{\checkmark}{S_{15}^{10}}, \overset{\checkmark}{S_{15}^{11}}, \overset{\checkmark}{S_{15}^{12}}, \overset{\checkmark}{S_{15}^{13}}, \overset{\checkmark}{S_{15}^{14}}, \overset{\checkmark}{S_{15}^{15}}, \overset{\checkmark}{S_{15}^{16}}$$

(c) For $i = 0 \ldots 5$: set $S_{13}(0) = 2^i$, $S_0(0) = \ldots = S_{12}(0) = S_{14}(0) = S_{15}(0)$=0 and $R_1(0) = R_2(0) = 0$. Then after 1 clock cycle we have:

- $X_0 = X_1 = X_2 = X_3 = 0$.
- $R_1(1)$=0x3E553E55 and $R_2(1)$=0x3E553E55.
- $S_{16} = 2^{i+17}$.
- $(S_0, S_1, \ldots, S_{15}) = (0, 0, \ldots, 2^i, 0, 0, 2^{i+17})$.
- (R_1, R_2)=(0x3E553E55,0x3E553E55).

Again, the positions of activated bits in R_1 and R_2 do not change from step 1a. However the $(i+18)^{th}, i = 0 \ldots 5$, bit of S_{15}, which corresponds to 2^{i+17} in the equations above, will change which allows us to determine the exact positions of these bits. Finally, the position of the remaining bit S_{15}^{17}, can be found from the fact that all positions of S_{15} are already determined during step 1a. Thus at the end of this step we have

$$\overset{\checkmark}{S_{15}^{17}}, \overset{\checkmark}{S_{15}^{18}}, \overset{\checkmark}{S_{15}^{19}}, \overset{\checkmark}{S_{15}^{20}}, \overset{\checkmark}{S_{15}^{21}}, \overset{\checkmark}{S_{15}^{22}}, \overset{\checkmark}{S_{15}^{23}}$$

2. **Determining the bit locations of S_{14}:**

(a) Set $S_0(0) = \ldots = S_{15}(0)$=0 and $R_1(0) = R_2(0) = 0$. Then after 2 clock cycles

we have:

- $X_0 = X_1 = X_2 = X_3 = 0$.
- $R_1(2)$=0x38A538A5 and $R_2(2)$=0x34813481.
- $S_{16} = 2^{31} - 1$.
- $(S_0, S_1, \ldots, S_{15}) = (0, 0, \ldots$,0,0x7FFFFFFF,0x7FFFFFFF).
- (R_1, R_2)=(0x38A538A5,0x34813481).

Thus all the activated bits, i.e., the bits that are set to "1" in the above process, belong to S_{14}, S_{15}, R_1 and R_2.

(b) For $i = 0 \ldots 7$: set $S_0(0) = 2^i$, $S_1(0) = \ldots = S_{15}(0)$=0 and $R_1(0) = R_2(0) = 0$. Then after 2 clock cycles we have:

- $X_0 = X_1 = X_2 = X_3 = 0$.
- $R_1(2)$=0x38A538A5 and $R_2(2)$=0x34813481.
- $S_{16} = 2^{15+i} + 2^{23+i}$.
- $(S_0, S_1, \ldots, S_{15}) = (0, 0, \ldots, 0, 2^i + 2^{i+8}, 2^{15+i} + 2^{23+i})$.
- (R_1, R_2)=(0x38A538A5,0x34813481).

The values of R_1 and R_2 remain the same as in step 2a. On the other hand, two bits of S_{14} will change which allows us to determine the position of the $(i + 9)^{th}, i = 0 \ldots 7$, bit in S_{14} which corresponds to 2^{i+8} in the equations above. Consequently, at the end of this step we have

$$\overset{\checkmark}{S_{14}^{9}}\ \overset{\checkmark}{S_{14}^{10}}, \overset{\checkmark}{S_{14}^{11}}, \overset{\checkmark}{S_{14}^{12}}, \overset{\checkmark}{S_{14}^{13}}, \overset{\checkmark}{S_{14}^{14}}, \overset{\checkmark}{S_{14}^{15}}, \overset{\checkmark}{S_{14}^{16}}$$

(c) For $i = 1 \ldots 7$: set $S_{15}(0) = 2^i$, $S_0(0) = \ldots = S_{14}(0)$=0 and $R_1(0) = R_2(0) = 0$. Then after 2 clock cycles we have:

- $X_0 \neq 0, X_1 = X_2 = X_3 = 0$.
- $R_1(2)$=0x38A538A5 and $R_2(2)$=0x34813481.
- $S_{16} = 2^{i+17}$.
- $(S_0, S_1, \ldots, S_{14}) = (0, 0, \ldots, 2^i, 2^{i+15})$.
- (R_1, R_2)=(0x38A538A5,0x34813481).

Again, the positions of activated bits in R_1 and R_2 do not change from step 2a. However the $(i+16)^{th}, i = 1 \ldots 7$, bit of S_{14}, which corresponds to 2^{i+15} in the equations above, is changed which allows us to determine the exact positions of these bits. Thus at the end of this step we have

$$\overset{\checkmark}{S_{14}^{17}}, \overset{\checkmark}{S_{14}^{18}}, \overset{\checkmark}{S_{14}^{19}}, \overset{\checkmark}{S_{14}^{20}}, \overset{\checkmark}{S_{14}^{21}}, \overset{\checkmark}{S_{14}^{22}}, \overset{\checkmark}{S_{14}^{23}}$$

3. **Determining the bit locations of S_{13}:**

(a) Set $S_0(0) = \ldots = S_{15}(0)$=0 and $R_1(0) = R_2(0) = 0$. Then after 3 clock cycles we have:

- X_0 =0xFFFFFFFF,$X_1 = X_2 = X_3 = 0$.
- $R_1(3)$=0x5334B0EC and $R_2(3)$=0xFE03AA92.
- $S_{16} = 2^{31} - 1$.
- $(S_0, S_1, \ldots, S_{15})$ = (0, 0, . . .,0,0x7FFFFFFF,0x7FFFFFFF,

0x7FFFFFFF).

- (R_1, R_2)=(0x5334B0EC,0xFE03AA92).

Thus all the activated bits, i.e. the bits that are set to "1" in the above process, belong to $S_{13}, S_{14}, S_{15}, R_1$ and R_2.

(b) For $i = 0 \ldots 7$: set $S_0(0) = 2^i$, $S_1(0) = \ldots = S_{15}(0)$=0 and $R_1(0) = R_2(0) = 0$. Then after 3 clock cycles we have:

- $X_0 \neq 0, X_1 = X_2 = X_3 = 0$.
- $R_1(3)$=0x5334B0EC and $R_2(3)$=0xFE03AA92.
- $S_{16} = 2^{15+i} + 2^{23+i}$.
- $(S_0, S_1, \ldots, S_{13}) = (0, 0, \ldots, 0, 2^i + 2^{i+8})$.
- (R_1, R_2)=(0x5334B0EC,0xFE03AA92).

The values of R_1 and R_2 remain the same as in step 3a. On the other hand, two bits of S_{13} are changed during the clocks which allows us to determine the position of the $(i+9)^{th}, i = 0 \ldots 7$, bit in S_{13} which corresponds to 2^{i+8} in the equations above. Consequently, at the end of this step we have

$$\overset{\checkmark}{S_{13}^{9}}\ \overset{\checkmark}{S_{13}^{10}}, \overset{\checkmark}{S_{13}^{11}}, \overset{\checkmark}{S_{13}^{12}}, \overset{\checkmark}{S_{13}^{13}}, \overset{\checkmark}{S_{13}^{14}}, \overset{\checkmark}{S_{13}^{15}}, \overset{\checkmark}{S_{13}^{16}}$$

(c) For $i = 1 \ldots 7$: set $S_{15}(0) = 2^i$, $S_0(0) = \ldots = S_{14}(0)$=0 and $R_1(0) = R_2(0) = 0$. Then after 3 clock cycles we have:

- $X_0 \neq 0, X_1 = X_2 = X_3 = 0$.

- $R_1(3)$=0x5334B0EC and $R_2(3)$=0xFE03AA92.
- $S_{16} = 2^{i+17}$.
- $(S_0, S_1, \ldots, S_{13}) = (0, 0, \ldots, 2^i, 2^{i+15})$.
- (R_1, R_2)=(0x5334B0EC,0xFE03AA92).

Again, the positions of activated bits in R_1 and R_2 do not change from step 3a. However the $(i+16)^{th}, i = 1 \ldots 7$, bit of S_{13}, which corresponds to 2^{i+15} in the equations above, is changed which allows us to determine the exact positions of these bits. Thus at the end of this step we have

$$\overset{\checkmark}{S_{13}^{17}}, \overset{\checkmark}{S_{13}^{18}}, \overset{\checkmark}{S_{13}^{19}}, \overset{\checkmark}{S_{13}^{20}}, \overset{\checkmark}{S_{13}^{21}}, \overset{\checkmark}{S_{13}^{22}}, \overset{\checkmark}{S_{13}^{23}}$$

4. **Determining the bit locations of S_{12}:**

(a) Set $S_0(0) = \ldots = S_{15}(0)$=0 and $R_1(0) = R_2(0) = 0$. Then after 4 clock cycles we have:

- X_0 =0xFFFFFFFF,$X_1 = X_2 = X_3 = 0$.
- $R_1(4)$=0x39A8912E and $R_2(4)$=0x14AE6F5C.
- $S_{16} = 2^{31} - 1$.
- $(S_0, S_1, \ldots, S_{15}) = (0, 0, \ldots$,0,0x7FFFFFFF,0x7FFFFFFF, 0x7FFFFFFF,0x7FFFFFFF).
- (R_1, R_2)=(0x39A8912E,0x14AE6F5C).

Thus all the activated bits, i.e. the bits that are set to "1" in the above process,

belong to $S_{12}, S_{13}, S_{14}, S_{15}, R_1$ and R_2.

(b) For $i = 0 \ldots 7$: set $S_0(0) = 2^i$, $S_1(0) = \ldots = S_{15}(0)$=0 and $R_1(0) = R_2(0) = 0$. Then after 4 clock cycles we have:

- $X_0 \neq 0, X_1 = X_2 = X_3 = 0$.
- $R_1(4)$=0x39A8912E and $R_2(4)$=0x14AE6F5C.
- $S_{16} = 2^{15+i} + 2^{23+i}$.
- $(S_0, S_1, \ldots, S_{12}) = (0, 0, \ldots, 0, 2^i + 2^{i+8})$.
- (R_1, R_2)=(0x39A8912E,0x14AE6F5C).

The values of R_1 and R_2 remain the same as in step 4a. On the other hand, two bits of S_{12} are changed during the clocks which allows us to determine the position of the $(i+9)^{th}, i = 0 \ldots 7$, bit in S_{12} which corresponds to 2^{i+8} in the equations above. Consequently, at the end of this step we have

$$\overset{\checkmark}{S_{12}^{9}}\ \overset{\checkmark}{S_{12}^{10}}, \overset{\checkmark}{S_{12}^{11}}, \overset{\checkmark}{S_{12}^{12}}, \overset{\checkmark}{S_{12}^{13}}, \overset{\checkmark}{S_{12}^{14}}, \overset{\checkmark}{S_{12}^{15}}, \overset{\checkmark}{S_{12}^{16}}$$

(c) For $i = 1 \ldots 7$: set $S_{15}(0) = 2^i$, $S_0(0) = \ldots = S_{14}(0)$=0 and $R_1(0) = R_2(0) = 0$. Then after 4 clock cycles we have:

- $X_0 \neq 0, X_1 = X_2 = X_3 = 0$.
- $R_1(4)$=0x39A8912E and $R_2(4)$=0x14AE6F5C.
- $S_{16} = 2^{i+17}$.
- $(S_0, S_1, \ldots, S_{12}) = (0, 0, \ldots, 2^i, 2^{i+15})$.

- (R_1, R_2)=(0x39A8912E,0x14AE6F5C).

Again, the positions of activated bits in R_1 and R_2 do not change from step 4a. However the $(i+16)^{th}, i = 1 \ldots 7$, bit of S_{12}, which corresponds to 2^{i+15} in the equations above, is changed which allows us to determine the exact positions of these bits. Thus at the end of this step we have

$$\overset{\checkmark}{S_{12}^{17}}, \overset{\checkmark}{S_{12}^{18}}, \overset{\checkmark}{S_{12}^{19}}, \overset{\checkmark}{S_{12}^{20}}, \overset{\checkmark}{S_{12}^{21}}, \overset{\checkmark}{S_{12}^{22}}, \overset{\checkmark}{S_{12}^{23}}$$

5. **Determining the bit locations of S_{11} and R_1^{22}:**

(a) Set $S_0(0) = \ldots = S_{15}(0)$=0 and $R_1(0) = R_2(0) = 0$. Then after 5 clock cycles we have:

- X_0 =0xFFFFFFFF,$X_1 = X_2 = X_3 = 0$.
- $R_1(5)$=0xBF289712 and $R_2(5)$=0x21513161.
- $S_{16} = 2^{31} - 1$.
- $(S_0, S_1, \ldots, S_{15}) = (0, 0, \ldots$,0,0x7FFFFFFF,0x7FFFFFFF, 0x7FFFFFFF,0x7FFFFFFF,0x7FFFFFFF,0x7FFFFFFF).
- (R_1, R_2)=(0xBF289712,0x21513161).

Thus all the activated bits, i.e., the bits that are set to "1" in the above process, belong to $S_{11}, S_{12}, S_{13}, S_{14}, S_{15}, R_1$ and R_2. At the end of this step, the 22^{th} bit of R_1 can be determined by matching and comparing with the previous values of R_1 ($\overset{\checkmark}{R_1^{22}}$).

(b) For $i = 0 \ldots 7$: set $S_0(0) = 2^i$, $S_1(0) = \ldots = S_{15}(0)$=0 and $R_1(0) = R_2(0) = 0$. Then after 5 clock cycles we have:

- $X_0 \neq 0, X_1 = X_2 = X_3 = 0$.
- $R_1(5)$=0xBF289712 and $R_2(5)$=0x21513161.
- $S_{16} = 2^{15+i} + 2^{23+i}$.
- $(S_0, S_1, \ldots, S_{11}) = (0, 0, \ldots, 0, 2^i + 2^{i+8})$.
- (R_1, R_2)=(0xBF289712,0x21513161).

The values of R_1 and R_2 remain the same as in step 5a. On the other hand, two bits of S_{11} are changed during the clocks which allows us to determine the position of the $(i+9)^{th}, i = 0 \ldots 7$, bit in S_{11} which corresponds to 2^{i+8} in the equations above. Consequently, at the end of this step we have

$$\overset{\checkmark}{S_{11}^{9}}\ \overset{\checkmark}{S_{11}^{10}}, \overset{\checkmark}{S_{11}^{11}}, \overset{\checkmark}{S_{11}^{12}}, \overset{\checkmark}{S_{11}^{13}}, \overset{\checkmark}{S_{11}^{14}}, \overset{\checkmark}{S_{11}^{15}}, \overset{\checkmark}{S_{11}^{16}}$$

(c) For $i = 16 \ldots 22$: set $S_{12}(0) = 2^i$, $S_0(0) = \ldots = S_{11}(0) = S_{13}(0) = S_{14}(0) = S_{15}(0)$=0 and $R_1(0) = R_2(0) = 0$. Then after 1 clock cycle we have:

- $X_1 = X_2 = 0$.
- $R_1(1) = R_2(1)$=0x3E553E55.
- $S_{16} = 2^{31} - 1$.
- $(S_0, S_1, \ldots, S_{15}) = (0, 0, \ldots, 0, 2^i, 0, 0, 0, 2^{31} - 1)$.
- (R_1, R_2)=(0x3E553E55,0x3E553E55).

Again, the positions of activated bits in R_1 and R_2 do not change from step 1a. However, the $(i+1)^{th}, i = 16 \ldots 22$, bit of S_{11}, which corresponds to 2^i in the equations above, is changed which allows us to determine the exact positions of these bits. Thus at the end of this step we have

$$\overset{\checkmark}{S_{11}^{17}}, \overset{\checkmark}{S_{11}^{18}}, \overset{\checkmark}{S_{11}^{19}}, \overset{\checkmark}{S_{11}^{20}}, \overset{\checkmark}{S_{11}^{21}}, \overset{\checkmark}{S_{11}^{22}}, \overset{\checkmark}{S_{11}^{23}}$$

6. **Determining the bit locations of S_{10} and R_2^{32}:**

   (a) For $i = 8 \ldots 22$: set $S_{11}(0) = 2^i$, $S_0(0) = \ldots = S_{10}(0) = S_{12}(0) = \ldots = S_{15}(0)$=0 and $R_1(0) = R_2(0) = 0$. Then after 1 clock cycle we have:

   - $X_1 = 2^{i+16}$, and $X_2 = 0$.
   - $(S_0, S_1, \ldots, S_{10}) = (0, 0, \ldots, 0, 2^i)$.
   - $R_1(1)$=0x3E553E55 and $R_2(1) \neq 0$ (For the first 8 values of *i*, $i = 8 \ldots 15$)
     - $i = 8, R_2(1)$=0x762072DD $\Longrightarrow \overset{\checkmark}{S_{10}^{9}}$
     - $i = 9, R_2(1)$=0xB1AE5BAD $\Longrightarrow \overset{?}{S_{10}^{10}}$ and $\overset{?}{R_2^{32}}$
     - $i = 10, R_2(1)$=0x723BE0C2 $\Longrightarrow \overset{\checkmark}{S_{10}^{11}}$
     - $i = 11, R_2(1)$=0x5B9F5463 $\Longrightarrow \overset{\checkmark}{S_{10}^{12}}$
     - $i = 12, R_2(1)$=0xCA8CAF3B $\Longrightarrow \overset{?}{S_{10}^{13}}$ and $\overset{?}{R_2^{32}}$, the position of R_2^{32} is activated again. So, all positions of S_{10}^{10}, R_2^{32} and S_{10}^{13} can be determined: $\overset{\checkmark}{S_{10}^{10}}, \overset{\checkmark}{S_{10}^{13}}$ and $\overset{\checkmark}{R_2^{32}}$.
     - $i = 13, R_2(1)$=0x0444DD9F $\Longrightarrow \overset{\checkmark}{S_{10}^{14}}$

- $i = 14, R_2(1)$=0x7BDDE38C $\Longrightarrow \overset{\checkmark}{S_{10}^{15}}$
- $i = 15, R_2(1)$=0x4DADBC44 $\Longrightarrow \overset{\checkmark}{S_{10}^{16}}$

- $R_1(1) = R_2(1)$ =0x3E553E55 (For the last 7 values of $i, i = 16 \ldots 22$). Again, the positions of activated bits in R_1 and R_2 do not change from step 1a. However the $(i+1)^{th}, i = 16 \ldots 22$, bit of S_{10}, which corresponds to 2^i in the equations above, is changed which allows us to determine the exact positions of these bits. Thus at the end of this step we have

$$\overset{\checkmark}{S_{10}^{17}}, \overset{\checkmark}{S_{10}^{18}}, \overset{\checkmark}{S_{10}^{19}}, \overset{\checkmark}{S_{10}^{20}}, \overset{\checkmark}{S_{10}^{21}}, \overset{\checkmark}{S_{10}^{22}}, \overset{\checkmark}{S_{10}^{23}}$$

Finally, we can find all unknown positions of S_{10} and the most significant bit place of R_2 at the end of this step.

7. **Determining the bit locations of S_9:**

(a) For $i = 8 \ldots 22$: set $S_{10}(0) = 2^i$, $S_0(0) = \ldots = S_9(0) = S_{11}(0) = \ldots = S_{15}(0)$=0 and $R_1(0) = R_2(0) = 0$. Then after 1 clock cycle we have:

- $X_1 = X_2 = 0$.
- $R_1(1) = R_2(1)$=0x3E553E55.
- $(S_0, S_1, \ldots, S_9) = (0, 0, \ldots, 0, 2^i)$.
- (R_1, R_2)=(0x3E553E55,0x3E553E55).

Again, the positions of activated bits in R_1 and R_2 do not change from step 1a. However the $(i+1)^{th}, i = 8 \ldots 22$, bit of S_9, which corresponds to 2^i in the

equations above, is changed which allows us to determine the exact positions of these bits. Thus at the end of this step we have

$$\overset{\checkmark}{S_9^9} \ldots, \overset{\checkmark}{S_9^{23}}$$

8. **Determining the bit locations of S_8, R_1^{15} and R_1^{18}:**

   (a) For $i = 8 \ldots 22$: set $S_9(0) = 2^i$, $S_0(0) = \ldots = S_8(0) = S_{10}(0) = \ldots = S_{15}(0)$=0 and $R_1(0) = R_2(0) = 0$. Then after 1 clock cycle we have:

   - $(S_0, S_1, \ldots, S_8) = (0, 0, \ldots, 0, 2^i)$.
   - $R_1(1) = R_2(1)$ =0x3E553E55 (For $i = 8 \ldots 14$).

     The values of R_1 and R_2 remain the same as in step 1a. On the other hand, two bits of S_8 are changed during the clocks which allows us to determine the position of the $(i+1)^{th}, i = 8 \ldots 14$, bit in S_8 which corresponds to 2^i in the equations above. Consequently, at the end of this step we have

$$\overset{\checkmark}{S_8^9}\ \overset{\checkmark}{S_8^{10}}, \overset{\checkmark}{S_8^{11}}, \overset{\checkmark}{S_8^{12}}, \overset{\checkmark}{S_8^{13}}, \overset{\checkmark}{S_8^{14}}, \overset{\checkmark}{S_8^{15}}$$

   - $R_2(1)$=0x3E553E55 and $R_1(1) \neq 0$ (For $i = 15, 16, 17$)
     - $X_1 = 2^{i-15}$, and $X_2 = 0$.
     - $i = 15, R_1(1)$=0xCAC8723B $\Longrightarrow \overset{?}{S_8^{16}}$ and $\overset{?}{R_1^{15}}$.
     - $i = 16, R_1(1)$=0x04DA5B9F $\Longrightarrow \overset{?}{S_8^{17}}$, $\overset{?}{R_1^{15}}$ and $\overset{?}{R_1^{18}}$. So, the other three places are activated. Then, the position of R_1^{15} can be determined

($\overset{\checkmark}{R_1^{15}}$). Besides, the position of S_8^{16} can be found easily after finding the position of R_1^{15} as well ($\overset{\checkmark}{S_8^{16}}$). However, positions of S_8^{17} and R_1^{18} are still unknown.

– $i = 17, R_1(1)$=0x7BA6CA8C $\Longrightarrow \overset{?}{S_8^{18}}$ and $\overset{?}{R_1^{18}}$, two positions are activated where R_1^{18} was just activated in the previous step. Then, R_1^{18} can be determined ($\overset{\checkmark}{R_1^{18}}$). Finally, S_8^{18} can be found easily after finding the position of R_1^{18} ($\overset{\checkmark}{S_8^{18}}$).

- For the last 5 values of $i = 18 \ldots 22$, all positions of $R_1(1)$ and $R_2(1)$ are already distinguishable based on their previous values. the $(i+1)^{th}, i = 18 \ldots 22$, bit of S_8, which corresponds to 2^i, is changed which allows us to determine the exact positions of these bits. Thus at the end of this step we have

$$\overset{\checkmark}{S_8^{19}}\ \overset{\checkmark}{S_8^{20}}, \overset{\checkmark}{S_8^{21}}, \overset{\checkmark}{S_8^{22}}, \overset{\checkmark}{S_8^{23}}, \overset{\checkmark}{R_1^{15}}, \overset{\checkmark}{R_1^{18}}$$

Thus, at the end of this step we are able to find all unknown positions of S_8 and two bit locations of R_1.

9. **Determining the bit locations of $S_0 \ldots S_7$:**

(a) For $j = 8 \ldots 1$

For $i = 8 \ldots 22$, set:

$S_j(0) = 2^i$, $S_0(0) = \ldots = S_{j-1}(0) = S_{j+1}(0) = \ldots = S_{15}(0)$=0 and $R_1(0) = R_2(0) = 0$. Then after 1 clock cycle we have:

- $X_1 = 0$.
- $(S_0, S_1, \ldots, S_j) = (0, 0, \ldots, 2^i)$.
- $R_1(1) \neq 0$ and $R_2(1) \neq 0$. All of activated positions of R_1 and R_2 are already distinguishable from the other registers based on their previous values.

Again, the positions of activated bits in R_1 and R_2 do not change from previous steps. However the $(i+1)^{th}, i = 8 \ldots 22$, bit of S_j,$j = 8 \ldots 1$, which corresponds to 2^i in the equations above, is changed which allows us to determine the exact positions of these bits. Thus at the end of this step we have

$$\overset{\checkmark}{S^9_{j-1}} \ldots, \overset{\checkmark}{S^{23}_{j-1}}$$

5.4.4 Determining the location of the remaining bits in R_1 and R_2

In this section, we show how to determine the position of the remaining bits of the memory cells. To achieve this, first we reset all registers to 0. Then, by examining the scan out chain after clocking the circuit with a prespecified number of clock cycles, as shown below, we are able to determine the position of these bits.

1. **Determining the bit locations of $R_1^6, R_1^{23}, R_1^{26}, R_1^{28}, R_1^{31}$ and R_2^7:**

   Set $S_0(0) = \ldots = S_{15}(0)$=0 and $R_1(0) = R_2(0) = 0$. During 5 clock cycles we have:

   (a) $R_1(1)$= 0x3E553E55, $R_2(1)$= 0x3E553E55.

(b) $R_1(2)$= 0x38A538A5, $R_2(2)$= 0x34813481.

(c) $R_1(3)$= 0x5334B0EC, $R_2(3)$= 0x FE03AA92.

(d) $R_1(4)$= 0x39A8912E, $R_2(4)$= 0x14AE6F5C.

(e) $R_1(5)$= 0xBF289712, $R_2(5)$= 0x21513161.

The location of the "1" bit in

$$R_1(2)[1b] \wedge R_1(3)[1c] \wedge R_1(4)[1d] \wedge (\overline{R_1(1)[1a] \vee R_1(5)[1e]})$$

corresponds to the position of R_1^6 where $R_1(2)[1b]$ is the value of R_1 in the 2^{nd} clock of the process 1b. In other words, the formula illustrates that the position of R_1^6 can be determined uniquely provided that this specific location must be "1" in steps $1b, 1c$ and $1d$ and be "0" in steps $1a$ and $1e$ respectively. Similarly, the other bits can be located as presented in Table 4.

$R_1(1)[1a] \wedge (\overline{R_1(2)[1b] \vee R_1(3)[1c] \vee R_1(4)[1d] \vee R_1(5)[1e]})$	$\Longrightarrow R_1^{23}$
$R_1(1)[1a] \wedge R_1(3)[1c] \wedge R_1(5)[1e] \wedge (\overline{R_1(2)[1b] \vee R_1(4)[1d]})$	$\Longrightarrow R_1^{26}$
$R_1(1)[1a] \wedge R_1(2)[1b] \wedge R_1(4)[1d] \wedge R_1(5)[1e] \wedge (\overline{R_1(3)[1c]})$	$\Longrightarrow R_1^{28}$
$R_1(3)[1c] \wedge R_1(5)[1e] \wedge (\overline{R_1(1)[1a] \vee R_1(2)[1b] \vee R_1(4)[1d]})$	$\Longrightarrow R_1^{31}$
$R_2(1)[1a] \wedge R_2(4)[1d] \wedge R_2(5)[1e] \wedge (\overline{R_2(2)[1b] \vee R_2(3)[1c]})$	$\Longrightarrow R_2^{7}$

Table 4: Formulas used in step 1

2. **Determining the bit locations of** $R_1^{24}, R_2^1, R_2^2, R_2^{12}, R_2^{13}, R_2^{17}, R_2^{24}, R_2^{25}, R_2^{26}, R_2^{29}$ **and** R_2^{30}:

   Set $S_{11}(0) = 2^{15}$, $S_0(0) = \ldots = S_{10}(0) = S_{12}(0) = \ldots = S_{15}(0)$=0 and $R_1(0) = R_2(0) = 0$. Then after one clock cycle we have:

   (a) $R_1(1)$= 0x3E553E55, $R_2(1)$= 0x4DADBC44.

   The location of the "1" bit in

$$R_1(2)[1b] \wedge R_1(4)[1d] \wedge (\overline{R_1(1)[1a] \vee R_1(3)[1c] \vee R_1(5)[1e] \vee R_1(1)[2a]})$$

   corresponds to the position of R_1^{24} and it can be determined uniquely.

   Similarly, the location of the other bits can be determined as shown in Table 5.

3. **Determining the bit locations of** $R_1^1, R_1^3, R_1^4, R_1^{14}, R_1^{19}, R_1^{29}, R_1^{30}, R_1^{32}, R_2^6, R_2^{11}, R_2^{18}$, R_2^{27} **and** R_2^{28}:

   Set $S_9(0) = 2^{15}$, $S_0(0) = \ldots = S_8(0) = S_{10}(0) = \ldots = S_{15}(0)$=0 and $R_1(0) = R_2(0) = 0$. Then after 1 clock cycle we have:

   (a) $R_1(1)$= 0xCAC8723B, $R_2(1)$= 0x3E553E55.

   The location of the "1" bit in

$$R_1(1)[1a] \wedge R_1(2)[1b] \wedge R_1(1)[2a] \wedge R_1(1)[3a] \wedge (\overline{R_1(3)[1c] \vee R_1(4)[1d] \vee R_1(5)[1e]})$$

   corresponds to the position of R_1^1 and it can be determined uniquely.

$R_2(1)[1a] \wedge R_2(2)[1b] \wedge R_2(5)[1e] \wedge (\overline{R_2(3)[1c] \vee R_2(4)[1d] \vee R_2(1)[2a]}) \implies R_2^1$
$R_2(3)[1c] \wedge (\overline{R_2(1)[1a] \vee R_2(2)[1b] \vee R_2(4)[1d] \vee R_2(5)[1e] \vee R_2(1)[2a]}) \implies R_2^2$
$R_2(1)[1a] \wedge R_2(3)[1c] \wedge R_2(4)[1d] \wedge R_2(1)[2a] \wedge (\overline{R_2(2)[1b] \vee R_2(5)[1e]}) \implies R_2^{12}$
$R_2(1)[1a] \wedge R_2(2)[1b] \wedge R_2(5)[1e] \wedge R_2(1)[2a] \wedge (\overline{R_2(3)[1c] \vee R_2(4)[1d]}) \implies R_2^{13}$
$R_2(1)[1a] \wedge R_2(2)[1b] \wedge R_2(3)[1c] \wedge R_2(5)[1e] \wedge R_2(1)[2a] \wedge (\overline{R_2(4)[1d]}) \implies R_2^{17}$
$R_2(2)[1b] \wedge R_2(4)[1d] \wedge R_2(1)[2a] \wedge (\overline{R_2(1)[1a] \vee R_2(3)[1c] \vee R_2(5)[1e]}) \implies R_2^{24}$
$R_2(5)[1e] \wedge R_2(1)[2a] \wedge (\overline{R_2(1)[1a] \vee R_2(2)[1b] \vee R_2(3)[1c] \vee R_2(4)[1d]}) \implies R_2^{25}$
$R_2(1)[1a] \wedge R_2(3)[1c] \wedge (\overline{R_2(2)[1b] \vee R_2(4)[1d] \vee R_2(5)[1e] \vee R_2(1)[2a]}) \implies R_2^{26}$
$R_2(1)[1a] \wedge R_2(2)[1b] \wedge R_2(3)[1c] \wedge R_2(4)[1d] \wedge (\overline{R_2(5)[1e] \vee R_2(1)[2a]}) \implies R_2^{29}$
$R_2(1)[1a] \wedge R_2(2)[1b] \wedge R_2(3)[1c] \wedge R_2(5)[1e] \wedge (\overline{R_2(4)[1d] \vee R_2(1)[2a]}) \implies R_2^{30}$

Table 5: Formulas used in step 2

Similarly, the location of the other bits can be determined as shown in Table 6.

4. **Determining the bit locations of** $R_1^2, R_1^7, R_1^8, R_1^9, R_1^{11}, R_1^{13}, R_1^{16}, R_1^{20}, R_1^{21}, R_1^{25}, R_1^{27}$, $R_2^3, R_2^8, R_2^9, R_2^{14}, R_2^{16}, R_2^{19}, R_2^{20}, R_2^{22}$ **and** R_2^{31}:

   Set $S_5 = 2^{16} + 2^{18} + 2^{20} + 2^{21}$=0x00350000, $S_9 = 2^{16} + 2^{20}$=0x00110000, $Sj = 0; \{j = 0 \ldots 15, j \neq 5, 9\ \}$ and $R_1(0) = R_2(0) = 0$. Then after 1 clock cycle we have:

   (a) $R_1(1)$= 0xCF571B36, $R_2(1)$= 0xC92206F5.

$R_1(1)[1a] \wedge R_1(2)[1b] \wedge R_1(3)[1c] \wedge R_1(4)[1d] \wedge R_1(1)[2a] \wedge (\overline{R_1(5)[1e]}) \vee R_1(1)[3a]$	$\Longrightarrow R_1^3$
$R_1(3)[1c] \wedge R_1(4)[1d] \wedge R_1(1)[3a] \wedge (\overline{R_1(1)[1a] \vee R_1(2)[1b] \vee R_1(5)[1e] \vee R_1(1)[2a]})$	$\Longrightarrow R_1^4$
$R_1(1)[1a] \wedge R_1(2)[1b] \wedge R_1(3)[1c] \wedge R_1(1)[2a] \wedge R_1(1)[3a] \wedge (\overline{R_1(4)[1d] \vee R_1(5)[1e]})$	$\Longrightarrow R_1^{14}$
$R_1(1)[1a] \wedge R_1(2)[1b] \wedge R_1(3)[1c] \wedge R_1(1)[2a] \wedge (\overline{\vee R_1(4)[1d] \vee R_1(5)[1e] \vee R_1(1)[3a]})$	$\Longrightarrow R_1^{19}$
$R_1(1)[1a] \wedge R_1(2)[1b] \wedge R_1(3)[1c] \wedge R_1(4)[1d] \wedge R_1(5)[1e] \wedge R_1(1)[2a] \wedge (\overline{R_1(1)[3a]})$	$\Longrightarrow R_1^{29}$
$R_1(1)[1a] \wedge R_1(2)[1b] \wedge R_1(4)[1d] \wedge R_1(1)[2a] \wedge (\overline{\vee R_1(3)[1c] \vee R_1(5)[1e] \vee R_1(1)[3a]})$	$\Longrightarrow R_1^{30}$
$R_1(5)[1a] \wedge R_1(1)[3a] \wedge (\overline{R_1(1)[1a] \vee R_1(2)[1b] \vee R_1(3)[1c] \vee R_1(4)[1d] \vee R_1(1)[2a]})$	$\Longrightarrow R_1^{32}$
$R_2(5)[1a] \wedge (\overline{R_2(1)[1a] \vee R_2(2)[1b] \vee R_2(3)[1c] \vee R_2(4)[1d] \vee R_2(1)[2a] \vee R_2(1)[3a]})$	$\Longrightarrow R_2^6$
$R_2(1)[1a] \wedge R_2(2)[1b] \wedge R_2(4)[1d] \wedge R_2(1)[2a] \wedge R_2(1)[3a] \wedge (\overline{R_2(3)[1c] \vee R_2(5)[1e]})$	$\Longrightarrow R_2^{11}$
$R_2(3)[1c] \wedge R_2(4)[1d] \wedge (\overline{R_2(1)[1a] \vee R_2(2)[1b] \vee R_2(5)[1e] \vee R_2(1)[2a] \vee R_2(1)[3a]})$	$\Longrightarrow R_2^{18}$
$R_2(1)[1a] \wedge R_2(2)[1b] \wedge R_2(3)[1c] \wedge R_2(4)[1d] \wedge R_2(1)[2a] \wedge R_2(1)[3a] \wedge (\overline{R_2(5)[1e]})$	$\Longrightarrow R_2^{27}$
$R_2(1)[1a] \wedge R_2(3)[1c] \wedge R_2(1)[2a] \wedge R_2(1)[3a] \wedge (\overline{R_2(2)[1b] \vee R_2(4)[1d] \vee R_2(5)[1e]})$	$\Longrightarrow R_2^{28}$

Table 6: Formulas used in step 3

The location of the "1" bit in

$$R_1(4)[1d] \wedge R_1(5)[1e] \wedge R_1(1)[3a] \wedge R_1(1)[4a] \wedge$$
$$(\overline{R_1(1)[1a] \vee R_1(2)[1b] \vee R_1(3)[1c] \vee R_1(1)[2a]})$$

corresponds to the position of R_1^2 and it can be determined uniquely. Similarly, the other bits can be located as depicted in Table 7.

5. **Determining the bit locations of** $R_1^5, R_1^{10}, R_1^{12}, R_1^{17}, R_2^4, R_2^5, R_2^{10}, R_2^{15}, R_2^{21}$ **and** R_2^{23}: Set S_5=0x011700000, S_9=0x01110000, $Sj = 0; \{j = 0 \ldots 15, j \neq 5, 9\ \}$ and $R_1(0) = R_2(0) = 0$. Then after 1 clock cycle we have:

(a) $R_1(1)$= 0x7D9C9DDD, $R_2(1)$=0x9E4C22CB.

Table 8 shows how the remaining bits can be located.

Note that the computations performed in this section can be performed simultaneously, i.e., in parallel, by observing the scan out results in section 5.4.3. In other words, these steps do not require any extra clocking for the circuit.

5.5 Summary and conclusions

In this chapter, we presented a scan based side channel attack on ZUC. Our attack allows the cryptanalyst to determine the bit positions corresponding to the cipher secret internal state in the scan out chain. To do so, we first used the key loading procedure to determine the least and the most significant byte locations of the LFSR register. Then we utilized the working mode procedure to determine the positions corresponding to the remaining bits in the LFSR and the memory cells R_1 and R_2.

$$R_1(1)[1a] \wedge R_1(3)[1c] \wedge R_1(1)[2a] \wedge (\overline{R_1(2)[1b] \vee R_1(4)[1d] \vee R_1(5)[1e] \vee R_1(1)[3a] \vee R_1(1)[4a]}) \implies R_1^7$$

$$R_1(2)[1b] \wedge R_1(3)[1c] \wedge (\overline{R_1(1)[1a] \vee R_1(4)[1d] \vee R_1(5)[1e] \vee R_1(1)[2a] \vee R_1(1)[3a] \vee R_1(1)[4a]}) \implies R_1^8$$

$$R_1(4)[1d] \wedge R_1(5)[1e] \wedge R_1(1)[4a] \wedge (\overline{R_1(1)[1a] \vee R_1(2)[1b] \vee R_1(3)[1c] \vee R_1(1)[2a] \vee R_1(1)[3a]}) \implies R_1^9$$

$$R_1(1)[1a] \wedge R_1(5)[1e] \wedge R_1(1)[2a] \wedge (\overline{R_1(2)[1b] \vee R_1(3)[1c] \vee R_1(4)[1d] \vee R_1(1)[3a] \vee R_1(1)[4a]}) \implies R_1^{11}$$

$$R_1(1)[1a] \wedge R_1(2)[1b] \wedge R_1(3)[1c] \wedge R_1(4)[1d] \wedge R_1(5)[1e] \wedge R_1(1)[2a] \wedge R_1(1)[3a] \wedge R_1(1)[4a] \implies R_1^{13}$$

$$R_1(3)[1c] \wedge R_1(4)[1d] \wedge R_1(5)[1e] \wedge (\overline{R_1(1)[1a] \vee R_1(2)[1b] \vee R_1(1)[2a] \vee R_1(1)[3a] \vee R_1(1)[4a]}) \implies R_1^{16}$$

$$R_1(4)[1d] \wedge R_1(5)[1e] \wedge R_1(1)[3a] \wedge (\overline{R_1(1)[1a] \vee R_1(2)[1b] \vee R_1(3)[1c] \vee R_1(1)[2a] \vee R_1(1)[4a]}) \implies R_1^{20}$$

$$R_1(1)[1a] \wedge R_1(3)[1c] \wedge R_1(1)[2a] \wedge R_1(1)[4a] \wedge (\overline{R_1(2)[1b] \vee R_1(4)[1d] \vee R_1(5)[1e] \vee R_1(1)[3a]}) \implies R_1^{21}$$

$$R_1(3)[1c] \wedge R_1(4)[1d] \wedge R_1(5)[1e] \wedge R_1(1)[4a] \wedge (\overline{R_1(1)[1a] \vee R_1(2)[1b] \vee R_1(1)[2a] \vee R_1(1)[3a]}) \implies R_1^{25}$$

$$R_1(1)[1a] \wedge R_1(5)[1e] \wedge R_1(1)[2a] \wedge R_1(1)[4a] \wedge (\overline{R_1(2)[1b] \vee R_1(3)[1c] \vee R_1(4)[1d] \vee R_1(1)[3a]}) \implies R_1^{27}$$

$$R_2(1)[1a] \wedge R_2(4)[1d] \wedge R_2(1)[2a] \wedge R_2(1)[3a] \wedge R_2(1)[4a] \wedge (\overline{R_2(2)[1b] \vee R_2(3)[1c] \vee R_2(5)[1e]}) \implies R_2^3$$

$$R_2(2)[1b] \wedge R_2(3)[1c] \wedge R_2(1)[4a] \wedge (\overline{R_2(1)[1a] \vee R_2(4)[1d] \vee R_2(5)[1e] \vee R_2(1)[2a] \vee R_2(1)[3a]}) \implies R_2^8$$

$$R_2(4)[1d] \wedge R_2(5)[1e] \wedge (\overline{R_2(1)[1a] \vee R_2(2)[1b] \vee R_2(3)[1c] \vee R_2(1)[2a] \vee R_2(1)[3a] \vee R_2(1)[4a]}) \implies R_2^9$$

$$R_2(1)[1a] \wedge R_2(2)[1b] \wedge R_2(3)[1c] \wedge R_2(4)[1d] \wedge R_2(5)[1e] \wedge R_2(1)[2a] \wedge R_2(1)[3a] \wedge \overline{R_2(1)[4a]} \implies R_2^{14}$$

$$R_2(3)[1c] \wedge R_2(1)[2a] \wedge (\overline{R_2(1)[1a] \vee R_2(2)[1b] \vee R_2(4)[1d] \vee R_2(5)[1e] \vee R_2(1)[3a] \vee R_2(1)[4a]}) \implies R_2^{16}$$

$$R_2(1)[1a] \wedge R_2(4)[1d] \wedge R_2(1)[2a] \wedge R_2(1)[3a] \wedge (\overline{R_2(2)[1b] \vee R_2(3)[1c] \vee R_2(5)[1e] \vee R_2(1)[4a]}) \implies R_2^{19}$$

$$R_2(4)[1d] \wedge R_2(1)[2a] \wedge (\overline{R_2(1)[1a] \vee R_2(2)[1b] \vee R_2(3)[1c] \vee R_2(5)[1e] \vee R_2(1)[3a] \vee R_2(1)[4a]}) \implies R_2^{20}$$

$$R_2(4)[1d] \wedge R_2(1)[2a] \wedge R_2(1)[4a] \wedge (\overline{R_2(1)[1a] \vee R_2(2)[1b] \vee R_2(3)[1c] \vee R_2(5)[1e] \vee R_2(1)[3a]}) \implies R_2^{22}$$

$$R_2(3)[1c] \wedge R_2(1)[2a] \wedge R_2(1)[4a] \wedge (\overline{R_2(1)[1a] \vee R_2(2)[1b] \vee R_2(4)[1d] \vee R_2(5)[1e] \vee R_2(1)[3a]}) \implies R_2^{31}$$

Table 7: Formulas used in step 4

$R_1(1)[1a] \wedge R_1(5)[1e] \wedge R_1(1)[2a] \wedge R_1(1)[3a] \wedge R_1(1)[4a] \wedge R_1(1)[5a] \wedge (\overline{R_1(2)[1b] \vee R_1(3)[1c] \vee R_1(4)[1d]})$	$\Longrightarrow R_1^5$
$R_1(1)[1a] \wedge R_1(5)[1e] \wedge R_1(1)[2a] \wedge R_1(1)[3a] \wedge R_1(1)[4a] \wedge (\overline{R_1(2)[1b] \vee R_1(3)[1c] \vee R_1(4)[1d] \vee R_1(1)[5a]})$	$\Longrightarrow R_1^{10}$
$R_1(1)[1a] \wedge R_1(2)[1b] \wedge R_1(1)[2a] \wedge R_1(1)[4a] \wedge R_1(1)[5a] \wedge (\overline{R_1(3)[1c] \vee R_1(4)[1d] \vee R_1(5)[1e] \vee R_1(1)[4a]})$	$\Longrightarrow R_1^{12}$
$R_1(1)[1a] \wedge R_1(2)[1b] \wedge R_1(1)[2a] \wedge R_1(1)[4a] \wedge (\overline{R_1(3)[1c] \vee R_1(4)[1d] \vee R_1(5)[1e] \vee R_1(1)[4a] \vee R_1(1)[5a]})$	$\Longrightarrow R_1^{17}$
$R_2(4)[1d] \wedge R_2(1)[5a] \wedge (\overline{R_2(1)[1a] \vee R_2(2)[1b] \vee R_2(3)[1c] \vee R_2(5)[1e] \vee R_2(1)[2a] \vee R_2(1)[3a] \vee R_2(1)[4a]})$	$\Longrightarrow R_2^4$
$R_2(1)[1a] \wedge R_2(3)[1c] \wedge R_2(4)[1d] \wedge R_2(1)[3a] \wedge R_2(1)[4a] \wedge (\overline{R_2(2)[1b] \vee R_2(5)[1e] \vee R_2(1)[2a] \vee R_2(1)[5a]})$	$\Longrightarrow R_2^5$
$R_2(1)[1a] \wedge R_2(3)[1c] \wedge R_2(4)[1d] \wedge R_2(1)[3a] \wedge R_2(1)[4a] \wedge R_2(1)[5a] \wedge (\overline{R_2(2)[1b] \vee R_2(5)[1e] \vee R_2(1)[2a]})$	$\Longrightarrow R_2^{10}$
$R_2(4)[1d] \wedge (\overline{R_2(1)[1a] \vee R_2(2)[1b] \vee R_2(3)[1c] \vee R_2(5)[1e] \vee R_2(1)[2a] \vee R_2(1)[3a] \vee R_2(1)[4a] \vee R_2(1)[5a]})$	$\Longrightarrow R_2^{15}$
$R_2(1)[1a] \wedge R_2(5)[1e] \wedge R_2(1)[3a] \wedge (\overline{R_2(2)[1b] \vee R_2(3)[1c] \vee R_2(4)[1d] \vee R_2(1)[2a] \vee R_2(1)[4a] \vee R_2(1)[5a]})$	$\Longrightarrow R_2^{21}$
$R_2(1)[1a] \wedge R_2(5)[1e] \wedge R_2(1)[3a] \wedge R_2(1)[5a] \wedge (\overline{R_2(2)[1b] \vee R_2(3)[1c] \vee R_2(4)[1d] \vee R_2(1)[2a] \vee R_2(1)[4a]})$	$\Longrightarrow R_2^{23}$

Table 8: Formulas used in step 5

Chapter 6

Conclusions and Future Work

6.1 Summary

Throughout this work, we presented side channel cryptanalytic attacks against three symmetric key ciphers: (i) SOSEMANUK, which is a software-based stream cipher of the eSTREAM software portfolio, (ii) Hummingbird, which is a hybrid structure of block cipher and stream ciphers dedicated for ultra-lightweight encryption and (iii) ZUC, which is a new stream cipher proposed for the 4G mobile standard.

The overall complexity of our differential fault analysis attack against SOSEMANUK can be summarized as follows: The average number of faults required to perform the attack is $4 \times 1536 = 6144$. This fault injection phase requires the attacker to reinitialize the cipher for 6144 times. The attack requires work equivalent to around 2^{48} iterations. The storage requirements amounts to $2^{38.17}$ bytes.

Our attack against Hummingbird is inspired by the currently known attacks against

AES. However, unlike AES, the key schedule properties of Hummingbird cannot be utilized by the attacker to further reduce the complexity of the attack. This forces the attacker to iterate the fault injection on every round of the 16-bit block cipher algorithm to recover the whole key. Each 64-bit round key can be found, on average, using 12.51 faulty encryptions. If we assume that the 256-bit secret key and 80-bit internal state have random distribution, then the whole cipher can be broken after around 50 faults. To fully recover the key, we have to guess 2^{48} values of three 16-bit internal state registers which brings the whole complexity of our attack to $O(2^{68})$.

Finally, we presented a scan based side channel attack against ZUC. Under reasonable assumptions, our cryptanalysis allows the attacker to ascertain the whole location of the internal registers of the LFSR and the memory cells. Consequently, this attack allows the cryptanalyst to efficiently recover the cipher secret internal state in a relatively small number of clock cycles.

6.2 Future work

When compared to other stream ciphers in the equivalent fault analysis model, DFA of SOSEMANUK requires a relatively smaller number of faults. For example, the DFA attack on RC4 given in [9] requires 2^{16} faults in random locations of its inner state. Another DFA attack on HC-128 [10] requires around 2^{13} faults in random locations. In future work, it will be interesting to see whether the number of faults for the DFA of SOSEMANUK and other stream ciphers can be drastically decreased in the assumed fault model.

A naive approach to prevent our attack is to use algorithm level redundancy and disable the device output if the two produced key stream values do not match. Another more efficient approach, which partially protects against fault attacks, is to add parity bits to all the inner state resisters and disable the device output if any of these parity checks is violated. Efficient fault analysis resistant implementations for SOSEMANUK, as well as for other stream ciphers, need to be addressed in future research.

Another related research direction is the exploration of different side channel attacks against the SHA-3 finalist [93] when operating in the MAC mode.

Developing cryptosystems that are inherently secure against side channel attacks, by design, is also a very challenging research direction.

Bibliography

[1] D. Kahn, *The Codebreakers: The Story of Secret Writing*, Macmillan Pub Co; Reissue edition, 1974.

[2] A.J. Menezes, P.C. Van Oorschot, and S.A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 5th edition, 2001.

[3] P. Kocher, *Timing attacks on implementations of Diffie-Hellmann, RSA, DSS, and other systems*, In proc. of CRYPTO '96, LNCS 1109, pp. 104-113, Springer-Verlag, 1996.

[4] P. Kocher, J. Jaffe, and B. Jun, *Differential power analysis*, In proc. of CRYPTO '99, LNCS 1666, pp. 388-397, Springer-Verlag, 1999.

[5] S.P. Skorobogatov and R.J. Anderson, *Optical fault induction attacks*, In proc. of CHES 2003, LNCS 2523, pp. 2-12, Springer-Verlag, 2003.

[6] D. Boneh, R.A. DeMillo, and R.J. Lipton, *On the importance of checking cryptographic protocols for faults*, In proc. of EUROCRYPT '97, LNCS 1233, pp. 37-51, Springer-Verlag, 1997.

[7] E. Biham and A. Shamir, *Differential Fault Analysis of Secret Key Cryptosystems*, In proc. of CRYPTO '97, LNCS 1294, pp. 513-525, Springer-Verlag, 1997.

[8] P. Dusart, G. Letourneux, and O. Vivolo, *Differential fault analysis on AES*, In proc. of Applied Cryptography and Network Security (ACNS) 2003, LNCS 2846, pp. 293-306, Springer-Verlag, 2003.

[9] J. Hoch and A. Shamir, *Fault Analysis of Stream Ciphers*, In proc. of CHES 2004, LNCS 3156, pp. 240-253, Springer-Verlag, 2004.

[10] A. Kircanski and A.M. Youssef, *Differential Fault Analysis of HC-128*, In proc. of AFRICACRYPT 2010, LNCS 6055, pp. 261-278, Springer-Verlag, 2010.

[11] A. Kircanski and A.M. Youssef, *Differential Fault Analysis of Rabbit*, In proc. of Selected Areas in Cryptography (SAC) 2009, LNCS 5867, pp. 197-214, Springer-Verlag, 2009.

[12] A. Berzati, C. Canovas-Dumas, and L. Goubin, *Fault Analysis of Rabbit: Towards a Secret-Key Leakage*, In proc. of INDOCRYPT 2009, LNCS 5922, pp. 72-87, Springer-Verlag, 2009.

[13] D. Engels, X. Fan, G. Gong, H. Hu and E. M. Smith, *Hummingbird: Ultra-Lightweight Cryptography for Resource-Constrained Devices*, In proc. of Financial Cryptography (FC) 2010, LNCS 6054, pp. 3-18, Springer-Verlag, 2010.

[14] Specification of the 3GPP Confidentiality and Integrity Algorithms 128-EEA3 & 128-EIA3. Document 2: ZUC Specification, EEA3 IEA3 ZUC v1.4, July 2010.

[15] Specification of the 3GPP Confidentiality and Integrity Algorithms 128-EEA3 & 128-EIA3. Document 2: ZUC Specification, v1.5, January 2011.

[16] C. Berbain, O. Billet, A. Canteaut, N. Courtois, H. Gilbert, L. Goubin, A. Gouget, L. Granboulan, C. Lauradoux, M. Minier, T. Pornin, and H. Sibert, SOSEMANUK*, a fast software-oriented stream cipher*, The eSTREAM Finalists, LNCS 4986, pp. 98-118, Springer-Verlag, 2008. Also available at http://www.ecrypt.eu.org/stream/sosemanukp3.html

[17] X. Fan, H. Hu, G. Gong, E. M. Smith and D. Engels, *Lightweight Implementation of Hummingbird Cryptographic Algorithm on 4-Bit Microcontroller*, The 1st international workshop on RFID Security and Cryptography 2009 (RISC '09), pp. 838-844, 2009.

[18] Y. Esmaeili Salehani, A. Kircanski , and A.M. Youssef, *Differential Fault Analysis of* SOSEMANUK, In proc. of AFRICACRYPT 2011, LNCS 6737, pp. 316-331, Springer-Verlag, 2011.

[19] Y. Esmaeili Salehani and A.M. Youssef, *Differential Fault Analysis of Hummingbird*, In proc. of the International Conference on Security and Cryptography (SECRYPT) 2011, 2011.

[20] *Data Encryption Standard*, Federal Information Processing Standard (FIPS) 46, National Bureau of Standards, 1977.

[21] J. Daemen and V. Rijmen, *AES proposal: Rijndael. In AES Round 1 Technical Evaluation*, CD-1: Documentation. NIST, 1998. Available at http://www.esat.kuleuven.ac.be/ rijmen/rijndael/ or http://www.nist.gov/aes.

[22] D. Wheeler and R. Needham, *TEA, a Tiny Encryption Algorithm*, In proc. of Fast Software Encryption (FSE) 1994, LNCS 1008, pp. 363-366, Springer-Verlag, 1994.

[23] D. Wheeler and R. Needham, *TEA extensions*, 1997. Available at www.ftp.cl.cam.ac.uk/ftp/users/djw3/

[24] C. Lim and T. Korkishko, *mCrypton - A Lightweight Block Cipher for Security of Low-cost RFID Tags and Sensors*, In proc. of Workshop on Information Security Applications (WISA) 2005, LNCS 3786, pp. 243-258, Springer-Verlag, 2005.

[25] D. Hong, J. Sung, S. Hong, J. Lim, S. Lee, B.-S; Koo, C. Lee, D. Chang, J. Lee, K. Jeong, H. Kim, J. Kim, and S. Chee, *HIGHT: A New Block Cipher Suitable for Low-Resource Device*, In proc. of CHES 2006, LNCS 4249, pp. 46-59, Springer-Verlag, 2006.

[26] F.-X. Standaert, G. Piret, N. Gershenfeld, and J.-J. Quisquater, *SEA: A Scalable Encryption Algorithm for Small Embedded Applications*, In proc. of Smart Card Research and Applications (CARDIS) 2006, LNCS 3928, pp. 222-236, Springer-Verlag, 2006.

[27] M.J.B. Robshaw, *Searching for compact algorithms: CGEN*, In proc. of Vietcrypt 2006, LNCS 4341, pp. 37-49, Springer-Verlag, 2006.

[28] A. Bogdanov, L.R.Knudsen, G. Leander, C. Paar, A. Poschmann, M.J.B. Robshaw, Y. Seurin, and C. Vikkelsoe, *PRESENT: An ultra-lightweight block cipher*, In proc. of CHES 2007, LNCS 4727, pp. 450-466, Springer-Verlag, 2007.

[29] M.I. Izadi, B. Sadeghiyan, S.S. Sadeghian, and H.A. Khanooki, *MIBS: a new lightweight Block Cipher*, In proc. of Cryptology and Network Security (CANS) 2009, LNCS 5888, pp. 334-348, Springer-Verlag, 2009.

[30] NESSIE, the New European Schemes for Signatures, Integrity and Encryption. Available at https://www.cosic.esat.kuleuven.be/nessie/

[31] ECRYPT, the European Network of Excellence for Cryptology. Available at http://www.ecrypt.eu.org/ecrypt1/

[32] eSTREAM, the ECRYPT Stream Cipher Project. Available at http://www.ecrypt.eu.org/stream/

[33] H. Wu, *The Stream Cipher HC-128*, The eSTREAM Finalists, LNCS 4986, pp. 39-47, Springer-Verlag, 2008. Also available at http://www.ecrypt.eu.org/stream/hcpf.html

[34] M. Boesgaard, M. Vesterager, and E. Zenner, *The Rabbit Stream Cipher*, The eSTREAM Finalists, LNCS 4986, pp. 69-83, Springer-Verlag, 2008. Also available at http://www.ecrypt.eu.org/stream/rabbitpf.html

[35] D.J. Bernstein, *The Salsa20 Family of Stream Ciphers*, The eSTREAM Finalists, LNCS 4986, pp. 84-97, Springer-Verlag, 2008. Also available at http://www.ecrypt.eu.org/stream/salsa20pf.html

[36] M. Hell, T. Johansson, A. Maximov, and W. Meier, *The Grain Family of Stream Ciphers*, The eSTREAM Finalists, LNCS 4986, pp. 179-190, Springer-Verlag, 2008, Also available at http://www.ecrypt.eu.org/stream/grainp3.html

[37] S. Babbage and M.Dodd, *The stream cipher Mickey-128*, The eSTREAM Finalists, LNCS 4986, pp. 191-209, Springer-Verlag, 2008, Also available at http://www.ecrypt.eu.org/stream/mickeyp3.html

[38] C. De Cannière and B. Preneel, *Trivium*, The eSTREAM Finalists, LNCS 4986, pp. 244-266, Springer-Verlag, 2008, Also available at http://www.ecrypt.eu.org/stream/triviump3.html

[39] R.L. Rivest, *The RC4 encryption algorithm*, RSA Data Security, Inc., 1992.

[40] H. Wu, *A New Stream Cipher HC-256*, In proc. of Fast Software Encryption (FSE) 2004, LNCS 3017, pp. 226-244, Springer-Verlag, 2004.

[41] C.G. Gunther, *Alternating step generators controlled by de Bruijn sequences*, In proc. of EUROCRYPT '87, LNCS 304, pp. 5-14, Springer-Verlag, 1987.

[42] C. Carlet, *Boolean functions for cryptography and error correcting codes*, in Boolean Methods and Models. Cambridge, U.K.: Cambridge Univ. Press, 2006.

[43] M. Matsui, *Linear Cryptanalysis Method for DES Cipher*, In proc. of EUROCRYPT'93, LNCS 765, pp. 386-397, Springer-Verlag, 1993.

[44] E. Biham and A. Shamir, *Differential Cryptanalysis of the Full 16-Round DES*, In proc. of CRYPTO '92, LNCS 740, Springer-Verlag, 1992.

[45] L.R. Knudsen, *Truncated and Higher Order Differentials*, In proc. of Fast Software Encryption (FSE) 1995, LNCS 1008, pp.196-211, Springer-Verlag, 1995.

[46] E. Biham, A. Biryukov, and A. Shamir, *Cryptanalysis of Skipjack Reduced to 31 Rounds using Impossible Differentials*, In proc. of EUROCRYPT '99, LNCS 1592, pp. 12-23, Springer-Verlag, 1999.

[47] L.R. Knudsen, *Partial and higher order differentials and its application to the DES*, BRICS report series, RS-95-9, ISSN 0909-0878, February 1995.

[48] D. Wagner, *The Boomerang Attack*, In proc. of Fast Software Encryption (FSE) 1999, LNCS 1636, pp. 156-170, Springer-Verlag, 1999.

[49] T. Jakobsen and L.R. Knudsen, *The Interpolation Attack on Block Ciphers*, In proc. of Fast Software Encryption (FSE) 1997, LNCS 1267, pp. 28-40, Springer-Verlag, 1997.

[50] E. Biham, *New Types of Cryptanalytic Attacks Using Related Keys*, In proc. of EUROCRYPT '93, LNCS 765, pp. 398-409, Springer-Verlag, 1994.

[51] A. Biryukov and A. Shamir, *Structural Cryptanalysis of SASAS*, In proc. of EUROCRYPT '01, LNCS 2045, pp. 394-405, Springer-Verlag, 2001.

[52] J. Daemen, L.R. Knudsen, and V. Rijmen, *The Block Cipher SQUARE*, In proc. of Fast Software Encryption (FSE) 1997, LNCS 1267, pp 149-165, Springer-Verlag, 1997.

[53] T. Siegenthaler, *Correlation-Immunity of Nonlinear Combining Functions for Cryptographic Applications*, IEEE Transactions on Information Theory 30 (5), pp. 776-780, 1984.

[54] J. Golic, *Cryptanalysis of alleged A5 stream cipher*, In proc. of EUROCRYPT '97, LNCS 1233, pp. 239-255, Springer-Verlag, 1997.

[55] R. Anderson, M. Bond, J. Clulow, and S. Skorobogatov, *Cryptographic processors - a survey*, In proc. of the IEEE, vol. 94, pp. 357-369, Feb. 2006.

[56] FIPS PUB 140-2 Security Requirements for Cryptographic Modules. Available at http://csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf.

[57] K. Gandolfi, C.Mourte, and F. Olivier, *Electromagnetic Analysis: Concrete Results*, In proc. of CHES 2001, LNCS 2162, pp. 251-261, Springer-Verlag, 2001.

[58] J.J. Quisquater and D. Samyde, *Electromagnetic analysis (EMA): measures and countermeasures for smart cards*, In proc. of E-smart 2001, LNCS 2140, pp. 200-210, Springer-Verlag, 2001.

[59] A. Shamir and E.Tramer, *Acoustic cryptanalysis: on nosy people and noisy machines*, In proc. of EUROCRYPT '04 rump session, May 2004.

[60] J.A. Halderman, S.D. Schoen, N. Heninger, W. Clarkson, W. Paul, J.A. Calandrino, A.J. Feldman, J. Appelbaum, and E.W. Felten, *Lest We Remember: Cold Boot Attacks on Encryption Keys*, In Proc. of 17^{th} USENIX Security Symposium (Sec '08), San Jose, CA, 2008.

[61] A. Kamal and A. M. Youssef, *Applications of SAT Solvers to AES key Recovery from Decayed Key Schedule Images*, In proc. of International Conference on Emerging Security Information, Systems and Technologies (SECURWARE) 2010, pp. 216-220, 2010.

[62] S. Chari, C. Jutla, J. Rao, and P. Rohatgi, *Towards sound approaches to counteract power-analysis attacks*, In proc. of CRYPTO '99, LNCS 1666, pp. 398-412, Springer-Verlag, 1999.

[63] L. Goubin, *A sound method for switching between boolean and arithmetic masking*, In proc. of CHES 2001, LNCS 2162, pp. 3-15, Springer-Verlag, 2001.

[64] B. Debraize and I.M. Corbella, *Fault analysis of the stream cipher Snow 3G*, In proc. of Workshop on Fault Diagnosis and Tolerance in Cryptography 2009, pp. 103-110 , 2009.

[65] M. Hojsik and B. Rudolf, *Floating fault analysis of Trivium*, In proc. of INDOCRYPT 2008, LNCS 5365, pp. 239-250, Springer-Verlag, 2008.

[66] F. Armknecht and W. Meier, *Fault Attacks on Combiners with Memory*, In proc. of Selected Areas in Cryptography (SAC) 2005, LNCS 3897, pp. 36-50, Springer-Verlag, 2005.

[67] E. Biham, L. Granboulan, and P.Q. Nguyen, *Impossible Fault Analysis of RC4 and Differential Fault Analysis of RC4*, In proc. of Fast Software Encryption (FSE) 2005, LNCS 3557, pp. 359-367, Springer-Verlag, 2005.

[68] C. Giraud and A. Thillard, *Piret and Quisquater's DFA on AES Revisited*, 2010. Available at http://eprint.iacr.org/2010/440

[69] G. Piret and J.J. Quisquater, *A Differential Fault Attack Technique against SPN Structures, with Application to the AES and KHAZAD*, In proc. of CHES 2003, LNCS 2779, pp. 77-88, Springer-Verlag, 2003.

[70] C.H. Kim, *Differential Fault Analysis against AES-192 and AES-256 with Minimal Faults*, In proc. of Fault Diagnosis and Tolerance in Cryptography (FDTC) 2010, pp. 3-9, IEEE Computer Society, 2010.

[71] I. Koren and C. Krishna, *Fault-Tolerant Systems*, Morgan Kaufmann publisher, USA, 2007.

[72] G. Bertoni, L. Breveglieri, I. Koren, P. Maistri, and V. Piuri, *Error Analysis and Detection Procedures for a Hardware Implementation of the Advanced Encryption Standard*, IEEE Transactions on Computers, Vol. 52, pp. 492-505, 2003.

[73] B. Yang, K. Wu, and R. Karri, *Scan based side channel attack on dedicated hardware implementations of data encryption standard*, In proc. of the International Test Conference (ITC) 2004, pp. 339-344, IEEE Computer Society, 2004.

[74] M. Agrawal, S. Karmakar, D. Saha, and D. Mukhopadhyay, *Scan Based Side Channel Attacks on Stream Ciphers and Their Counter-Measures*, In proc. of INDOCRYPT 2008, LNCS 5365, pp. 226-238, Springer-Verlag, 2008.

[75] Y. Liu, K. Wu, and R. Karri, *Scan-based attacks on linear feedback shift register based stream ciphers*, ACM Trans. Design Autom. Electr. Syst. 16(2): 20, 2011.

[76] G. Sengar, D. Mukhopadhyay, and D.R. Chowdhury, S*ecured flipped scan-chain model for crypto-architecture*, IEEE Trans. on CAD of Integrated Circuits and Systems 26(11), 2080-2084, 2007.

[77] P. Ekdahl and T. Johansson, *A New Version of the Stream Cipher SNOW*, In proc. of Selected Areas in Cryptography (SAC) 2002, LNCS 2295, pp.47-61, Springer-Verlag, 2002.

[78] R. Anderson, E. Biham, and L.R.Knudsen, *Serpent: A proposal for the advanced encryption standard*, NIST AES Proposal, 1998.

[79] H. Ahmadi, T. Eghlidos, and S. Khazaei, *Improved guess and determine Attack on SOSEMANUK*, 2006. Available at: http://www.ecrypt.eu.org/stream/sosemanukp3.html

[80] Y. Tsunoo, T. Saito, M. Shigeri, T. Suzaki, H. Ahmadi, T. Eghlidos, and S. Khazaei, *Evaluation of* SOSEMANUK *with regard to guess-and-determine attacks*, 2006. Available at http://www.ecrypt.eu.org/stream/sosemanukp3.html

[81] J.-K. Lee, D.-H Lee, and S. Park, *Cryptanalsyis of* SOSEMANUK *and SNOW 2.0 Using Linear Masks*, In proc. of ASIACRYPT 2008, LNCS 5350, pp. 524-538, Springer-Verlag, 2008.

[82] J.Y. Cho and M. Hermelin, *Improved linear cryptanalysis of* SOSEMANUK, In proc. of Information, Security and Cryptology (ICISC) 2009, LNCS 5984, pp.101-117, Springer-Verlag, 2010.

[83] D. Lin and G. Jie, *Guess and Determine Attack on* SOSEMANUK, In proc. of the International Conference on Information Assurance and Security, Vol. 1, pp. 658-661, 2009.

[84] X. Feng, J. Liu, Z. Zhou, C. Wu, and D. Feng, *A Byte-Based Guess and Determine Attack on* SOSEMANUK, In proc. of ASIACRYPT 2010, LNCS 6477, pp. 146-157, Springer-Verlag, 2010.

[85] M.O. Saarinen, *Cryptanalysis of Hummingbird-1*, In proc. of Fast Software Encryption (FSE) 2011, to appear, 2011. Also available at: http://eprint.iacr.org/2010/612.pdf

[86] Specification of the 3GPP Confidentiality and Integrity Algorithms 128-EEA3 & 128-EIA3. Document 4: Design and Evaluation Report, EEA3 EIA3 Design Evaluation v1.1, August, 2010.

[87] Specification of the 3GPP Confidentiality and Integrity Algorithms 128-EEA3 & 128-EIA3. Document 4: Design and Evaluation Report , EEA3 EIA3 Design Evaluation v1.3, January, 2011.

[88] First international workshop on ZUC algorithm, 2010. Available at http://www.dacas.cn/zuc10/

[89] Z. Liu, L. Zhang, J. Jing, and W. Pan, *Efficient Pipelined Stream Cipher ZUC Algorithm in FPGA*, In the first international workshop on ZUC algorithm, 2010. Also available at www.dacas.cn/zuc10/pdf/zuc-08.pdf

[90] R. Nara, N. Togawa, M. Yanagisawa, and T. Ohtsuki, *Scan-based attack against elliptic curve cryptosystems*, In Proc. of the 15th Asia and South Pacific Design Automation Conference (ASP-DAC 2010), pp. 407-412, 2010.

[91] R. Nara, N. Togawa, M. Yanagisawa, and T. Ohtsuki, *A scan-based attack based on discriminators for AES cryptosystems*, IEICE Transactions on Fundamentals of Electronics, vol. E92-A, no.12, pp.3229-3237, 2009.

[92] Yang, b., Wu, K., Karri, R., Secure scan: a design-for-test architecture for crypto chips, In proc. of the 42nd Annual Conference on Design Automation, pp. 135-140, 2005.

[93] National Institute of Standards and Technology, Information Technology Laboratory, *Cryptographic Hash Algorithm Competition*. Available at: http://csrc.nist.gov/groups/ST/hash/sha-3/index.html