Cryptanalysis of Some AES-based Cryptographic Primitives

Riham AlTawy

A Thesis

in

The Concordia Institute for Information

Systems Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy at
Concordia University
Montréal, Québec, Canada

March 2016

CONCORDIA UNIVERSITY
SCHOOL OF GRADUATE STUDIES

This is to certify that the thesis prepared

By: Riham AlTawy

Entitled: Cryptanalysis of Some AES-based Cryptographic Primitives

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

Doctor of Philosophy in Information Systems Engineering

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

Signed by the final examining committee:

Dr. Tarek Zayed — Chair

Dr. Haward M. Heys — External Examiner

Dr. Dongyu Qiu — External to Program

Dr. Mohammad Mannan — Examiner

Dr. Jeremy Clark — Examiner

Dr. Amr M. Youssef — Thesis Supervisor

Approved by

Chair of Department or Graduate Program Director

Dr. Amir Asif

Dean of Faculty

Cryptanalysis of Some AES-based Cryptographic Primitives

Approved by
Dissertation Committee:

_____

_____

_____

_____

_____

Abstract

Cryptanalysis of Some AES-based Cryptographic Primitives

Riham AlTawy, Ph.D.
Concordia University, 2016

Current information security systems rely heavily on symmetric key cryptographic primitives as one of their basic building blocks. In order to boost the efficiency of the security systems, designers of the underlying primitives often tend to avoid the use of provably secure designs. In fact, they adopt ad hoc designs with claimed security assumptions in the hope that they resist known cryptanalytic attacks. Accordingly, the security evaluation of such primitives continually remains an open field. In this thesis, we analyze the security of two cryptographic hash functions and one block cipher. We primarily focus on the recent AES-based designs used in the new Russian Federation cryptographic hashing and encryption suite *GOST* because the majority of our work was carried out during the open research competition run by the Russian standardization body TC26 for the analysis of their new cryptographic hash function Streebog. Although, there exist security proofs for the resistance of AES-based primitives against standard differential and linear attacks, other cryptanalytic techniques such as integral, rebound, and meet-in-the-middle attacks have proven to be effective. The results presented in this thesis can be summarized as follows:

Initially, we analyze various security aspects of the Russian cryptographic hash function GOST R 34.11-2012, also known as Streebog or Stribog. In particular, our work investigates five security aspects of Streebog. Firstly, we present a collision analysis of the compression function and its internal cipher in the form of a series of modified rebound attacks. Secondly, we propose an integral distinguisher for the 7- and 8-round compression function. Thirdly, we investigate the one wayness

of Streebog with respect to two approaches of the meet-in-the-middle attack, where we present a preimage analysis of the compression function and combine the results with a multicollision attack to generate a preimage of the hash function output. Fourthly, we investigate Streebog in the context of malicious hashing and by utilizing a carefully tailored differential path, we present a backdoored version of the hash function where collisions can be generated with practical complexity. Lastly, we propose a fault analysis attack which retrieves the inputs of the compression function and utilize it to recover the secret key when Streebog is used in the keyed simple prefix and secret-IV MACs, HMAC, or NMAC. All the presented results are on reduced round variants of the function except for our analysis of the malicious version of Streebog and our fault analysis attack where both attacks cover the full round hash function.

Next, we examine the preimage resistance of the AES-based Maelstrom-0 hash function which is designed to be a lightweight alternative to the ISO standardized hash function Whirlpool. One of the distinguishing features of the Maelstrom-0 design is the proposal of a new chaining construction called 3CM which is based on the 3C/3C+ family. In our analysis, we employ a 4-stage approach that uses a modified technique to defeat the 3CM chaining construction and generates preimages of the 6-round reduced Maelstrom-0 hash function.

Finally, we provide a key recovery attack on the new Russian encryption standard GOST R 34.12-2015, also known as Kuznyechik. Although Kuznyechik adopts an AES-based design, it exhibits a faster diffusion rate as it employs an optimal diffusion transformation. In our analysis, we propose a meet-in-the-middle attack using the idea of efficient differential enumeration where we construct a three round distinguisher and consequently are able to recover 16-bytes of the master key of the reduced 5-round cipher. We also present partial sequence matching, by which we generate, store, and match parts of the compared parameters while maintaining negligible probability of matching error, thus the overall online time complexity of the attack is reduced.

Acknowledgments

I wish to address my first thanks to my supervisor, Professor Amr Youssef, whose knowledge, kindness, patience, and availability have largely contributed to the existence of this work. Our frequent fruitful discussions have helped improve my confidence in my cryptanalytic abilities and allowed me to better focus on my research goals.

Next, I want to thank Aleksandar Kircanski for jumpstarting my interest in hash function cryptanalysis, and for tolerating my silly questions regarding second-order collisions. I also like to express my gratitude to my lab colleagues at the CIISE Crypto Lab for their friendship and social support.

Finally, thank you Karim for enduring long days and nights full of cryptanalysis and writing papers, and thank you Mama for always pushing me and encouraging me throughout all my studies.

RIHAM ALTAWY

To my family for their love and support

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 General Overview and Motivation

Ever since ancient times, cryptography has been used to protect the privacy of governmental and military communications. With the development of telecommunications during the twentieth century, cryptography has become more important as it evolved to ensure not only information confidentiality, but also its integrity and authenticity. A cryptosystem is usually described by an algorithm that states the series of operations to be performed on the message. Historically, such operations were applied on messages by hand or with the aid of mechanical machines such as the German Enigma [110]. Also, the concept of security by obscurity where the whole cryptographic algorithm is kept a secret was widely adopted. Currently, computers are integral components of the modern networked IT society. Cryptographic primitives are implemented in small processors used by almost every digital object such as mobile phones and credit cards. Accordingly, hiding the workings of the adopted cryptographic algorithms cannot scale with such wide deployment of the utilized security systems. In fact, modern cryptography follows the principle of Auguste Kerckhoffs [81] which states that cryptographic systems must use openly described algorithms while hiding only little information. Such secret information is referred to as a key that is known to the intended parties involved in the communication. Thus, if such key is compromised, setting a new key is enough to regain the system security without the need to change the whole cryptosystem. Additionally and more importantly, it enables the continually ongoing public analysis of the cryptosystem which strengthens the confidence of the cryptographic

community in its security.

Academic research in cryptography revolves around two high level research directions which are the proposal of new cryptosystems and the cryptanalysis of such systems. Designers of modern cryptosystems often opt for non provably secure designs as they are more efficient in terms of their running time, hardware area, and power consumption. Alternatively, the designers of such systems specify the assumed expected effort and resources required to violate the security requirements of their cryptosystems. Such effort and resources rely on the cryptosystem parameters which are chosen such that the effort and resources required by the respective generic attacks are infeasible to be realized practically. On the other hand, cryptanalysts study the workings of such systems and try to devise new approaches to attack their security properties. An attack on a given cryptosystem is considered successful when any of its security properties can be violated with an effort and resources less than those specified by its designers [110].

Symmetric-key cryptosystems are cryptographic primitives that employ one cryptographic key for both the encryption and decryption procedures [110]. Such primitives include block and stream ciphers, Message Authentication Code (MAC) schemes, and dedicated authenticated encryption algorithms. Hash functions are publicly computable deterministic unkeyed functions. However, they are commonly included in the category of symmetric-key primitives due to the fact that they are often built on a block cipher core and cryptanalyzed using block cipher approaches. One of the most prominent block ciphers is the Advanced Encryption Standard (AES) [45] as it is the U.S. standardized encryption algorithm and is widely deployed in security protocols. The AES design offers a heuristic proof for the lower bound complexities against standard differential and linear attacks [28, 100]. Additionally, ever since its inception, no attacks have been discovered that compromise its security in practice. For that reason, many proposals of other symmetric-key primitives such as hash functions and authenticated encryption schemes are adopting AES-based designs.

Generally, symmetric-key encryption algorithms that are built using rigorous security proofs may not be suitable in terms of running time for the current demands and constraints of modern security systems. For that reason, most of the practically used and standardized cryptographic primitives gain

their strength from withstanding years of cryptanalysis without being broken. In other words, such algorithms do not provide a proof of absolute security, but because they have been studied for several years by cryptanalysts, the confidence in their resistance to various attacks is strengthened. Such demand for analysis is usually motivated by the significance of the cryptographic primitive and/or its participation in public research competitions. During such competitions, the standardization body calls for submissions for a cryptographic primitive or analysis results on a given primitive. In response to such call, designers from the cryptographic community submit their proposals or research papers that are evaluated by other researchers through multiple analysis rounds until a winner is chosen. Examples of such competitions include the U.S. NIST's block cipher standard AES (1997-2000) [117] and hash function standard Secure Hash Algorithm-3 (SHA-3) (2007-2012) [118] competitions, and the Russian TC26 open research competition for the analysis of the standard hash function Streebog (2013-2015) [138]. Additionally, a new Competition for Authenticated Encryption: Security, Applicability, and Robustness (CAESAR) [41] funded by NIST was initiated in 2013 to select a portfolio of authenticated encryption algorithms, where the winning schemes are expected to be announced in 2017. These competitions have lead to a flurry in both the design and cryptanalysis of cryptographic algorithms. Particularly, the Russian open research competition on Streebog has motivated the majority of our work which aims to propose cryptanalytic attacks on the Russian cryptographic hashing standard to provide better lower bounds on its security margins.

Cryptanalysis starting from the mid-2000s has improved significantly, especially in the area related to hash functions. More precisely, it has been shown that the commonly used hash functions by that time offered only a very limited security margin. In particular, a completely new chapter in the history of the analysis of Merkle-Damgård (MD) construction that adopt Add-Rotate-Xor (ARX)-based designs was opened when Wang *et al*. [143, 144] managed to enhance multi-block differential cryptanalysis to a point that finding collisions for MD5 became utterly feasible (under a minute on a PC) and a vast reduction of the security margin was obtained for Secure Hash Algorithm 1 (SHA-1). Although Wang's random looking collisions by themselves do not pose any harm, later it was shown that inserting these blocks in certain places of popular documents format (such as PostScript or MS word) can lead to two different files that hash to the same value. Another elegant yet severe consequence

of this attack was demonstrated in [137] through the construction of two different X.509 certificates with two different public keys and the same MD5 hash. Such attack allows impersonation and defeats directly the authentication mechanism of security protocols using MD5. In response to these dramatic cryptanalytic attacks, the National Institute for Standards and Technology (NIST) decided to develop another hash function through a public competition [118]. The SHA-3 competition ended in October 2012 with Keccak [69] being announced as the new U.S. cryptographic hashing standard [40]. During this competition, there has been a conceptual shift in hash function designs through the proposal of several AES-based hash functions. Indeed, at the same time when most of the standardized ARX-based hash functions were failing to resist the techniques introduced by Wang *et al.*, the already existing ISO standard AES-based Whirlpool [126] was not affected by these attacks. The inclination towards AES-based hash function designs was clearly evident among the SHA-3 competition proposals (e.g., the SHA-3 finalists Grøstl [60], JH [149], and LANE [73]). Additionally, Streebog [102], the new Russian hash standard which is officially known as GOST R 34.11-2012, is also among the recently proposed AES-based hash functions. Streebog has been proposed by a group of Russian designers [102], and chosen by the Russian standardization body TC26 to be the new cryptographic hashing encryption standard. In order to boost the confidence of the cryptographic community in Streebog, TC26 has called for an international open research competition dedicated to the analysis of its new standard. The work presented in chapters 3, 4, 5, 6, and 7 of this thesis has been chosen by TC26 as a winner of the open research competition dedicated for the analysis of Streebog.

AES adopts the wide trail design strategy [45] which is a design approach that mitigates standard differential and linear attacks. This strategy provides upper bounds for the probability of any differential or linear trail. More precisely, the wide trail strategy ensures that full difference diffusion after two rounds such that no sparse differential path can be constructed. In the case of AES, the minimum number of active Sboxes of any 4-round path is 25. All the attacks on the AES block cipher in the secret key setting have time, memory, or data complexities that makes such attacks far from being realized practically, and consequently, they do not directly threaten the security of AES. Such attacks include boomerang related-key attack [30], biclique attacks [32], and meet-in-the-middle attacks [48, 49, 52, 96]. Accordingly, the new Russian encryption standard Kuznyechik [3] is designed

based on the wide trail strategy. However, unlike AES, Kuznyechik employs a full state optimum diffusion transformation which ensures full diffusion after one encryption round and results in a minimum of 17 active Sboxes in any 2-round path.

Today, almost every electronic transaction with security requirements relies on the underlying encryption and cryptographic hashing primitives. The research scope of this thesis lies in the analysis of standardized and/or significant AES-based cryptographic primitives, which is important work since compromising their security results in the direct compromise of the security of the whole system employing them. Finding and exhibiting weaknesses of such cryptographic primitives allows fixing them which foils adversaries and prevents any attempt to compromise systems of critical importance such as the IT infrastructure of government organizations and banks which are probably going to adopt such standardized primitives.

The cryptanalytic results presented in this thesis are motivated by the need to devise cryptanalytic methods to analyze the security of AES-based cryptographic primitives in order to provide better lower bounds of their security margins. The main goal of this thesis is to enhance the state of knowledge of the cryptographic and security communities regarding the real security of the analyzed primitives.

1.2 Thesis contributions

In this thesis, we investigate five security properties of the Russian cryptographic hash function Streebog. We also study the resistance of the Maelstrom-0 hash function to preimage attacks. Finally, we propose a key recovery attack on the Russian encryption standard Kuznyechik. The contributions of this thesis are as follows:

- We analyze the collision resistance of the Streebog compression function and its internal cipher with respect to rebound attacks and present practical collision examples on reduced round version of the function to verify our results.

- We study the structural integral properties of reduced-round versions of the Streebog compression function and its internal permutation, where we present 7 and 8-round distinguishers for the

compression function.

- We investigate the security of Streebog and its compression function, assessing their resistance to the meet-in-the-middle preimage attacks, and present a pseudo preimage attack on the compression function and use it to produce hash function preimages.

- We propose a malicious version of Streebog where we exploit the randomness of the independent round constants to provide a backdoored version of the hash function where collisions can be feasibly generated. Our proposed attack has a practical complexity and is verified by example.

- We present a differential fault analysis attack on Streebog, where we consider the function when used in the secret key setting. We propose a two-stage attack to recover the secret inputs of the function when it is used in various MAC schemes.

- We investigate the security of Maelstrom-0 and its compression function, assessing their resistance to the meet-in-the-middle preimage attacks, and propose a four stage approach which combines a 2-block multicollision attack [56, 57] with a meet-in-the-middle attack to bypass the effect of its finalization step and generate preimages of the reduced Maelstrom-0 hash function.

- We propose a key recovery meet-in-the-middle attack on the Russian encryption standard, Kuznyechik, using efficient differential enumeration [49]. We also present partial sequence matching which enables us to lower the overall time complexity of the attack and reduce its memory requirements.

The above contributions have been published in [7, 11, 12, 14–17]. Other works conducted during the tenure of this Ph.D. have been published in [4, 6, 8–10, 13, 84].

Chapter 2

Background

In this chapter, we provide a brief overview of the two basic classes of cryptosystems. We also present a high level literature survey on the different design approaches for symmetric-key encryption algorithms and hash functions, and the well known cryptanalytic methods used for their analysis.

2.1 The Role of Cryptography

A cryptosystem is a system that provides essential security properties required by communication entities. For example, as depicted in Figure 2.1, if two users, Alice and Bob are exchanging messages remotely, the assurance that the exchanged messages are not disclosed, modified, or fabricated is not guaranteed. Moreover, if they are communicating wirelessly, it becomes utterly easy for adversaries to either intercept and read their messages passively, or actively engage in the communication by modifying, deleting, or inserting messages.

Cryptography provides a framework of various cryptographic algorithms to ensure essential security requirements for the safe communication between Alice and Bob. Such requirements include [110]:

- Confidentiality: Keeping information secret from all but those who are authorized to see it.

- Integrity: Ensuring information has not been altered by unauthorized or unknown means.

- Entity authentication: Corroboration of the identity of an entity.

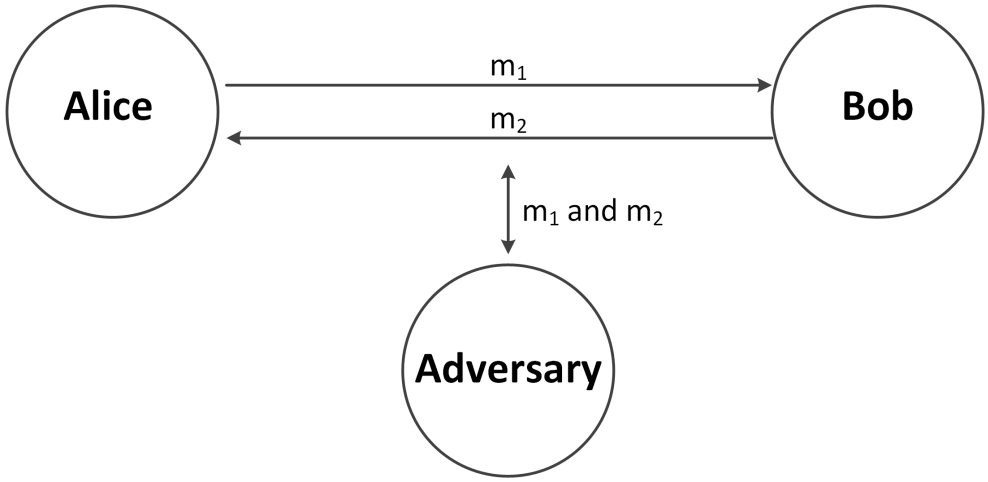

Figure 2.1: An insecure unencrypted communication channel

- Non repudiation: Preventing the denial of previous commitments or actions.

From a high level perspective, cryptosystems are divided into two different design classes: symmetric-key and asymmetric-key primitives. They both differ in the roles of both Alice and Bob, employed operations, rate of data throughput, and key management. In what follows, we give a brief overview of these two classes of cryptographic algorithms

2.1.1 Symmetric-key Primitives

Symmetric-key primitives are also known as private key algorithms, and only one key is required in a communication session. Let $Enc(m, K)$ and $Dec(m, K)$ denote encrypting and decrypting message m with key K, respectively. As depicted in Figure 2.2, both Alice and Bob have symmetric roles as they share a secret key K which is used to encrypt and decrypt the exchanged information. As long as the secret key is known only to Alice and Bob, an adversary cannot disclose the contents of their encrypted communication. Symmetric-key primitives often adopt rather simple operations (e.g., xor, shift, and table lookups), thus they are characterized by their high data throughput and suitability to resource constrained environments. However, they suffer from the key distribution problem where it requires an additional mechanism to allow Alice and bob to exchange the private key securely. Cryptosystems that employ symmetric-key cryptography include block ciphers, stream ciphers, Message Authentication Code (MAC) schemes, and authenticated encryption algorithms.

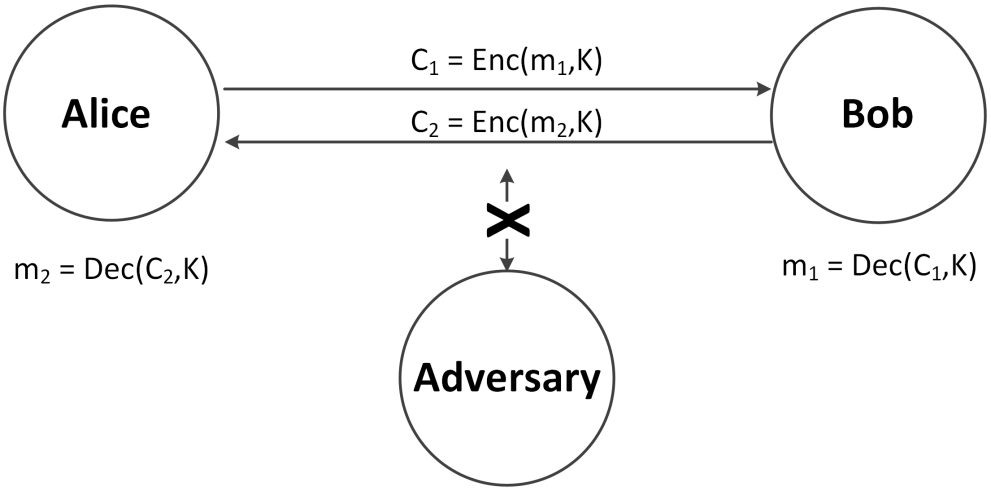

Figure 2.2: Alice and Bob using symmetric-key encryption

2.1.2 Asymmetric-key Primitives

Asymmetric-key primitives are also known as public-key algorithms as the key material includes both private and public information. More precisely, as depicted in Figure 2.3, in a given communication session, each participant has two different keys called a key pair (K_{prv}, K_{pub}): a private and a public key. Bob's public key Kb_{pub} is used by Alice to encrypt messages that only he can decrypt using his private key Kb_{prv}. For that, a given entity's public key can be advertised publicly and used by anyone to initiate secure communication with it, as only the owner of the corresponding private key can disclose the contents of the communication. Because of the fact that the information required for encryption is public, asymmetric-key cryptosystems do not have a problem in managing key distribution. In fact, public-key cryptosystems are often used to enable the distribution of secret session keys in communications protected using symmetric-key encryption. On the other hand, public-key primitives are several orders of magnitude slower than symmetric-key algorithms and their keys are considerably long (e.g., 1024-2048 bit). Cryptosystems that employ asymmetric-key cryptography include digital signatures such as DSS [82], public-key encryption such as RSA [77], and key agreement protocols such as the Diffie-Hellman protocol [125].

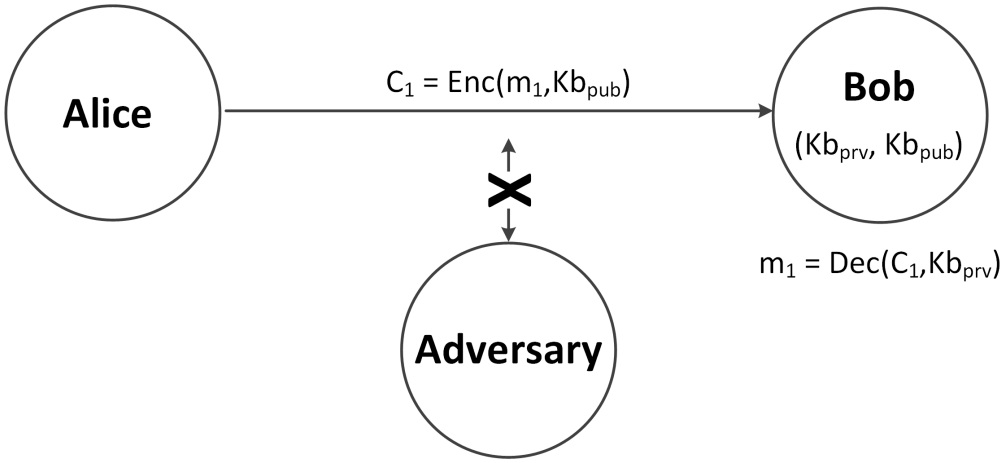

Figure 2.3: Alice and Bob using asymmetric-key encryption

2.2 Block Ciphers

A block cipher is a bijective function that maps a block of n bits of the plaintext into a block of n bits ciphertext parametrized by a secret key of k bits. Let $Enc(p, K)$ and $Dec(c, K)$ denote the encryption and decryption of the n-bit plaintext and ciphertext blocks using a k-bit key, respectively. Formally, a block cipher encompasses the following two mappings [110]:

$$Enc(p, K) : \{0,1\}^n \times \{0,1\}^k \mapsto \{0,1\}^n$$
$$Dec(c, K) : \{0,1\}^n \times \{0,1\}^k \mapsto \{0,1\}^n$$

such that for a given key K, $p = Dec(Enc(p, K), K)$ where the key K is chosen at random from the k-bit key space. A block cipher algorithm defines encryption, decryption, and round subkey generation procedures. Initially, the secret key is processed through a number of mixed linear and nonlinear transformations to generate a set of subkeys to be used in both the encryption and decryption procedures. In the encryption process, the plaintext is used to initialize the encryption state which changes continually by being processed through iterating a nonlinear function for a specific number of rounds [134]. In each round, a given secret subkey is mixed with the current state to progressively obscure its contents. Block ciphers usually employ rather simple operations in their round function which by themselves are insufficient to deliver the required security properties, However, cascading them in a given order in the round function and iterating such function for enough number of rounds makes the algorithm

resistant to known attacks. A round function typically consists of linear and nonlinear transformations. The linear transformations are often implemented using xores, cyclic shifts, and finite field multiplications by constants. On the other hand, nonlinear transformations can be designed using Sboxes, and modular additions.

Modern block ciphers adopt one of two design architectures: Fiestel Network or Substitution Permutation Network (SPN) which are both depicted in Figure 2.4. The former approach which is adopted by the past encryption standard, DES [116] requires more rounds of iterating the mixing round function than the SPN structure to achieve full diffusion of the secret key bits. This is due to the fact that the round function is applied only to half the encryption state and consequently, half the new updated state remains unprocessed. At first the plaintext is loaded in the encryption state (L_0, R_0), for $n/2$-bit blocks L_0 and R_0. After being processed for r rounds the state contains the ciphertext (R_r, L_r). For $1 \leq i \leq r$, round i transforms state (L_{i-1}, R_{i-1}) to state (L_i, R_i) as follows:

$$L_i = R_{i-1},$$
$$R_i = L_{i-1} \oplus f(R_{i-1}, K_i),$$

where K_i denotes the generated subkey for round i and f is the mixing function. On the other hand, as shown in Figure 2.4, in the SPN structure, the round mixing function is applied to the whole state which is initially loaded by the plaintext. Each round applies a key mixing transformation followed by a substitution and permutation/linear transformation. All the employed transformations must be invertible in order to enable decryption of the generated ciphertext. On the other hand, the mixing function f in Fiestel designs may not be invertible because decryption is achieved by running the same r-round encryption procedure on the ciphertext and using the subkeys in the reverse order.

2.2.1 Security Requirements

It is required that block ciphers resist attacks where the adversary can have additional knowledge and capabilities than only observing random ciphertext. This extra knowledge results in the adversary mounting known-plaintext, chosen-plaintext, chosen-ciphertext, and adaptive attacks [110].

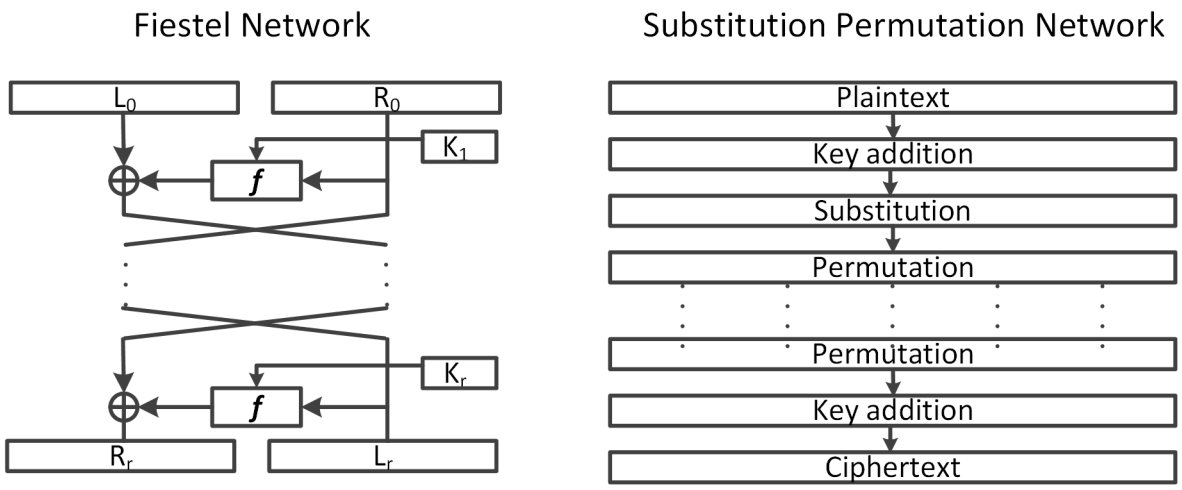

Figure 2.4: Fiestel and Substitution Permutation Network constructions.

Also, recently, there has been an inclination from the cryptographic community to consider the resistance of block cipher to additional attack scenarios. For example, current block ciphers are analyzed with respect to related-key attacks [30] in which the adversary is assumed to adopt a chosen-plaintext model while assuming that encryption is performed using different unknown keys that have a known/chosen difference. Also, analysis of block ciphers using the known-key and chosen-key [90] models has become popular due to the utilization of block ciphers in constructing hash functions. Particularly, when a block cipher-based compression function is used in a hash function, both inputs to of the internal block ciphers are known. More precisely, according to the adopted mode of operation, either the message or the chaining value is processed through the key generation procedure, and thus the attacker can control the key input. Since all the analyzed primitives in this thesis adopt an AES-based design, in what follows, we give a brief description of the AES block cipher.

2.2.2 The Advanced Encryption Standard

AES is the U.S. standardized block cipher [117] which was originally proposed by Daemen and Rijmen [45]. AES is an SPN cipher that follows the wide trail strategy which ensures that the full diffusion of a difference in one byte is achieved after two execution rounds. The algorithm defines encryption and key schedule procedures. In what follows, we give a brief overview of the specifications of AES.

- Encryption: AES encrypts blocks of 128 bits which are initially loaded in a state of 4×4 bytes. It allows the use of three key sizes: 128, 192, and 256 bits where the encryption procedure updates the state for 10, 12, and 14 rounds, respectively. As depicted in Figure 2.5, during one round, the following four round transformations are used to update the state:

- *SubBytes (SB)*: A nonlinear transformation that substitutes each byte from the state by another byte from the AES Sbox.

- *ShiftRows (SR)*: A linear cyclic shift transformation that rotates the i^{th} row of the state to the left by i places, for $i = 0, 1, 2,$ and 3.

- *MixColumns (MC)*: A linear transformation that left multiplies a constant MDS matrix [45] by the state. Each column of the state is multiplied independently.

- *AddKey (AK)*: A linear transformation which es the 16-byte subkey with the state.

Initially, the plaintext block is loaded in the state and then it is ed with the secret master key before the beginning of the first round. Also, the MixColumns transformation is omitted in the last round.

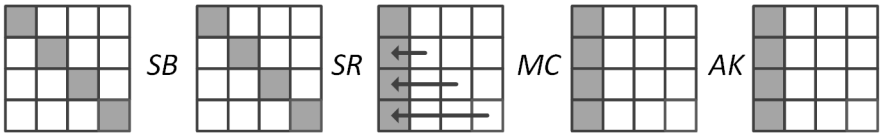

Figure 2.5: The AES round function.

- Subkeys Generation: Round subkeys are 128-bit keys where the i^{th} round subkey K_i is generated from its preceding round subkey K_{i-1}, and the initial key K_0 is evaluated from the master key. One round of the subkey generation procedure consists of a linear part using operations and circular shifts and a nonlinear part using four AES Sbox lookups. More details of the exact working of the AES key schedule can be found in [45].

- Decryption: AES is an SPN construction where all its transformations are invertible. Accordingly, the AES decryption is implemented by applying the inverse round transformations in reverse order on the ciphertext using the round keys that are applied in the reverse order.

2.3 Hash Functions

A hash function is a function that maps arbitrary strings into strings of fixed length. It takes a message as an input and computes a fingerprint for this message (sometimes called the message digest). The message is seen as a sequence of bits of arbitrary length, and the fingerprint is a sequence of bits of a fixed size, for example 256 bits (the output size of the hash function). The hash function compresses the message, and generates a fingerprint that depends on all the bits of the message. Accordingly, it can be used to uniquely identify and guarantee the integrity of this message. In what follows, we define the fundamental properties of hash functions. Moreover, we discuss different applications of hash functions, and describe their basic design principles.

2.3.1 Cryptographic Properties and Applications

A perfect hash function should behave like a random oracle [36]. The only way to get information about the fingerprint of a message is to recalculate it, and the result must be random and exhibit great avalanche effect. The most important properties that are related to the use of a hash function to produce unique identifiers are: preimage resistance, second preimage resistance, and collision resistance [110]. Let H be a hash function with n-bit output, i.e., H is a deterministic function that takes an arbitrary length input, and outputs a binary string of length n. Formally, $H : \{0,1\}^* \mapsto \{0,1\}^n$.

Definition 2.1 *A hash function H is preimage resistant (one-way) if for essentially all pre-specified outputs, it is computationally infeasible to find any input which hashes to that output, i.e., to find any preimage x such that $H(x) = y$ when given any y for which a corresponding input is not known.*

Definition 2.2 *A hash function H is second-preimage resistant (weak collision resistant) if it is computationally infeasible to find any second input which has the same output as any specified input, i.e., given x, to find a 2nd-preimage $y \neq x$ such that $H(x) = H(y)$.*

Definition 2.3 *A hash function H is collision resistant (strong collision resistant) if it is computationally infeasible to find any two distinct inputs x and y which hash to the same output, i.e., $H(x) = H(y)$*

Collision resistance implies second-preimage resistance but not vice versa. In practice, collision resistance is the most difficult to satisfy because generating a random colliding message pair is less restrictive than finding a second preimage or a preimage of a given digest. Consequently, breaking collision resistance is the main objective of most of the attacks on hash functions. However, a good hash function should exhibit other properties than that mentioned above. For example, one would expect that flipping an input bit would lead to approximately half the output bits being flipped (avalanche property) and that one cannot practically guess some input bits when given the output of the hash function (local one-wayness). Inability to provide such properties or other properties, such as the resistance to pseudo-collision [142] (also known as free-start collision) where one can find two different messages and two different chaining values that hash to the same digest, semi free-start collision where one can find two different messages and two equal chaining values which are not equal to the standard IV that hash to the same digest, and second order collision attacks is categorized as a certificational weakness. These weak properties do not imply a break of a hash function but are enough to shed doubt on its design principles.

The hash function is a very versatile cryptographic primitive, and many cryptographic systems are based on it, that is why it has gained a reputation for being the *swiss army knife* of cryptography. Applications that employ hash functions in the core of their operation include:

- Digital signatures *(Hash-and-Sign)*: Signature schemes such as RSA or ElGamal [110] are used to authenticate the message and the signer, but they require complex calculations. Generally, instead of applying a signature scheme directly to a long message, the signature is applied to a hash of the message. Thus, the signing operation is performed on a small identifier and consequently is less expensive.

- Commitment schemes: E-bidding and digital cash protocols [67, 146] demand that participants commit to their decisions. Firstly, a participant reveals the hash of her randomized decision, then later reveals its contents. The hash provides no usable information about the decision, but it ensures that the participant cannot change it after revealing the hash.

- Message Authentication Code (MAC): To authenticate a message M, two participants share a secret key K, and add an identifier $MAC_K(M)$ to the message. An adversary should not be able to

calculate the MAC without knowing the key. A simple way to construct a MAC is to get the message and the key in a hash function (e.g., $HMAC_K(M) = H(K \oplus opad \parallel H(K \oplus ipad \parallel M))$) [25].

2.3.2 Generic Attacks

A generic attack is an attack that is applicable to all hash functions irrespective of their internal structure, as opposed to certain attacks that exploit specific vulnerabilities of a particular design. The complexity of a generic attacks is evaluated by the number of the hash function evaluations needed to mount this attack. The generic complexities of preimage, second preimage, and collision attacks for a hash function H with $n-$bit output are provided below [110].

- Pre-image attack: In this attack, the adversary is given a hash, and she must find a message that produces this hash. The generic attack is the exhaustive search, which has a complexity of 2^n.

- Second preimage attack: The adversary is provided with a message, and she must find another message that has the same hash value. This attack is generally easier than preimage attack, because one can reuse parts of the first message to figure out the second. However, the generic attack is still the exhaustive search, which requires 2^n hash computations.

- Collision attack: In this attack, the adversary has to find two messages with the same hash. A collision attack is easier to launch than preimage and second preimage attacks because the opponent has the choice of two messages. The generic attack is based on the birthday paradox [110], which has a complexity of $2^{n/2}$. Table. 2.1 provides the parameters of the three generic attacks. In what follows, we recall the birthday paradox and its relation to collision search.

The birthday paradox [110] is a mathematical property, which states that the probability of finding two people among a random group of $1.2\sqrt{365} = 23$ people that share the same birthday is about 50 percent [110]. The same rationale applies to finding a pair of messages that collides under a random hash function H. If H is a hash function with n-bit output, then the complexity of finding collisions is $1.2\sqrt{2^n} \approx 2^{n/2}$.

Attack	Input	Output	Property	Complexity
Pre-image	$H(M_1)$	M_2	$H(M_1) = H(M_2)$	2^n
2^{nd} preimage	M_1	M_2	$H(M_1) = H(M_2), M_1 \neq M_2$	2^n
Collision	-	M_1, M_2	$H(M_1) = H(M_2)$	$2^{n/2}$

Table 2.1: Generic attack parameters.

2.3.3 Hash Function Construction

Rather than building directly a function that compresses an arbitrary input size, cryptographic hash functions are constructed from iterating the execution of a *compression function*. A compression function takes a fixed size input and compresses it to a fixed size output. However, a hash function must be able to process arbitrary size input. Consequently, it needs a *domain extender* to extend the domain of the compression function. Thus, given a compression function, the domain extender produces a function with arbitrary-length input. In what follows, we give a brief overview on the Merkle-Damgård domain extender which is the most widely used domain extender.

Merkle-Damgård (MD) Construction

As depicted in Figure 2.6, the easiest way to build a hash function is to iterate over the compression function and use the last calculated output as the hash value. In 1989, Ralph Merkle [111] and Ivan Damgård [46] independently proposed a domain extender algorithm and proved that if the one-way compression function is collision resistant, then the hash function constructed using it is also collision resistant. However, the MD construction is vulnerable to the length extension attack where given an unknown hash function input x and its digest $H(x)$, it is easy to find the value of $H(x||pad(x)||y)$, where y is a message chosen by the attacker and pad is the padding function of the hash function which is used to make the length of the input message a multiple of the message block length. Other domain extenders build on the MD construction and propose a finalization stage to mitigate the length extension attack. Such finalization stage usually applies the compression function on the summation of either the intermediate chaining values [61] or the messages [102].

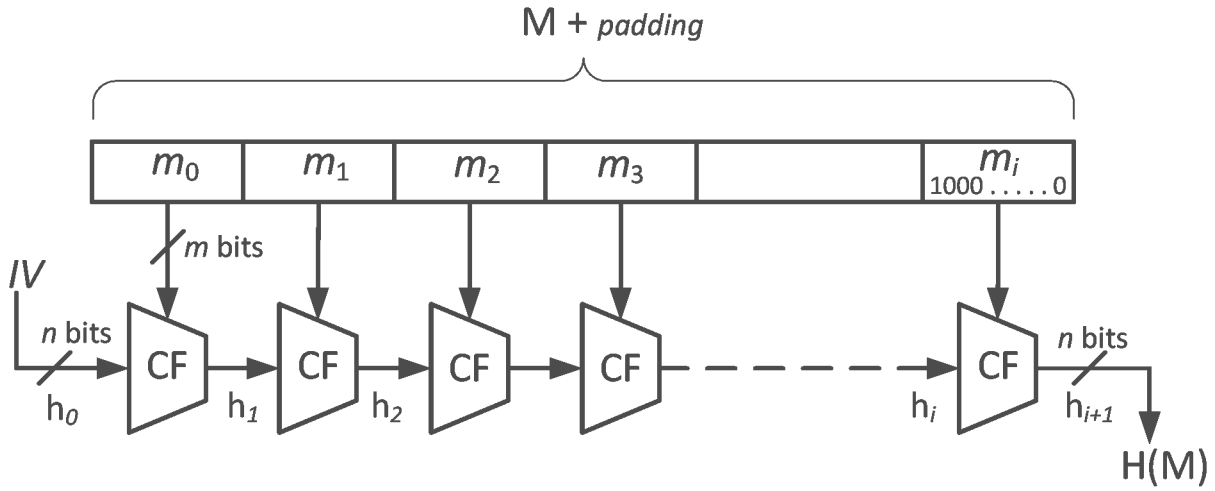

Figure 2.6: Merkle-Damgård construction

2.3.4 Block Cipher-based Compression Functions

The most common method for constructing a compression function is to employ a block cipher core. All the analyzed hash functions in this thesis adopt block cipher-based compression functions. There are many advantages for building a compression function from a block cipher. For instance, block ciphers are usually well studied cryptographic primitives and various efficient implementations for different block ciphers exist. Additionally, if a block cipher is already implemented on a device, a block cipher-based hash function can be added to the system with minimal additional cost. Preneel *et al*. [123] have analyzed the security of different modes of operation that are used for turning a block cipher into a compression function. The three most frequently used modes are Davies-Meyer (DM), Matyas-Meyer-Oseas (MMO) and Miyaguchi-Preneel (MP) which are shown in Figure 2.7.

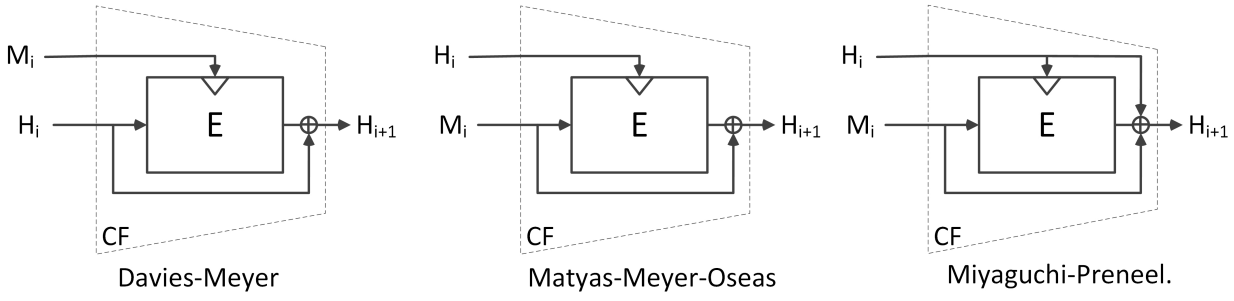

Figure 2.7: Three main block cipher modes to construct compression functions.

2.4 Overview of the Employed Cryptanalytic Methods

A successful cryptanalytic attack violates the security requirements of the cryptographic primitive using effort less than that claimed by its designers. For example a block cipher key recovery attack is expected to retrieve the k-bit master key with a time complexity less than 2^k, which is the time required by the generic brute force attack. A given cryptanalytic attack is characterized by the following complexities:

- *Time complexity*: The amount of required computations to successfully launch the attack. Such amount is usually expressed in terms of the number of executions of the analyzed cryptographic primitive.

- *Data complexity*: The amount of both input and output data required to be available to the attacker in order to successfully implement the attack. In the case of keyed primitives, the data complexity often involves the number of queries to the encryption and/or decryption oracle.

- *Memory complexity*: The amount of storage required to perform the cryptanalytic attack.

In what follows, we give a brief description of the basic cryptanalytic techniques which are employed in developing the attacks on the primitives that are investigated in this thesis.

- Differential cryptanalysis: Differential cryptanalysis [28] is one of the most important methods for the analysis of hash functions and block ciphers. Particularly, most of the currently developed cryptanalytic techniques are based on differential cryptanalysis. Such techniques include the rebound attack [107], the boomerang attack [31, 140], and meet-in-the-middle attacks with differential enumeration [52]. In a differential attack, one tries to follow how the difference between two inputs evolves after it propagates through execution rounds. The input difference propagation through linear transformations is deterministic. However, a given input difference to a nonlinear transformation such as an Sbox results in multiple output differences depending on the value of the input. Accordingly, the cryptanalyst studies the differential properties [28] of the analyzed nonlinear layer and chooses an output difference that occurs with high probability. Then, by adding more rounds, one constructs a differential path whose probability is equal to the product of probabilities of the analyzed rounds. Now given the output difference at the state before the last round and the corresponding ciphertext pairs,

some bits of the last round subkey are guessed to compute the value of the portions of the state where the output difference is given. The cryptanalyst then picks the guessed subkey bits that generated the maximum number of results with a difference equal to the expected one as the right subkey bits. Since one guesses only parts of the last round subkey which is usually smaller than the master key, significant gain over exhaustive key search is achieved. In the case of hash functions, since differences in the execution state can be canceled by introducing differences in the message blocks, the attacker first constructs a differential path that has zero output difference and then searches for a pair of messages to follow it in order to produce a collision [143, 144].

- Truncated differential cryptanalysis: Truncated differential cryptanalysis was proposed by Knudsen [89], and it proved to be effective against AES-based hash functions. A truncated trail is a differential path that does not specify the value of the differences in the path, but rather it indicates whether a given bit/byte is active or not, i.e., the differences are not fully specified. Truncated differential cryptanalysis was first applied by Knudsen on the round reduced DES [89] and Skipjack [91] block ciphers. AES-based hash functions have been extensively analyzed using truncated differentials during the SHA3 competition. Specifically, the rebound attack [107] and its improvements [95, 104, 121] rely on truncated trails in describing the difference propagation. The rebound attack has been used to analyze nearly all the AES-based SHA3 submissions including LANE [101], JH [128], Echo [75, 121], and Grøstl [107, 109, 121].

- Integral attacks: Integral cryptanalysis was proposed by Knudsen and Wagner in [88]. It is considered as a dual to differential cryptanalysis and is efficient against ciphers that are resistant to differential attacks. While in differential cryptanalysis, one considers the propagation of differences between pairs of values to obtain probable differentials, in integral cryptanalysis, we consider the propagation of sums of many values to obtain integrals. Integral cryptanalysis is specifically designed for block ciphers which use only bijective transformations. An integral covers several rounds of the cipher and describes how the summation properties of a set of input values would be affected by each successive round.

Before being formalized in [88], the idea of integral attacks has been explored under several names. It was first discovered during the analysis of the square cipher [43] and named the square attack. Following this, the attack was generalized into the saturation attack and was used to analyze

the Twofish cipher [98]. Ever since higher order integrals have been introduced in [88], integral cryptanalysis has been used to analyze block ciphers in the known key setting [87, 113, 131] and to present distinguishers for the components of hash functions.

- Meet-in-the-Middle Attacks: Meet in the middle (MitM) attacks have drawn a lot of attention since the inception of the original attack which was proposed in 1977 by Diffie and Hellman [50] for the analysis of the Data Encryption Standard (DES). Soon after, the attack became a generic approach to be used for the analysis of ciphers with non complicated key schedules. For this class of ciphers, one can separate the execution into two independent parts where each part can be computed without guessing all the bits of the master key. The first execution part covers encryption rounds from the plaintext to some intermediate state and the other part covers decryption rounds from the corresponding ciphertext to the same internal state. At this point, the attacker has knowledge of the same intermediate state from two independent executions where the right key guess produces matching states. A typical MitM attack can be launched with as low as one known plaintext-ciphertext pair. Accordingly, with the recent growing interest in low data complexity attacks [35], MitM attacks have witnessed various improvements and have been widely adopted for the analysis of various cryptographic primitives. The increasing motivation for adopting low data complexity attacks for the analysis of ciphers is backed by the fact that security bounds are better perceived in a realistic model. Particularly, in a real life scenario, security protocols impose restrictions on the amount of plaintext-ciphertext pairs that can be eavesdropped and/or the number of queries permitted under the same key. MitM attacks have been applied on block ciphers such as AES [47, 48, 52] and LBlock [10]. In the context of hash functions, MitM attacks are used to generate preimages [132, 150]

- Fault Analysis Attacks: Fault analysis is an implementation dependent attack where the attacker applies some kind of physical intervention during the computation of the internal state of the primitive to corrupt random or known bits in the state. Consequently, the attacker observes the correct and faulty outputs and performs some analysis to gain non negligible information about the secret information embedded in the hardware. Fault injection can be done in many ways which include power glitches, clock pulses, and laser radiation [42, 136].

Fault analysis was first introduced when Boneh *et al.* showed how the private key of the RSA-CRT-algorithm can be successfully recovered by observing the correct signature and then injecting

a fault and acquiring the faulty signature [34]. Afterwards, Biham and Shamir extended the idea to cover symmetric-key primitives where they introduced differential fault analysis (DFA) [29]. DFA combines fault analysis with differential cryptanalysis where the difference between faulty and genuine ciphertexts is exploited. DFA attacks have been widely used for recovering the secret inputs of block ciphers and keyed hash functions [55, 64, 85, 139].

Chapter 3

Collision Analysis of Streebog

In this chapter, we investigate the collision resistance of the Russian cryptographic hashing standard GOST R 34.11-2012, also known as Streebog, where we analyze its compression function and internal cipher with respect to rebound attacks. First, we analyze the differential properties of the Streebog Sbox differential distribution table and show how these properties affect the complexity of the rebound attack. As for the internal cipher, we introduce differences in both the key schedule and message encryption, and propose a new message differential path such that a local collision is enforced every two rounds. Accordingly, the Sbox matching complexity caused by its differential bias is bypassed. As a result, we efficiently produce free-start 5-round collision and 7-round near collision examples for the internal cipher. Moreover, we show that the compression function is vulnerable to semi free-start 7.75 round collision, 8.75 and 9.75 round near collision attacks and present an example for a 4.75 round 50-byte near colliding message pair. Our results are summarized in Table 3.1 .

Target	#Rounds	Time	Memory	Attack
Internal cipher	5	2^8	2^8	Free-start collision
	8	2^{64}	2^8	
Compression function	7.75	2^{184}	2^8	Semi free-start collision
	4.75	2^8	-	
	7.75	2^{72}	2^8	Semi free-start near collision (50 bytes)
	8.75	2^{128}	2^8	
	9.75	2^{184}	2^8	

Table 3.1: Summary of our collision analysis of Streebog.

3.1 Introduction

The attacks by Wang *et al.* on MD5 [144] and SHA-1 [143] followed by the SHA-3 competition [118] have led to a flurry in the area of hash function cryptanalysis. The primary targets of these attacks are the Add-Rotate-Xor (ARX) based hash functions where one can find differential patterns that propagate with acceptable probabilities. Additionally, using message modification techniques, significant complexity reduction is achieved. Consequently, during the SHA-3 competition, different design concepts were introduced, out of which are the Advanced Encryption Standard (AES) based designs that are known for their resistance to standard differential attacks due to the wide trail strategy. The ISO standard Whirlpool [126], the SHA-3 finalist Grøstl [60], and the new Russian hash standard Streebog [2] are among the proposed AES-based hash functions.

Streebog was proposed in 2010 [102]. It has an output length of 512/256-bit. The compression function employs a 12-round AES-like cipher with 8×8-byte internal state preceded with one round of nonlinear whitening of the chaining value. The compression function operates in Miyaguchi-Preneel (MP) mode and is plugged in Merkle-Damgård domain extender with a finalization step [2]. Streebog officially replaces the previous standard GOST R 34.11-94 which has been theoretically broken in [105, 106] and recently analyzed in [103].

Due to the significance of this standard, its security has been thoroughly investigated in a series of works appearing in a relatively short time. These works include the analysis of the collision resistance of its compression function and internal cipher by AlTawy *et al.* [7] and Wang *et al.* [145]. An integral analysis of the compression function has been presented by AlTawy and Youssef where integral distinguishers for the reduced compression function was proposed [11]. Moreover, preimage attacks on the reduced hash function have been independently proposed by Altawy and Youssef [12] and Zou *et al.* [152], and later the attacks were improved by Bingka *et al.* [99]. Also, Kazymyrov and Kazymyrova presented an analysis of the algebraic aspects of the function [79], and a long second preimage attack was proposed by Guo *et al.* [65]. Furthermore, a malicious version of the whole hash function where practical collisions are generated was presented in [17]. Finally, the function was investigated in the secret-key setting and a differential fault analysis attack has been proposed in [14] to recover the secret key when Streebog is used in various MAC schemes.

The rebound attack is a differential attack [107] proposed by Mendel *et al.* during the SHA-3 competition to construct differential paths for AES-based hash functions. Previous literature related to the rebound attack includes Mendel *et al.* first proposal on the ISO standard Whirlpool and the SHA-3 finalist Grøstl [107, 108]. In particular, Mendel *et al.* presented a 4.5-round collision, 5.5-round semi free-start collision and 7.5-round near collision attacks on the Whirlpool compression function. As for Grøstl-256, a 6-round semi free-start collision is given. Subsequently, rebound attacks have been applied to other AES-based hash functions such as LANE [101], JH [128], and Echo [75]. Various tweaks have been applied to the basic rebound attack in order to construct differential paths that cover more rounds such as merging multiple in-bounds [95], super Sbox cryptanalysis [63], extended 5-round inbound [95], and linearized match-in-the-middle and start-from-the-middle techniques [104]. Lastly, Sasaki *et al.* [132] presented a free-start collision and near collision attacks on Whirlpool by inserting difference in the intermediate keys to cancel the difference propagation in the message and thus creating local collisions every 4 rounds. Previous work findings were often reported on reduced rounds of the compression function, internal block cipher and/or its internal permutations [104, 106].

In the first part of this chapter, we investigate the security of the Streebog hash function, assessing its resistance to rebound attacks. We efficiently produce free-start collision and near collision for the internal cipher (E) reduced to 5 and 7.75 rounds by employing the concept of local collisions. Specifically, we present a message differential path such that a local collision is enforced every 2 rounds. Thus we bypass the complexity of the rebound matching in the message in-bounds by using the same differentials as in the key path. Consequently, in contrast to [132], finding one key satisfying the key path is practically sufficient for finding a message pair following the message path.

In the second part of this chapter, we present a practical 4.75 round 50 (out of 64) bytes near colliding message pair for the compression function and show that it is vulnerable to semi free-start 7.75 round collision, 8.75 and 9.75 round near collision attacks. Examples for the internal cipher attacks and the 4.75 round compression function near-collision attack are provided to validate our results.

The rest of the chapter is organized as follows. In the next section, the specification of the Streebog hash function along with the notation used throughout the chapter are provided. A brief

overview of the rebound attack is given in Section 3.3. Afterwards, in Sections 3.4 and 3.5, we provide detailed description of our attacks, differential patterns, and the complexities of the attacks. Finally, the chapter is concluded in Section 3.6.

3.2 Specification of Streebog

Streebog outputs a 512 or 256-bit hash value, where half the last state is truncated when adopting the 256-bit output. The standard specifies two different *IVs* to be used with the two output lengths. The function can process messages of length up to $2^{512} - 1$. The compression function iterates over 12 rounds of an AES-like cipher with an 8×8 byte internal state and a final round of key mixing. The compression function operates in Miyaguchi-Preneel mode and is plugged in Merkle-Damgård domain extender with a finalization step. The input message M is padded into a multiple of 512 bits by appending one followed by zeros. The message length for MD-strengthening is further included as an extra separate block, followed by a block of a checksum evaluated by the modulo 2^{512} addition of all message blocks as a finalization step. More precisely, let $n = \lfloor \frac{|M|}{512} \rfloor$ and the input message $M = x\|m_n\|..\|m_1\|m_0$, where $|M|$ is length of M, and x is an un-complete or an empty block. The message is padded as follows: let $m_{n+1} = 0^{511-|x|}\|1\|x$, then the padded message $M = m_{n+1}\|m_n\|..\|m_1\|m_0$. Let $\sigma = m_{n+1} + .. + m_1 + m_0$. The compression function g_N is

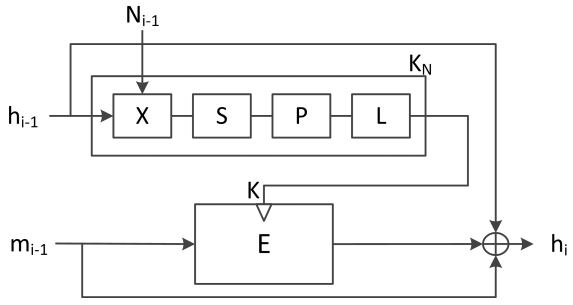

Figure 3.1: The Streebog compression function g_N

parameterized by N which is the counter for the bits that are hashed so far, where $N = 0$ when the compression function is used in the finalization stage and thus it is denoted by g_0. The compression function g_N is fed with three inputs: the chaining value h_{i-1}, a message block m_{i-1}, and the counter of bits hashed so far $N_{i-1} = 512 \times i$. (see Figure 3.1). Let h_i be a 512-bit chaining variable. The

first state is loaded with the initial value *IV* and assigned to h_0. The hash value of M is computed as follows:

$$h_i \leftarrow g_N(h_{i-1}, m_{i-1}, N_{i-1}) \text{ for } i = 1, 2, .., n+2$$
$$h_{n+3} \leftarrow g_0(h_{n+2}, |M|)$$
$$h(M) \leftarrow g_0(h_{n+3}, \sigma),$$

where $h(M)$ is the hash value of M. As depicted in Figure 3.1, the compression function g_N consists of:

- K_N: a nonlinear whitening round of the chaining value. It takes a 512-bit chaining variable h_{i-1} and a counter of the bits hashed so far N_{i-1} and outputs a 512-bit key K.
- E: an AES-based cipher that iterates over the message for 12 rounds in addition to a finalization key mixing round. The cipher E takes a 512-bit key K and a 512-bit message block m as a plaintext. As shown in Figure 3.2, it consists of two similar parallel flows for the state update and the key scheduling.

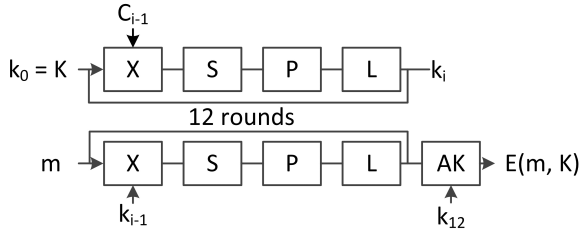

Figure 3.2: The internal block cipher (E)

Both K_N and E operate on an 8×8 byte key state K. E updates an additional 8×8 byte message state M. In one round, a given state is updated by the following sequence of transformations:

- AddKey(X): XOR with either a round key, a constant, or the counter of bits hashed so far (N).
- SubBytes (S): A nonlinear byte bijective mapping.
- Transposition (P): Byte permutation.
- Linear Transformation (L): Row multiplication by an MDS matrix in GF(2).

Initially, state K is loaded with the chaining value h_{i-1} and updated by K_N as follows:

$$k_0 = L \circ P \circ S \circ X[N_{i-1}](K).$$

Now K contains the key k_0 to be used by the cipher E. The message state M is initially loaded with the message block m and $E(k_0, m)$ runs the key scheduling function on state K to generate 12 round keys $k_1, k_2, .., k_{12}$ as follows:

$$k_i = L \circ P \circ S \circ X[C_{i-1}](k_{i-1}), \text{ for } i = 1, 2, .., 12,$$

where C_{i-1} is the i^{th} round constant. The state M is updated as follows:

$$M_i = L \circ P \circ S \circ X[k_{i-1}](M_{i-1}), \text{ for } i = 1, 2, ..., 12.$$

The final round output is given by $E(k_0, m) = M_{12} \oplus k_{12}$. The output of g_N in the Miyaguchi-Preneel mode is $E(K_N(h_{i-1}, N_{i-1}), m_{i-1}) \oplus m_{i-1} \oplus h_{i-1}$ as shown in Figure 1. For further details, the reader is referred to [2].

- Notation: Let M and K be (8×8)-byte states denoting the message and key state, respectively. The following notation will be used throughout the chapter:

- M_i: The message state at the beginning of round i.
- M_i^U: The message state after the U transformation at round i, where $U \in \{X, S, P, L\}$.
- $M_i[r, c]$: A byte at row r and column c of state M_i.
- $M_i[\text{row } r]$: Eight bytes located at row r of M_i state.
- $M_i[\text{col } c]$: Eight bytes located at column c of M_i state.
- $m \xrightarrow{r_i} n$: A transition from an m active bytes state at round i to an n active bytes state at round $i + 1$.

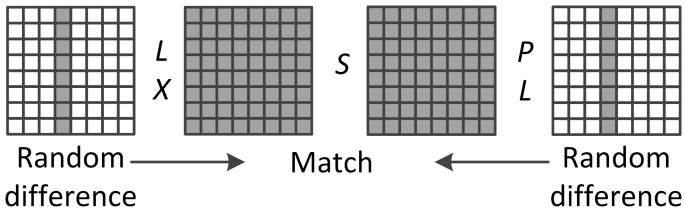

Figure 3.3: The inbound phase of the rebound attack.

- $m \xleftarrow{r_i} n$: A transition from an n active bytes state at round $i+1$ to an m active bytes state at round i.

Same notation applies to K.

3.3 The Rebound Attack

The rebound attack [107] is proposed by Mendel *et al.* for the cryptanalysis of AES-based hash functions. It is a differential attack that follows the inside-out or start from the middle approach which is used in the boomerang attack [140]. The rebound attack is composed of three phases, one inbound and two outbounds. The compression function, internal block cipher or permutation of a hash function is divided into three parts. If C is a block cipher, then C is expressed as $C = C_{fw} \circ C_{in} \circ C_{bw}$. The middle part is the inbound phase and the forward and backward parts are the two outbound phases. In the inbound phase, a low probability XOR differential path is used and all possible degrees of freedom are used to satisfy the inbound path. In the two outbound phases, high probability truncated paths [89] are used. In other words, one starts from the middle satisfying C_{in}, then hash forward and backward to satisfy C_{fw} and C_{bw} probabilistically. For an 8×8 byte state, the basic rebound attack finds two states satisfying an inbound phase over two rounds $8 \xrightarrow{r_i} 64 \xrightarrow{r_{i+1}} 8$. The main idea is to pick random differences at each of the two eight active bytes sates. Then propagate both backward and forward until the output and input of the full active state Sbox, respectively. Using the Sbox differential distribution table (DDT), find values that satisfy input and output differentials. This process is further illustrated in Figure 3.3. The last step of the attack is called the Sbox matching phase and its complexity depends on the Sbox DDT. If the probability of differentials that have solutions is p, then the matching probability is given by p^8. In the following, we analyze the Sbox used in Streebog and investigate how it affects

the complexity of the rebound attack. The Streebog Sbox DDT has the following properties:

- Out of the 65536 differentials, there are 27300 possible non trivial differentials, i.e., nonzero (input, output) difference pairs that have solutions. Thus the probability that a randomly chosen differential is possible $\approx 0.42 = 2^{-1.3}$
- Each possible differential can have 2, 4, 6, or 8 solutions.
- A given input difference has a minimum of 98 and a maximum of 114 output differences.
- A given output difference has a minimum of 90 and a maximum of 128 input differences.
- For a given input (output) difference the average number of output (input) difference is 107.

From the analysis of the Sbox DDT, one can estimate the complexity of the inbound matching part of the rebound attack. Let us consider the basic inbound path $8 \xrightarrow{r_1} 64 \xrightarrow{r_2} 8$. One can find a pair of states satisfying this path as follows:

1. Compute the Sbox DDT.
2. Choose a random 8 differences for M_2^L active bytes.
3. Propagate the differences in M_2^L backwards until M_2^S (output difference).
4. for each row in M_1^P
    a. Choose a random difference for one active byte, propagate it forward to M_2^X (input difference). Propagating one active byte in M_1^P through the L transformation results in full active row in M_2^X.
    b. Using the Sbox DDT, determine if the corresponding row differences in M_2^X and M_2^S have solutions. If one byte differential pair is not possible, go to step 4.a.

One can repeat step (4.a) at most 2^8 times since we variate only one byte. However, the success probability of step 4.b. (finding solutions for the whole active row) is $2^{-1.3\times 8} \approx 2^{-10}$ which cannot be easily satisfied by randomizing one byte difference. One would often have to restart at step 2, i.e., pick another output difference. The same situation takes place when we move to the next row and pick a

new output difference. In this case we have to start from row 0. As a result, the complexity of finding solutions to the 8 rows is not purely added [107]. Based on our experimental results, the complexity of this inbound path is in the order of 2^{18}. However finding this match means finding at least 2^{64} actual state values for M_2^X, such that both M_2^X and $M_2^X \oplus$ (input difference) follow the inbound path. Each value out of the 2^{64} values is a new starting point to satisfy the two outbound paths. In the following section, we present our attack on the internal block cipher of the Streebog compression function.

All the compression function paths require that the full active state which is the most expensive part of the path to be placed in the middle and consequently difference in the first and last sates must be equal so that they cancel out after feedforward. To efficiently extend the attack to more rounds, several proposals that solve wider inbounds have been published in [104]. In Section 3.4, we briefly recall two of the practical proposals which we use in our attack.

3.4 Attacks on the Internal Block Cipher (E)

Verifying the ideal behaviour of the internal primitives of a hash function is important to evaluate its resistance to distinguishing attacks [37]. In this section we investigate the internal block cipher (E) and, by employing the idea of successive local collisions, we present a message differential path that collides every two rounds. This message differential path enables us to efficiently produce 5-round semi free-start collision and 7.75-round 40 bytes (out of 64) semi free-start near collision. The main idea of our approach is to first find a pair of keys that follows a given differential path and then use it to search for a pair of messages satisfying the message path. The approach of creating local collisions works perfectly if the key and the message flows are identical and the initial key is the input chaining value. To this end, one can keep similar differential patterns and the state message difference is cancelled after the X transformation, so that a collision is obtained after the Miyaguchi-Preneel feedforward. However, in the compression function of Streebog the key used in the internal cipher is the result of applying the K_N transformation on the input chaining value. Similar differential patterns can be obtained when considering the internal block cipher. In our attack on the Streebog internal cipher, we present a message differential path such that a local collision is enforced every two rounds. Specif-

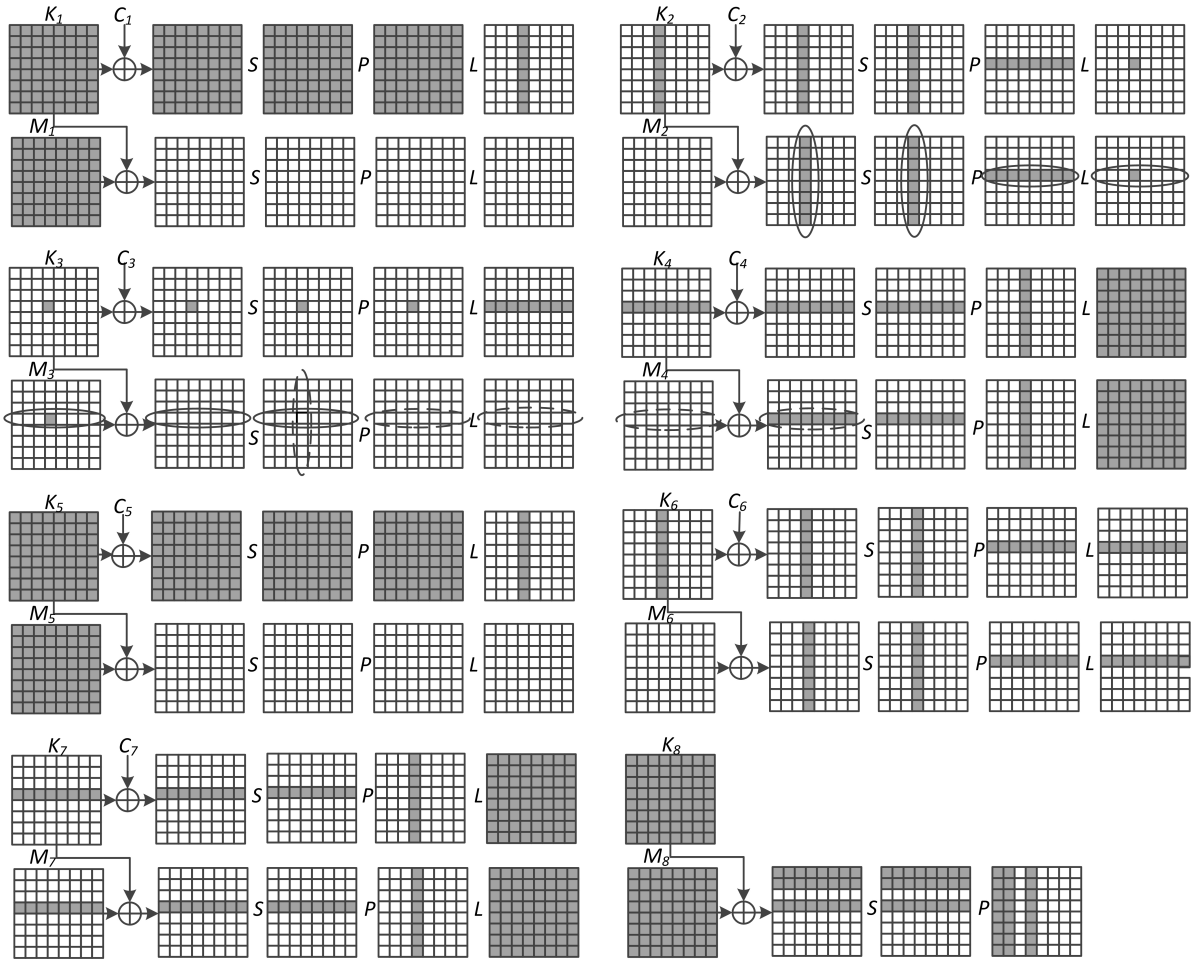

Figure 3.4: 7.75 round differential path. Active bytes are gray colored. Ellipses mark the row and column restricted by the two inbounds.

ically, we first search for a pair of keys that satisfies the key differential path, then we use the Sbox differentials in the key path for the message path. Consequently, we bypass the complexity caused by the Sbox DDT matching in the message differential path and only one key pair is required to search for a message pair. In [132], Sasaki *et al.* presented a message differential path that creates local collisions every four rounds for the Whirlpool compression function and reported that they had to try 109 key pairs to search for a message pair that collides every 4 round. Furthermore, they estimated an increase in the message search complexity by a factor of 2^7 and attributed this to the imbalance of the Sbox DDT. Given the Streebog Sbox DDT, finding one key pair that follows the 8-round differential path takes up to two hours on a 4-core Intel i7 CPU running at 2.67GHz. Accordingly, it is important that the message differential path requires only one key pair to be satisfied. In what follows, we give the details of our approach.

3.4.1 5-round Free-start Collision

Since the Streebog's Sbox DDT is biased with possible differential probability ≈ 0.42, we bypass the Sbox matching phase by using a message differential path such that local collisions are created every two rounds. The used key and message paths are given by:

$$\text{Key: } 64 \xrightarrow{r_1} 8 \xrightarrow{r_2} 1 \xrightarrow{r_3} 8 \xrightarrow{r_4} 64$$

$$\text{Message: } 64 \xrightarrow{r_1} 0 \xrightarrow{r_2} 1 \xrightarrow{r_3} 0 \xrightarrow{r_4} 64 \xrightarrow{r_5} 0$$

This message differential path allows us to bypass the rebound matching part completely in our message search because the same input and output Sbox differences in the key path are used for the message path. Thus the matching probability is 1. Unlike the differential paths in [132], our message differential path is satisfied practically using only one key pair. In this attack, we do not use the matching part of the rebound attack in either the key or the message; we only search for one byte value in the message to find a common solution between two rounds which can be considered as a meet in the middle approach. As depicted in Figure 3.4, the steps for finding a key pair can be summarized as follows:

1. Choose a random difference and a random value for byte $K_2^L[3, 3]$
2. Hash backward until K_1.
3. Hash forward until K_5.

Accordingly, we have a key pair following the given key path. Let the differences in M_2^X, M_2^S, M_4^X, and M_4^S be the same as the differences in K_2^X, K_2^S, K_4^X, and K_4^S, respectively. Having the same differences in the message states as in the key states implies that no differential matching is needed at the Sboxes of rounds 2 and 4, and guarantees that the differences in K_3 and M_3 cancel out. Similar observation applies to K_5 and M_5.

To search for a conforming message pair, we need to find a common solution between the Sboxes of rounds 2 and 4 possible solutions. This can be achieved as follows. Since $M_2^X[\text{col } 3]$ and $M_2^S[\text{col } 3]$ differentials are possible, then from the Sbox DDT there are at least 2^8 values for $M_2^X[\text{col } 3]$

that satisfy the path until M_3^S. For all solution M_2^X[col 3], hash forward until M_3^S. Because M_2^X[col 3] is one column after the P, L, X, and S transformations, its transformed value becomes M_3^S[row 3] as indicated by the ellipse in Figure 3.4. We store all possible values of M_3^S[row 3] in a list l. As for M_4^X[row 3], and M_4^S[row 3], hashing all possible solutions backwards restricts the values of M_3^S[col 3]. However we do not store the results in a another list. Because the two restricted results intersect in only one byte $M_3^S[3, 3]$ (the intersection of the two ellipses in Figure 3.4), we compare byte $[3, 3]$ of each backward result against byte $[3, 3]$ from each entry in list l. The success probability for finding a one byte match is 2^{-8} which can be easily fulfilled by the number of entries in l. Once a match is found, we assign the matching list row to M_3^S[row 3] and the backwards column to M_3^S[col 3]. The rest of the 49 unrestricted bytes are free and can be used to satisfy a longer outbound.

3.4.2 8-round Collision and 7.75-round Near Collision Attacks

Extending the 5 round path to 8 rounds adds complexity to the key search part because we need to use an improved version of the rebound attack to get a key pair following a longer differential path. We employ the following message and key differential paths:

$$\text{Key: } 64 \xrightarrow{r_1} 8 \xrightarrow{r_2} 1 \xrightarrow{r_3} 8 \xrightarrow{r_4} 64 \xrightarrow{r_5} 8 \xrightarrow{r_6} 8 \xrightarrow{r_7} 64$$

$$\text{Message: } 64 \xrightarrow{r_1} 0 \xrightarrow{r_2} 1 \xrightarrow{r_3} 0 \xrightarrow{r_4} 64 \xrightarrow{r_5} 0 \xrightarrow{r_6} 8 \xrightarrow{r_7} 8 \xrightarrow{r_8} 0$$

and use the start form the middle technique [104] to solve the key inbound phase between rounds 3 and 5. This approach finds states following a $1 \longrightarrow 8 \longrightarrow 64 \longrightarrow 8$ transition. Unlike the basic inbound that yields 2^{64} solutions, using this approach on Streebog results in only one solution. For AES Sboxes, a solution is expected in a time complexity of 2^8 and memory complexity of 2^8. However, for Streebog's biased Sbox DDT, one practical solution is found between 33 minutes to 2 hours on an 4-core Intel i7 CPU running at 2.67GHz. Accordingly, it is crucial that the key outbound phase has high probability if one is aiming for practical results and no rebound matching is used in the message search so that one key is enough to get a conforming message pair. In the following steps, we briefly describe the procedure we used for solving the $1 \longrightarrow 8 \longrightarrow 64 \longrightarrow 8$ key inbound phase. Figure 3.5 further illustrates the process.

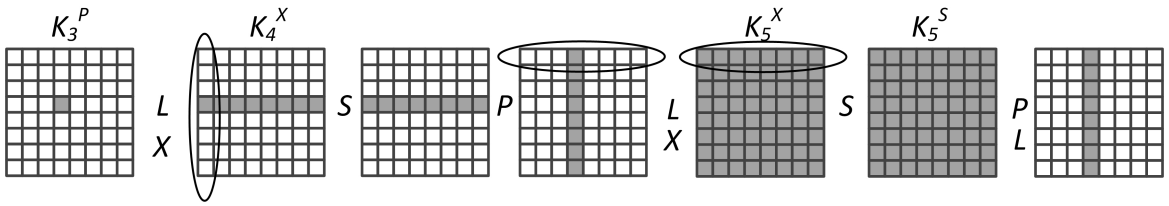

Figure 3.5: Start from the middle approach.

1. Solve the basic inbound $8 \longrightarrow 64 \longleftarrow 8$ as explained in Section 3.

2. From the DDT, each byte difference in K_5^X has at least 2 and at most 8 values, such that any value satisfies the path from K_4^X to K_6.

3. To enforce the transition from 8 active bytes in K_4^X to 1 active byte in K_3^P, do the following:

   a. Create a table T_L of all possible 255 byte difference values d_3 (candidates for $K_3^P[3, 3]$) and their corresponding 8 byte difference values $L(d_3)$ (candidates for K_4^X[row 3]). These values are the result of applying the linear transformation L to a difference at column 3.

   b. Each candidate difference for $K_4^X[row\ 3]$ has 8 active bytes that can be manipulated independently. More precisely, to change the difference value of byte i in K_4^X[row 3], one has to switch between 2^8 or more possible values of K_5^X[row i]. As illustrated by the ellipses in Figure 3.5, a change in the values of K_5^X[row 0] is reflected on the difference value of byte 0 in K_4^X[row 3]

   c. Go through the entries in table T_L and change the values of K_5^X rows one by one until a match is found, if not, restart from step 1.

In [104], the authors follow a different process that consists of three phases to solve this inbound. Their process is supposed to take less time (2^5 vs 2^8) but more memory requirements are needed. However, for Streebog's DDT both approaches were close in the running time and we only needed one key. Finally, by hashing the obtained key pair two rounds backward and two rounds forward, we get a conforming key pair that follows the key differential path. Once we have the key, we can directly get a message pair in the same way as explained in the previous section for the 5-round collision. This message pair satisfies the message differential path up until M_6^L. However, to have an 8-round collision, we need the difference in K_8 to cancel the difference in M_8 after the X transformation in

round 8. Since both L and P transformations are linear, then this condition is satisfied if the 8 byte differences in K_7^S and M_7^S are equal. The difference in K_7^S is already set from the key search stage, so we randomize the 49 unrestricted bytes in M_3^S, hash forward till M_7^S and compare the resulting 8 differences with K_7^S. The probability that the 8 byte differences are equal is 2^{-64}. To verify the applicability of this attack, we have implemented a 7.75-round near collision attack where we were checking if only 5 out of 8 byte differences are equal in M_7^S and K_7^S. In Figure 3.4, the implemented 7.75-round differential pattern, with 2^{40} time and 2^8 state memory complexities is given. Table 3.3 shows an example for a free-start 5-round collision and 7.75-round near collision for the internal cipher (E). Both the 5-round semi free-start collision and the 7.75 semi free-start near collision are demonstrated by one example because the 7.75 semi free-start near collision path collides at round 5.

3.5 Attacks on the Streebog Compression Function

As depicted in Figure 3.1, the compression function of Streebog employs a nonlinear whitening round K_N of the chaining value. This extra round randomizes the chaining value before being introduced as a key for the block cipher E. As long as there is no difference in the chaining value, most of the differential trails proposed for Whirlpool are also applicable on the Streebog compression function.

In what follows, we consider semi free-start collision attacks on the compression function. Because of the extra round K_N which is as a whitening stage for the chaining value, a free-start collision would not be feasible. This is due to the feedforward and the asymmetry in the key and message flows. Several approaches are used to extend the inbound phase can be used to construct collision paths for the compression function. The extended 5 round inbound presented in [95] finds a pair of states satisfying the $8 \xrightarrow{r_1} 64 \xrightarrow{r_2} 8 \xrightarrow{r_3} 8 \xrightarrow{r_4} 64 \xrightarrow{r_5} 8$ transition in 2^{64} time and 2^8 memory. The main idea is to solve two independent $8 \xrightarrow{r_1} 64 \xrightarrow{r_2} 8$ and $8 \xrightarrow{r_4} 64 \xrightarrow{r_5} 8$ inbounds and use the freedom to choose key values that connect the resulting 8 differences and 64 byte values. However, unlike the basic inbound, it provides only one solution or starting point for the outbound paths. Using different outbounds with the extended inbound, a semi free-start 7.75-round collision, and 7.75-round,

8.75-round, and 9.75-round near collisions are obtained.

7.75-round Semi Free-start Collision.

This is obtained by using two outbounds in the form of $8 \longrightarrow 1$. The probability of a transition from 8 active bytes to 1 active byte through L is $2^{-8\times7} = 2^{-56}$. Given the following path:

$$1 \xrightarrow{r_1} 8 \xrightarrow{r_2} 64 \xrightarrow{r_3} 8 \xrightarrow{r_4} 8 \xrightarrow{r_5} 64 \xrightarrow{r_6} 8 \xrightarrow{r_7} 1,$$

one can produce a semi free-start collision. We need two transitions from 8 to 1 in both the forward and backward directions, and the one active byte in the first and last states to be equal so that they cancel out after the feedforward. Thus, one needs to try $2^{56+56+8}$ times to satisfy the outbound phase. In other words, we need 2^{120} inbound solutions. If the complexity of one inbound solution is 2^{64}, then the time complexity of 7.75 rounds semi free-start collision is 2^{184} and the memory complexity is 2^8, as we can pass one active byte through X, S and P transformations with probability one.

7.75-round Semi Free-start Near Collision.

While aiming for collision requires both differences in the first and last states to be exactly in the same place so that they cancel out after the feedforward, near collision requires only few differences to cancel out. A 50-byte near collision is obtained by extending the 5-round inbound with two transitions from 8 to 8 in both directions with no additional cost. Using the following path:

$$8 \xrightarrow{r_1} 8 \xrightarrow{r_2} 64 \xrightarrow{r_3} 8 \xrightarrow{r_4} 8 \xrightarrow{r_5} 64 \xrightarrow{r_6} 8 \xrightarrow{r_7} 8$$

one active byte would cancel out with probability 2^{-8} after feedforward. Consequently, The complexity of 7.75 rounds semi free-start 50-byte collision is 2^{72}. To demonstrate the correctness of the above concept, we have implemented a 4.75-round 50-byte near collision with a shorter practical inbound $8 \xrightarrow{r_2} 64 \xrightarrow{r_3} 8$ as shown in Figure 3.6 with a time complexity of 2^{18}. A 4.75-round near colliding pair is given in Table 3.2 using the $IV = 0$ and $N = 0$.

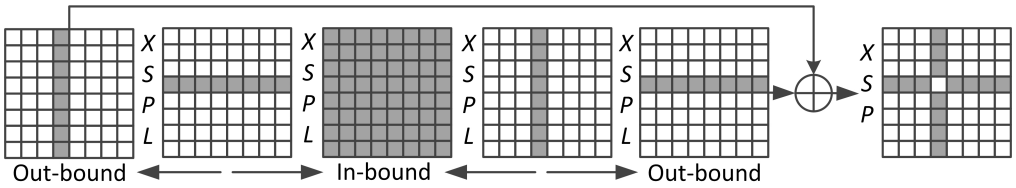

Figure 3.6: 4.75 round near collision path

8.75-round Semi Free-start Near Collision.

Using one transition from 8 to 1 in the forward outbound has a complexity of 2^{56} and results in the following path:

$$8 \xrightarrow{r_1} 8 \xrightarrow{r_2} 64 \xrightarrow{r_3} 8 \xrightarrow{r_4} 8 \xrightarrow{r_5} 64 \xrightarrow{r_6} 8 \xrightarrow{r_7} 1 \xrightarrow{r_8} 8$$

The probability that one active byte is canceled by the feedforward is 2^{-8}. Consequently the complexity of 8.75 rounds semi free-start 50-byte collision is $2^{64+56+8} = 2^{120}$.

9.75-round Semi Free-start Near Collision.

With a complexity of 2^{196}, a 9.75-round 50-byte near collision can be obtained with a lower complexity of 2^{184}. By adding two 8 to 1 transitions in both the forward and the backward directions for a complexity of 2^{112} and two 1 to 8 transitions in rounds one (backward) and nine (forward) for no additional cost, the following path:

$$8 \xrightarrow{r_1} 1 \xrightarrow{r_2} 8 \xrightarrow{r_3} 64 \xrightarrow{r_4} 8 \xrightarrow{r_5} 8 \xrightarrow{r_6} 64 \xrightarrow{r_7} 8 \xrightarrow{r_8} 1 \xrightarrow{r_9} 8$$

results in a 50-byte near collision. Additional complexity of 2^8 is needed for a one byte cancellation after the feedforward.

3.6 Conclusion

In this chapter, we have analyzed the collision resistance of the Streebog compression function and internal cipher. As for the internal cipher, we have proposed a new message differential path such that a local collision is enforced every two rounds. Accordingly, the Sbox matching complexity caused by its DDT bias is bypassed. As a result, we have efficiently produced free-start 5-round collision and 7.75-round near collision examples for the internal cipher. Moreover, the compression function is investigated and we have noted that the Streebog compression function key whitening round K_N enhances its resistance to free-start collision attacks. However, we have showed that the Streebog compression function is vulnerable to semi free-start 7.75 round collision, 8.75 and 9.75 round near collision attacks and presented an example for a 4.75 round 50-byte near colliding message pair.

It should be noted that our results considers only complete full round, after which, the feedforward is applied. More specifically, unlike the results presented in [99, 145], we respect the wide trail strategy and all of our results that are given on $n + 0.75$ rounds are in fact applicable on the n-round versions. On the other hand, all the results presented in [99, 145] try to align the input and output differences of the differential path by applying the feedforward in the middle of the round after the transpose transformation.

m								m'								Difference at M_4							
cd	ed	17	46	d8	d7	f0	f3	cd	ed	17	59	d8	d7	f0	f3	00	00	00	1f	00	00	00	00
3e	d6	22	7a	99	4a	c9	ea	3e	d6	22	0c	99	4a	c9	ea	00	00	00	76	00	00	00	00
cc	5d	e2	f0	14	4f	f0	3c	cc	5d	e2	ea	14	4f	f0	3c	00	00	00	1a	00	00	00	00
4b	bc	31	41	dd	99	68	0d	4b	bc	31	4d	dd	99	68	0d	ba	38	7a	00	6f	93	95	37
b4	d1	27	0f	2d	ed	55	28	b4	d1	27	58	2d	ed	55	28	00	00	00	57	00	00	00	00
d8	ca	c8	79	22	fa	c8	14	d8	ca	c8	f6	22	fa	c8	14	00	00	00	8f	00	00	00	00
9f	06	fe	94	b3	3d	20	6a	9f	06	fe	80	b3	3d	20	6a	00	00	00	14	00	00	00	00
5a	d6	10	10	51	4c	a3	7a	5a	d6	10	2b	51	4c	a3	7a	00	00	00	3b	00	00	00	00

Table 3.2: Example of a 4.75-round near collision for the compression function.

m								m'								Difference at M_7^P							
ba	aa	da	d1	92	9e	95	f5	3b	16	1b	b0	76	fe	1e	78								
3a	4a	35	2c	61	a8	84	f1	4c	03	4f	12	d1	a3	b4	bd								
44	38	38	e2	d2	fa	5e	ec	c6	a7	81	ff	3a	c7	3e	36								
27	00	09	05	4f	53	05	f2	6c	76	3e	0a	d6	92	72	00								
cd	02	30	bb	3e	b4	54	df	47	7e	c6	e0	a4	6e	23	1a								
fc	c6	de	98	54	4e	5c	b6	28	a4	20	68	ee	e1	01	11	d7	4d	00	c8	00	00	00	00
60	dc	52	73	dc	c9	5d	f1	43	20	0a	43	12	ba	fe	a0	ff	60	00	60	00	00	00	00
72	99	45	8d	9b	c8	73	f2	8a	d2	ff	b3	19	f4	e4	25	15	3c	00	c9	00	00	00	00
																1b	49	00	ae	00	00	00	00
				k								k'				03	81	00	42	00	00	00	00
																1a	ed	00	ea	00	00	00	00
f4	d7	d6	42	05	a4	b9	7a	75	6b	17	23	e1	c4	32	f7	37	8e	00	60	00	00	00	00
2f	70	68	1a	2c	59	f4	4e	59	39	12	24	9c	52	c4	02	61	b8	00	f2	00	00	00	00
8b	7b	44	12	38	36	84	87	09	e4	fd	0f	d0	0b	e4	5d								
63	04	2f	7d	de	3d	b9	9f	28	72	18	72	47	fc	ce	6d								
78	db	37	55	73	39	f7	30	f2	a7	c1	0e	e9	e3	80	f5								
3f	f2	8d	fb	23	a9	6a	8a	eb	90	73	0b	99	06	37	2d								
20	18	3a	e4	63	85	3a	81	03	e4	62	d4	ad	f6	99	d0								
b5	58	8a	e7	d3	34	20	4d	4d	13	30	d9	51	08	b7	9a								

Table 3.3: Example of a 5-round collision and 7.75-round near collision for the internal block cipher (E).

Chapter 4

Integral Distinguishers for Streebog

In this chapter, we investigate the structural integral properties of reduced-round versions of the Streebog compression function and its internal permutation. Specifically, we present forward and backward higher order integrals that can be used to distinguish 4 and 3.5 rounds, respectively. Using the start from the middle approach, we combine the two proposed integrals to get 6.5-round and 7.5-round distinguishers for the internal permutation and 6-round and 7-round distinguishers for the compression function using 2^{64} and 2^{120} middle input states, respectively. Moreover, following the simplified representation of AES [62], we extend our original work to 8 rounds by considering an alternative representation of the twelve rounds Streebog internal cipher. In Table 4.1, we provide a summary of our results on the underlying primitives of the Streebog hash function.

Target	#Rounds	Complexity Time,Mem	Attack
Compression function	6	2^{64} states	Integral distinguisher
	7	2^{120} states	
	8	2^{128} states	
Internal permutation	6.5	2^{64} states	
	7.5	2^{120} states	

Table 4.1: Summary of the integral cryptanalysis results on the Streebog primitives.

4.1 Introduction

Modern cryptanalytic approaches target both the hash function and its underlying ciphers or permutations as these components are expected to provide certain properties and verifying their ideal behaviour is important to evaluate the resistance of the hash function to distinguishing attacks [37]. Particularly, the analysis of hash functions underlying block ciphers or permutations has resulted in new attack models for block ciphers, e.g., known key [87]. Such model is due to the fact that there is no secret key when block cipher based structures are used as the hash function building blocks.

In the first part of the chapter, we focus on the integral properties and their applications to present the first known integral distinguishers for the Russian cryptographic hash standard compression function and its internal permutation. We present a 4-round 8^{th} order integral for the forward direction and a 3.5-round 8^{th} order integral for the backward direction, where both integrals are satisfied by 2^{64} inputs. In the second part, we present 6.5-round and 7.5-round distinguisher for the internal permutation using 2^{64} and 2^{120} middle inputs, respectively and 6-round and 7-round integral distinguishers for the compression function that are satisfied by 2^{64} and 2^{120} middle input states, respectively. Lastly, we show how using the simplified representation of AES [62], we can extend our attacks on the compression function to cover 8 rounds.

The rest of this chapter is organized as follows. In the next section, a brief overview of integral cryptanalysis is given. Afterwards, in Sections 4.3, we provide detailed description of the integral patterns and the complexities of the distinguishers. An alternative representation of the Streebog internal cipher and an 8-round integral distinguisher for the compression function are given in Section 4.4. Finally, the chapter is concluded in Section 4.5.

4.2 Integral cryptanalysis

Integral cryptanalysis was proposed by Knudsen and Wagner in [88]. It is considered as a dual to differential cryptanalysis and is efficient against ciphers that are resistant to differential attacks. While In differential cryptanalysis, one considers the propagation of differences between pairs of val-

ues to obtain probable differentials, in integral cryptanalysis, we consider the propagation of sums of many values to obtain integrals. Integral cryptanalysis is specifically designed for block ciphers which use only bijective transformations. An integral is a set of values with a specific input and output sums. It covers several rounds of the cipher and describes how the summation properties of a set of input values would be affected by each successive round.

Before being formalized in [88], the idea of integral attacks has been explored under several names. It was first discovered during the analysis of the square cipher [43] and named the square attack. Following this, the attack was generalized into the saturation attack and was used to analyze the Twofish cipher [98]. Ever since higher order integrals have been introduced in [88], integral cryptanalysis has been used to analyze block ciphers in the known key setting [87, 113, 131] and to present distinguishers for the components of hash functions.

Integrals properties.

For a given collection of (8×8)-byte states, a typical integral uses m chosen input states, where m equals $2^8 \times$ (number of active bytes). A state byte position can have any of the following properties:

- C: A constant byte, where all the bytes at this position in the m states are equal. However, If two byte position at the same state have the C property, that does not necessarily mean that they are equal.

- A: An active byte, where all the bytes at this position in the m states are different. Specifically, if $m = 2^8$, then each byte in that position takes a value between 0 and $2^8 - 1$ only once.

- A^d: An active byte that participates in a d^{th}-order integral. If a byte takes 2^8 different values, then A^d means that this particular byte takes all values exactly $2^{8(d-1)}$ times. A byte with the A^d property also satisfies the A property.

- A_i^d: An active byte that participates in a d^{th}-order integral within a group. In particular, the string concatenation of all bytes with subscript i take the 2^{8d} values exactly once. A byte with the A_i^d property satisfies both the A_d and A property.

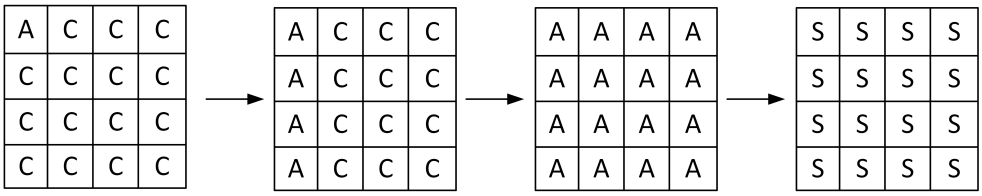

Figure 4.1: A 3-round first order integral for Rijndael

- S: The sum of all bytes at this position can be predicted. All the C, A, A^d, and A_i^d properties satisfies the S property where their predictable sum is zero. The S property is the weakest of them all as it reveals so little about the relation between bytes at similar positions in a set of states.

In order to be able to use an integral as a distinguisher, we expect that at least one entry in the output set of values satisfies a predictable property. Similar to truncated differentials [89] where one considers if a specific entry is active or not, in a given integral we consider if an input has an A property or a C property. As mentioned earlier, a typical integral uses $2^{8\times\#\text{ active bytes}}$ inputs. An integral having one active byte is called a first order integral and can be satisfied with 2^8 chosen inputs. On the other hand considering an integral with a group of active bytes results in a higher order integral.

An example of a 3-round first order integral for Rijndeal is given in the first proposal [88] by Knudsen and Wagner and is shown in Figure 4.1. To further explain the idea of integral propagation through successive rounds, we give a detailed example on the above Rijndeal first order integral. One round applies 4 transformations on a state , which are byte substitution (SB), row cyclic shifting (SR), linear transformation (MC), and Key addition (AK). Consider a set of 2^8 input states, such that they have different values in $M[0, 0]$ and equal values in the rest of the fifteen bytes. the transformation SB keeps the same property because it is bijective so each byte is substituted with a unique one. Afterwards, the SR transformation affects only the constant bytes keeping the state of the integral as is, then the MC transformation mixes the active byte with three constant bytes in column 0 and results in a column full of active bytes. Finally, due to the fact that the AK transformation XORs the same key with all the 256 state, the sum of all states remain the same at the end of the round. As with differential propagation, after two encryption rounds all the sixteen bytes in all the 256 states become active. However, this integral can go one more encryption round and we get a 256 states that sum to

zero in all the sixteen byte positions.

Constructing a integral distinguisher can be viewed as a zero sum problem. Accordingly, to estimate the expected complexity of having a random set of states produce a distinguisher with a final balanced properties, the k-sum problem [141] was introduced in [87] to model this complexity. The k-sum problem finds a set of k inputs $x_1, ..., x_k$ such $\sum_{i=1}^{k} f(x_i) = 0$ for a given permutation f. This problem has a time and memory complexity of $O(k2^{n/(1+log_2 k)})$, where n is the size of the state in bits. The k-sum problem is the best generic known approach suited to this case to find the zero sum. However, it does not provide the structured propertied of the distinguisher as hashing rounds progress and has high memory requirements.

Previous literature related to integral cryptanalysis of hash functions include the analysis of Minier *et al.* of the three SHA-3 candidates; Hamsi-256, LANE-256 and Grøstl-512 [112] and recently Grøstl-512 [114], and Knudsen's attack on whirlpool internal block cipher [86].

4.3 Distinguishers for the Streebog Primitives

The compression function of the Streebog hash function employs an AES-based cipher. In Figure 4.2, we present an 8^{th} order integral distinguisher for the Streebog internal cipher. In this distinguisher, the sum of all the bytes in all the states after four rounds of encryption with the same key is equal zero. To build this distinguisher, we consider 2^{64} input states M_1 that have equal values in 56 bytes and differ in only eight bytes. These states differs in the eight bytes in column three such that each state $M_1[\text{col } 3]$ (out of the 2^{64}) state takes a value between 0 and $2^{64} - 1$ only once (the place of the column is arbitrary). After four complete rounds of hashing forward (encryption) we get 2^{64} states M_5, such that all the 64 bytes sum to zero.

The fact that the Streebog round transformations are bijective allows us to build integrals in the backward direction (decryption). In Figure 4.3, we present a backward integral for 3.5 rounds of Streebog internal permutation. Although the third round integral properties are still giving a lot of information about the integral, i.e., $M_2[\text{col } 0, 1, ..7]$ all have grouped 8^{th} order properties, we only get

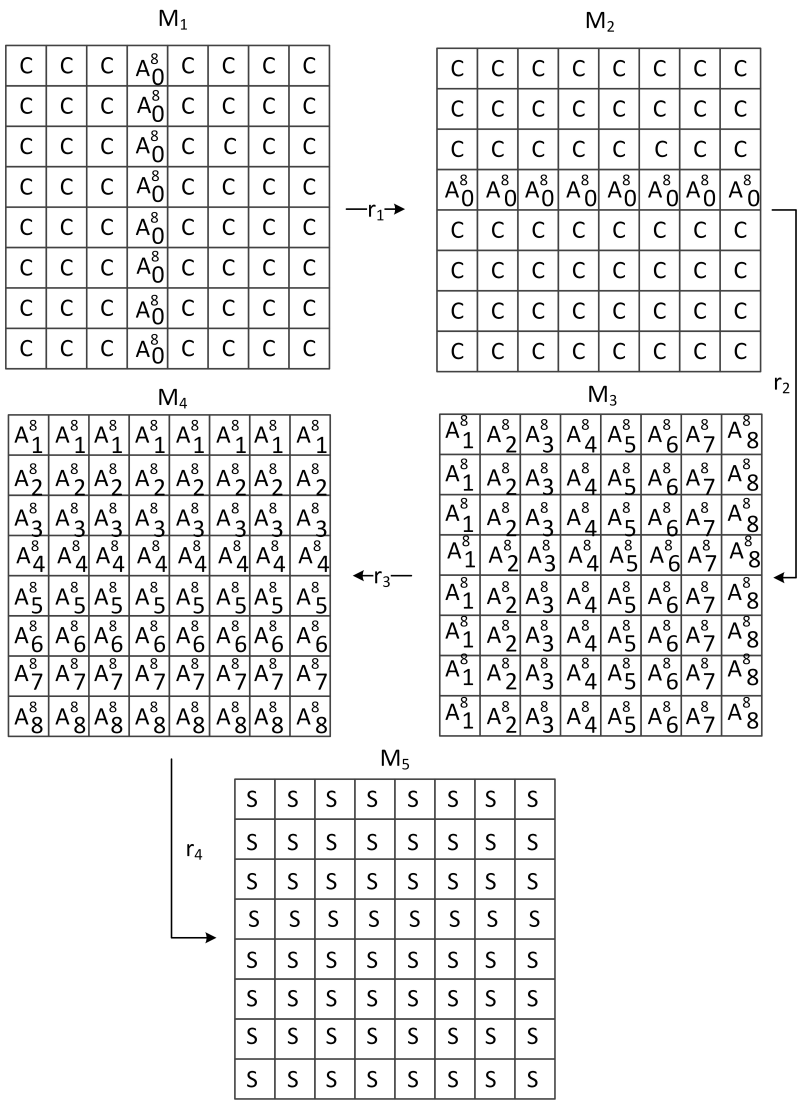

Figure 4.2: An example for a forward 4-round 8^{th} order integral for the Streebog permutation. S means the sum is equal zero

S property integral at states M_1^S after applying the inverse linear transformation that processes the state row by row. Consequently, extending the backward integral to four rounds does not preserve the S property because the nonlinear substitution transformation does not preserve this property. To construct the backward distinguisher, we consider 2^{64} input states M_4 that have equal values in 56 bytes and differ in only eight bytes. These states differs in the eight bytes in row three such that each state row $M[\text{row } 3]$ takes a value between 0 and $2^{64} - 1$ only once. Following 3.5 rounds of hashing backward (decryption) we get 2^{64} states, such that all the 64 bytes sum to zero.

In order to cover more rounds, we employ the start from the middle approach. Using this approach we can combine the forward and backward integrals over more than 7 rounds of the Streebog

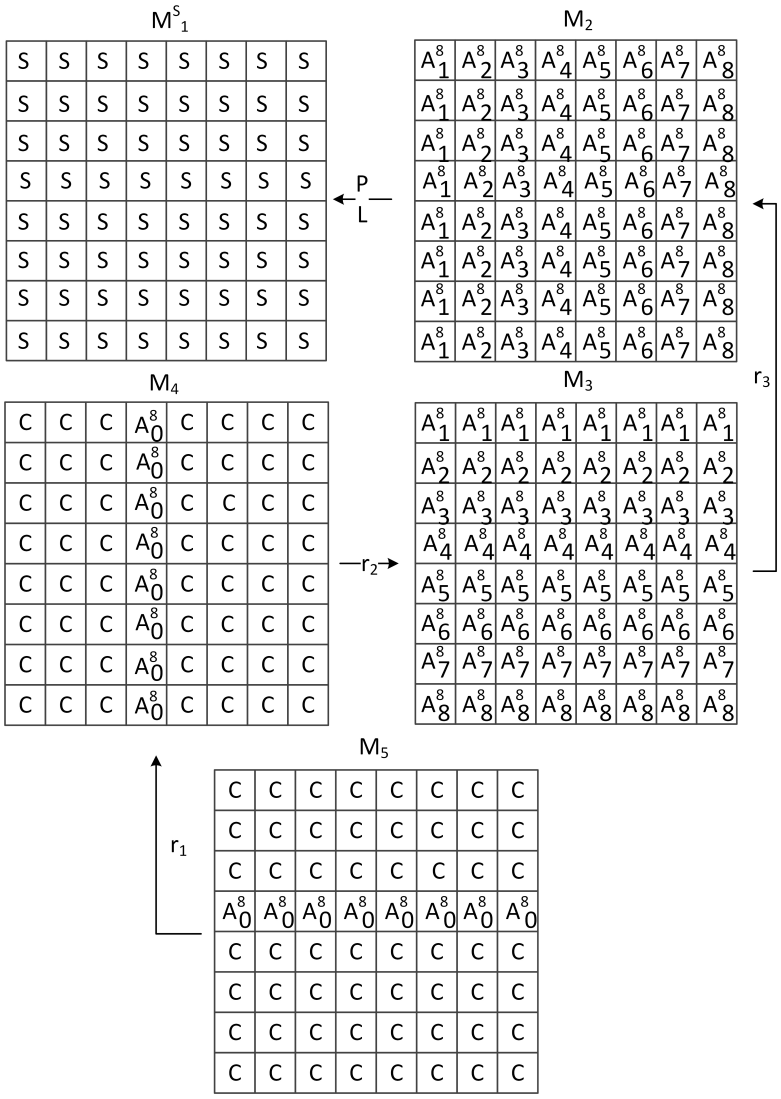

Figure 4.3: An example for a backward 3.5-round 8^{th} order integral for the Streebog permutation. S means the sum is equal zero

internal permutation. In Figure 4.4, a 15^{th} order integral 7.5-round distinguisher for the Streebog permutation is given. Moreover, we can obtain an 8^{th} order integral to distinguish 6.5 rounds of the internal permutation by using 2^{64} middle states only. Such integral is obtained by combining the forward integral shown in Figure 4.2 with only the two rounds that start with states M_4 from the backward integral shown in Figure 4.3. The 7.5-round integral is constructed by choosing a set of 2^{120} middle states M_4 a structure that have equal values in 49 bytes and differ in 15 bytes. Each middle state different bytes takes a value between 0 and $2^{120} - 1$ only once. Finally, hashing forward for 4 rounds and backward for 3.5 rounds we obtain the 7.5-round integral distinguisher for the Streebog internal permutation. Although both the forward and backward integrals are 8^{th} order integrals, one

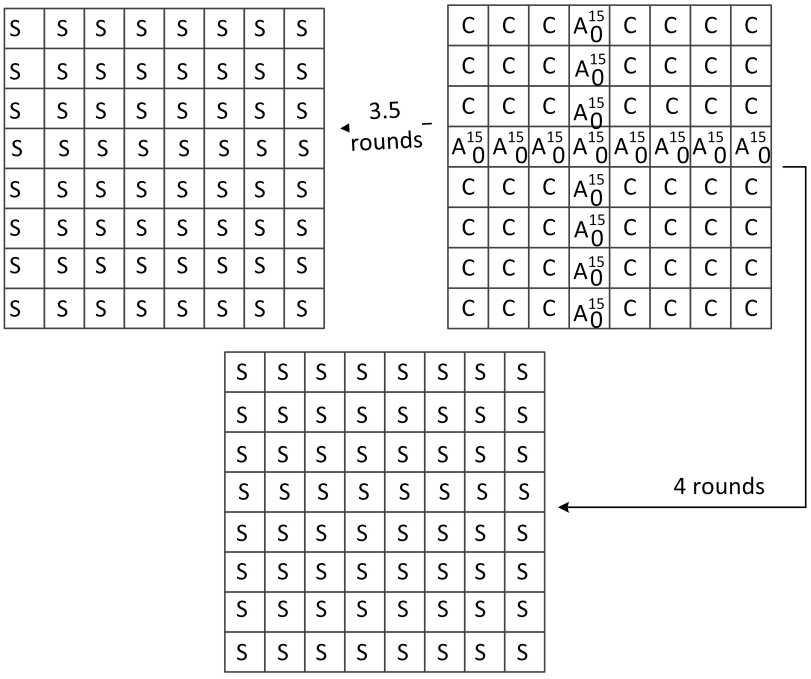

Figure 4.4: An example for a 7.5-round 15^{th} order integral for the Streebog internal permutation. S means the sum is equal zero

can perceive the set of 2^{120} middle states used for the 15^{th} order integral as a set of 2^{56} sets of the forward 4-round integral and also 2^{56} sets of the backward 3.5-round integral.

Compression Function Distinguishers.

A 7-round 15^{th} distinguisher for the reduced compression function can be obtained after applying the Miyaguchi-Preneel feedforward and we would still have a fully balanced integral. The compression function distinguisher is shown in Figure 4.5. Additionally, one can construct a compression function integral distinguisher that covers 6 rounds which are equivalent to half of the compression function rounds using 2^{64} middle states only (See Figure 4.6). This distinguisher is obtained by combining the forward integral shown in Figure 4.2 with only the two rounds that start with states M_4 from the backward integral shown in Figure 4.3.

4.4 Extending the Distinguisher to 8 Rounds

In this representation, the internal cipher is viewed as a sequence of six *super rounds* proceeded and followed by a transpose operation. Each *super round* replaces two consecutive regular rounds and

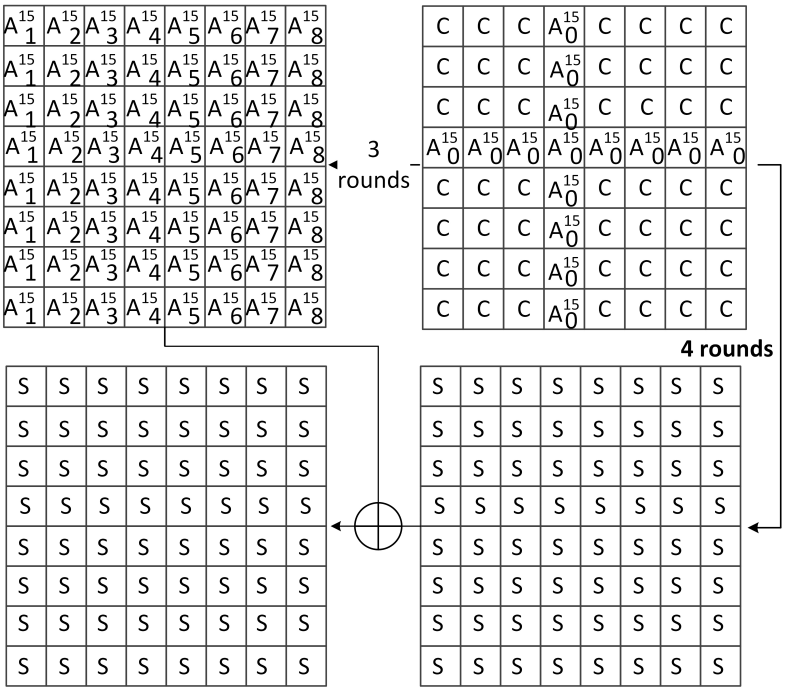

Figure 4.5: An example for a 7-round 15^{th} order integral for the Streebog compression function. S means the sum is equal zero

is composed of two 64-bit transformations:

- A non-linear transformation (SS) which consists of eight 64-bit bijective super Sboxes and operates on the eight rows simultaneously.

- A linear transformation (C) which consists of eight linear transformations applied on the eight 64-bit columns.

In order to demonstrate the new representation of the cipher, we denote the composition of two transformation A and B by $A \cdot B$ instead of using the classical notation $B \circ A$ as reading the former notation from left to right describes the successive transformations that are applied to the input. In Streebog, a two consecutive rounds are composed of the following transformations:

$$S \cdot P \cdot L \cdot X \cdot S \cdot P \cdot L \cdot X,$$

since P can be applied before or after S, then the two rounds can be written as:

$$P \cdot (S \cdot L \cdot X \cdot S) \cdot P \cdot L \cdot X,$$

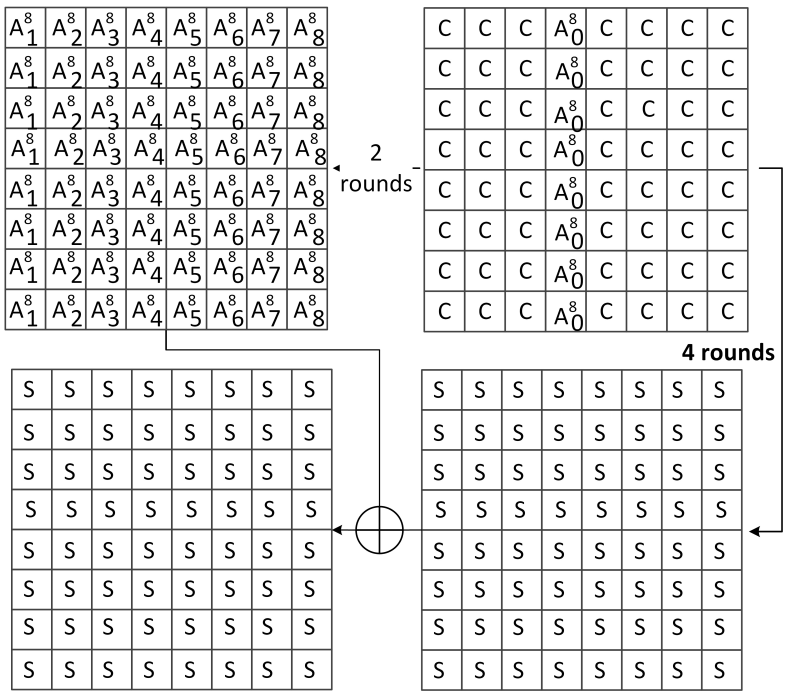

Figure 4.6: An example for a 6-round 8^{th} order integral for the Streebog compression function. S means the sum is equal zero

where $(S \cdot L \cdot X \cdot S)$ represents the super Sbox layer SS that operate on the 64-bit rows. Moreover, if one observes an r repetitions of the above 2-round representation $P \cdot SS \cdot P \cdot L \cdot X$,, one can view it as an r repetition of the following shifted pattern:

$$SS \cdot (P \cdot L \cdot X \cdot P)$$

To this end, one can see that due to the fact that P is a transpose operation which means that it works on both rows and column, the right composed transformation $(P \cdot L \cdot X \cdot P)$ is a linear mapping that operates on the individual columns of the input state. Accordingly, our two round representation is composed of $SS = S \cdot L \cdot X \cdot S$ and $C = P \cdot L \cdot X \cdot P$, and we only need to add a transpose P operation before and after the first and last round, respectively. Consequently, the Streebog internal cipher can be expressed as:

$$X \cdot P \cdot (SS \cdot C)^6 \cdot P.$$

In what follows, we describe how this new representation is used to present an eight round integral distinguisher of the Streebog compression function.

Figure 4.7 depicts the proposed eight round distinguisher for the internal cipher. Since SS is a bijective row mapping and C is a column linear mapping, it can be seen by following the row-wise transitions of the input sate Z through transformations SS, C, and SS, results in a pattern of an all balanced output. Afterwards, C mixes all balanced columns through an MDS matrix multiplication and P only transposes the result. Hence, we get an output state with a predictable sum in the forward direction. Similarly, when following the transitions of the input state for the backward direction $C^{-1}(Z)$ through $(SS.C.SS)^{-1}$ transitions, we get an all balanced state which also has a predictable sum. Since the compression function output is given by the Xor addition of the cipher input state and the output states along with the input chaining value, which is composed of an all constant state, we get a compression function output with a predictable sum. The main idea of this distinguisher using the new

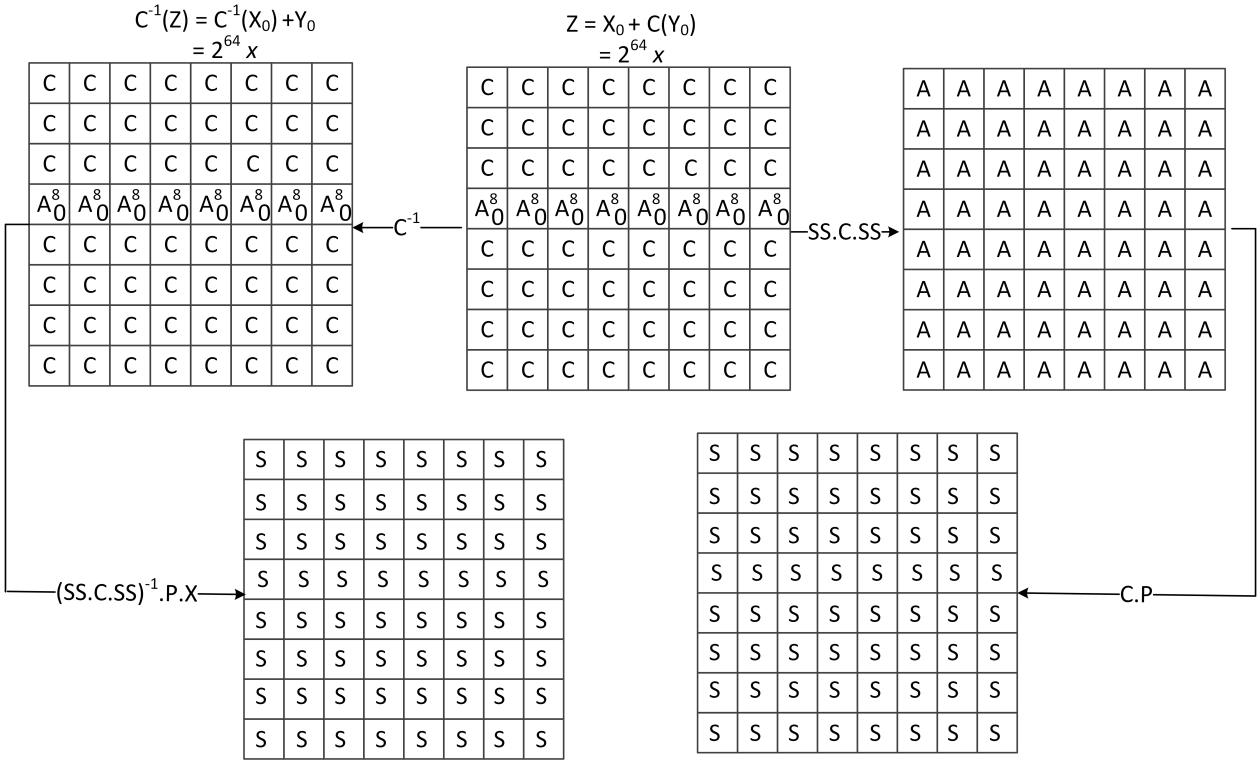

Figure 4.7: An eight round distinguisher for the Streebog internal cipher

representation is that the input state of the forward direction Z and the input state of the backward direction $C^{-1}(Z)$ are linked together through C^{-1}, although, they do not map into each other through C. Let, $X_{(0,0,0,x,0,0,0,0)}$ or $Y_{(0,0,0,y,0,0,0,0)}$ denote a structure of 2^{64} states, where all the rows have zero value except the forth row which takes the 2^{64} values of x or y, for $x, y \in \{0,1\}^{64}$. In this distinguisher, we use a 2^{128} chosen middle blocks structure $Z = X_0 \oplus C(Y_0)$, where X_0 and Y_0 denote $X_{(0,0,0,x,0,0,0,0)}$

and $Y_{(0,0,0,y,0,0,0,0)}$, respectively, and $X_0 \oplus C(Y_0)$ denotes the set $\{X \oplus C(Y), X \in X_0, Y \in Y_0\}$. Accordingly, Z can be partitioned into 2^{64} structures $X_0 \oplus C(0,0,0,y,0,0,0,0,) = X_{C(0,0,0,y,0,0,0,0,)}$ of 2^{64} blocks each, one for each value of the 2^{64} values of y. Accordingly, the proposed eight round distinguisher requires 2^{128} middle states and has a time complexity of 2^{128}.

4.5 Conclusion

In this chapter, we have analyzed the integral properties of the compression function and the internal permutation of the new Russian cryptographic hashing standard GOST R 34.11-2012. As for the internal permutation, we have proposed two integral distinguishers that cover 4 and 3.5 rounds in the forward and backward directions, respectively. Moreover, we have shown that using the start from the middle approach, we are able to combine these two integrals to obtain a 7.5-round and 6.5-round distinguishers for the internal permutation in the known-key setting that holds with probability 1 and are satisfied by 2^{120} and 2^{64} middle states, respectively. Furthermore, we extended this approach based on the integral output properties to the compression function after applying the feedforward to distinguish 6 and 7 rounds out of 12 rounds with probability 1 and 2^{64} and 2^{120} middle states, respectively. Finally, we have shown that by adopting an alternative representation of the Streebog internal cipher, we can further extend the distinguisher to 8 rounds of the compression function.

Chapter 5

Preimage Analysis of Streebog

In this chapter, we investigate the preimage resistance of the Streebog hash function. In particular, we apply a meet in the middle preimage attack on the compression function which allows us to obtain a 5-round pseudo preimage for a given compression function output with time complexity of 2^{448} and memory complexity of 2^{64}. Additionally, we adopt a guess and determine approach to obtain a 6-round chunk separation that balances the available degrees of freedom and the guess size. The proposed chunk separation allows us to attack 6 out of 12 rounds with time and memory complexities of 2^{496} and 2^{112}, respectively. Finally, by employing a multicollision attack, we show that preimages of the 5 and 6-round reduced hash function can be generated with time complexity of 2^{481} and 2^{505}, respectively. The two preimage attacks have equal memory complexity of 2^{256}. Our results are summarized in Table 5.1.

Target	#Rounds	Time	Memory	Attack
Compression function	5	2^{448}	2^{64}	Pseudo preimage
	6	2^{496}	2^{112}	
Hash function	5	2^{481}	2^{256}	Preimage
	6	2^{505}	2^{256}	

Table 5.1: Summary of the preimage cryptanalytic results on Streebog presented in this chapter.

5.1 Introduction

Following the work of Lai and Massey [94], the meet-in-the-middle (MitM) preimage attack was proposed by Aoki and Sasaki [20]. The main idea of the proposed technique is to divide the attacked rounds into two independent executions such that each execution is affected by a different set of inputs. The outputs of the two executions meet at a matching point where a solution is selected to satisfy both executions. The MitM preimage attack has been applied to MD4 [20, 66], MD5 [20], HAS-160 [71], and all functions of the SHA family [18, 19, 66]. The attack exploits the fact that all the previously mentioned functions are ARX-based and operate in the Davis-Mayer (DM) mode, where the state is initialized by the chaining value and some of the expanded message blocks are used independently each round. Thus, one can determine which message blocks affect each execution for the MitM attack. However, several AES-based hash functions operate in the Miyaguchi-Preneel mode, where the input message is fed to the initial state which undergoes a chain of successive transformations. Consequently, the process of separating independent executions becomes relatively more complicated.

In FSE 2011, Sasaki proposed the first MitM preimage attack on several AES hashing modes [130]. In the same work, a 5-round pseudo preimage attack on the compression function of Whirlpool was presented and used for a second preimage attack on the whole hash function. Afterwards, Wu *et al.* applied the MitM preimage attack on Grøstl [150] and used a time-memory trade off approach to improve the time complexity of the 5-round attack on the Whirlpool compression function. Lastly, a pseudo preimage attack on the 6-round Whirlpool compression function and a memoryless preimage attack on the reduced hash function were proposed in [132].

In the first part of this chapter, we present a pseudo preimage attack on the compression function reduced to 5 out of 12 rounds by employing the partial matching and initial structure concepts [130]. In particular, we present an execution separation for the compression function that balances the degrees of freedom in both execution directions with their corresponding matching probability [150]. In the second part of the chapter, we extend the attack by one round using the guess and determine approach [132], which allows us to guess parts of the state that belongs to one execution. The proposed 6-round

chunk separation maximizes the overall complexity of the attack by balancing the adopted degrees of freedom and the guess size. Finally, we show how to generate preimages of the Streebog hash function using the presented pseudo preimage attacks on the compression function.

The rest of the chapter is organized as follows. In the next section, a brief overview of the MitM preimage attack and the used approaches is given. Afterwards, in Sections 5.3 and 5.4, we provide detailed description of the attacks and their corresponding complexity. In Section 5.5, we show how preimages of the hash function are generated using the attacks presented in Sections 5.3 and 5.4. Finally, the chapter is summarized and a short discussion is provided in Section 5.6.

5.2 MitM Preimage Attacks on AES-based Hash Functions

The first preimage attack on AES-based hash functions [130] was proposed for the cryptanalysis of the AES cipher operating in several hashing modes. It is a meet in the middle attack where the attacked rounds are divided at a given round (starting point) into two independent executions called the forward and backward chunks. To maintain the independence constraint, each chunk must be influenced by a different set of inputs. These set of inputs are often called the chunk neutral bytes, e.g., if a change in a given byte affects the forward chunk only, then this byte is known as a forward neutral byte, and consequently, it is a forward degree of freedom as well. Accordingly, the degree of freedom for each execution direction is the number of independent starting values for each execution. Hence, the output of the forward and the backward executions can be independently calculated and stored. Similar to all MitM attacks, the two separated chunks must meet at a common round (matching point) for matching a solution from both the forward and backward directions that satisfies both executions. This is accomplished by adopting the cut and splice technique [20] that employs the mode of operation of the hash functions which chains the input and output states through feedforwarding. More precisely, this technique regards the first and last states as successive rounds. Subsequently, the whole attacked rounds behave in a cyclic manner and one can find a common matching point between the forward and backward executions and one can also select any starting point.

Improvements to this attack aim to stretch the starting and matching points over more than one round state and hence extend the number of the overall attacked rounds. Specifically, the initial structure approach [130] provides the means for the starting point to cover a few successive transformations where bytes in the states belong to both the forward and backward chunks. Although, neutral bytes of both chunks are shared within the initial structure, independence of both executions is achieved in the rounds at the edges of the initial structure. Additionally, the partial matching technique [20] allows only parts of the state to be matched at the matching point. This method is used to extend the matching point further and makes use of the fact that round transformations may update only parts of the state. Thus the remaining unchanged parts can be used for matching. This approach is highly successful in ARX-based hash functions which are characterized by the slow diffusion of their round update functions and so some state variables remain independent in one direction while execution is in the opposite direction. The unaffected parts of the states at each chunk are used for partial matching at the matching point. However, in AES-based hash functions, full diffusion is achieved after two rounds and this approach can be used to extend the matching point of two states for a limited number of transformations. Once a partial match is found, the inputs of both chunks that resulted in the matched values are selected and used to evaluate the remaining undetermined parts of the state at the matching point to check for a full state match.

Figure 5.1 illustrates the MitM preimage attack approaches when a hash function operates in the Miyaguchi-Preneel mode. The red and blue arrows denote the forward and backward executions on the message state, respectively. In what follows, we apply the techniques discussed in this section to

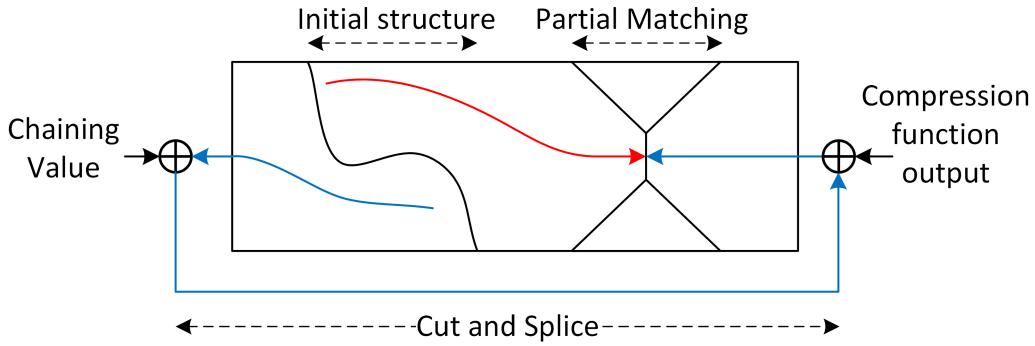

Figure 5.1: MitM preimage attack techniques for hash functions operating in MP mode.

derive a 5-round pseudo preimage attack on the Streebog compression function.

5.3 5-round Pseudo Preimage of the Compression Function

For a compression function CF that operates on a chaining value h and a message block m, a preimage attack is defined as follows: given h and x, where x is the compression function output, find m such that $CF(h, m) = x$. However, in a pseudo preimage attack, only x is given and we must find h and m such that $CF(h, m) = x$. Generally, pseudo preimages of the compression function of some narrow pipe constructions are important because they can be turned to preimages of the hash function with little cost [110]. As for Streebog, the impact of the pseudo preimage attacks on its compression function is demonstrated in Section 6, where we combine these attacks with 2^t multicollision to produce preimages for the hash function. Pseudo preimage attacks are adopted when the compression function operates in Davis-Mayer mode where the first state is initialized by the chaining value. Subsequently, using the cut and splice technique enforces changes in the first state through the feedforward. Additionally, the initial phase of MitM preimage attack usually produces pseudo preimages when the function operates in the Miyaguchi-Preneel mode and the complexity of finding a preimage is higher than the available bits that can be chosen freely in the message. Consequently, the chaining value is utilized as a source of randomization to satisfy the number of multiple restarts required by the attack. As a result, we end up with a pseudo preimage rather than a preimage of the compression function output.

The attack on the compression function starts by chunk separation. Specifically, we divide five rounds of Streebog execution into a forward chunk and a backward chunk around a starting point (initial structure). The adopted chunk separation is shown in Figure 5.2. The forward chunk starts at M_3 and ends at M_4^P which is the input state to the matching point. The backward chunk starts at M_1^P and ends after the feedforward at M_4^L which is the output state of the matching point. The red bytes are the neutral bytes for the forward chunk and after choosing them in the initial structure, all other red bytes can be independently calculated. White bytes in the forward chunk are the ones whose values depend on the neutral bytes of the backward chunk which are the blue bytes in the initial structure.

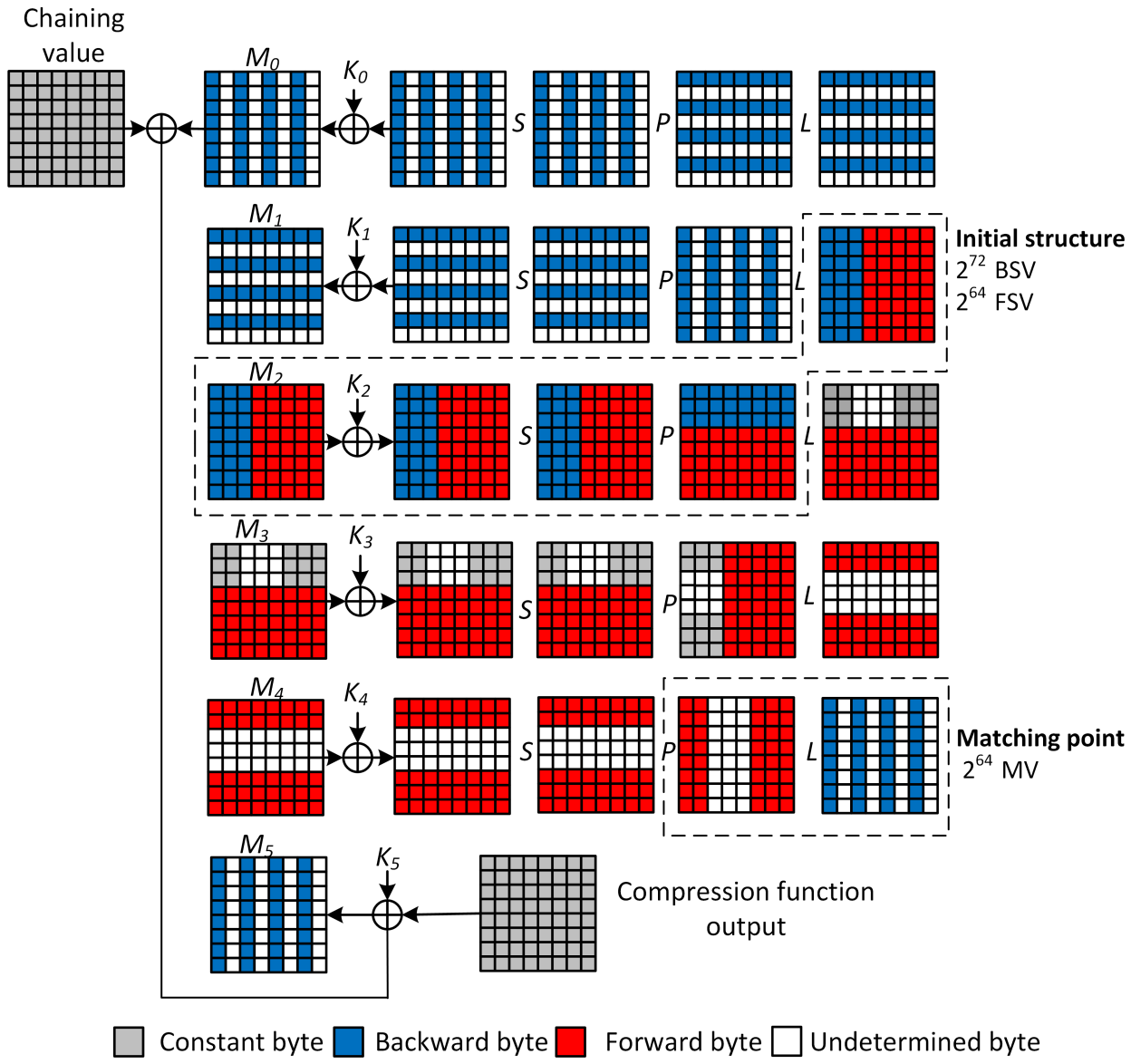

Figure 5.2: Chunk separation for a 5-round MitM pseudo preimage attack on Streebog compression function. BSV: Backward starting value, FSV: Forward starting value, MV: Matching value.

Accordingly, their values are undetermined, these bytes cannot be evaluated until a partial match is found. Same rationale applies to the backward chunk and the blue bytes. Grey bytes are constants which are either given (compression function output) or chosen (chaining value and constants in the initial structure).

In the initial structure, we try to balance the degrees of freedom in each direction and the number of known bytes at the end of each chunk. The degrees of freedom in both directions should produce candidate pairs at the matching point to satisfy the matching probability. More precisely, to minimize

the complexity, the total degrees of freedom in both chunks must be greater than the matching size. For further clarification, we first explain the idea behind the initial structure. The main point is to choose several bytes as neutral bytes so that the number of output bytes of the L and L^{-1} transformations at the start of each chunk that are constant or relatively constant is maximized. A relatively constant byte is a byte whose value is affected by the degrees of freedom in one execution direction but remains constant from the opposite execution perspective. The initial structure for the 5-round MitM preimage attack on the compression function of Streebog is shown in Figure 5.3. We start by randomly choosing

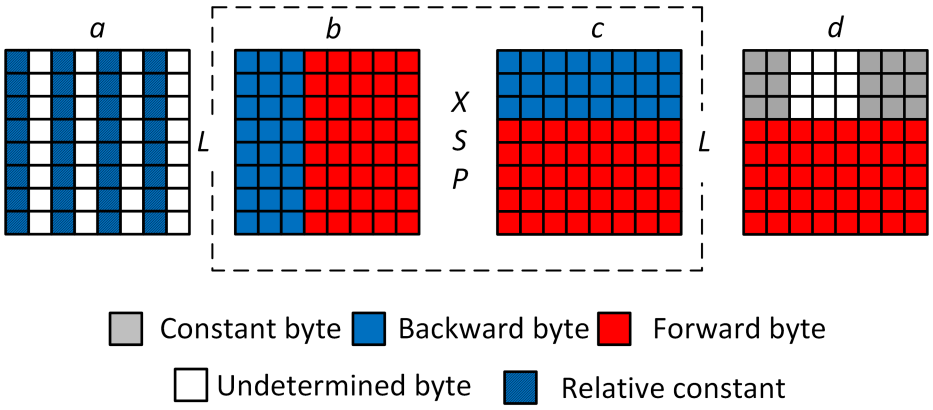

Figure 5.3: Initial structure for the 5-round attack on the Streebog compression function.

the five constant bytes in d[row 0] and then determine the values of blue bytes in c[row 0] so that after applying L on c[row 0], we maintain the chosen five constants. Since we need five constant bytes in d[row 0], we only need five free variables in c[row 0] to solve a system of five equations when the other three bytes are fixed. Accordingly, for any of the first three rows in state c, we can randomly choose any three blue bytes and compute the remaining five so that the output of L maintains the previously chosen five constants at d[row 0]. To this end, we have nine free blue bytes (three for each row in state c). Thus the backward degrees of freedom is 2^{72} which means that we can start the backward execution by 2^{72} different starting values and hence 2^{72} different output values at the matching point M_4^L. Similarly, we choose 32 constants in state a and for each row in state b we randomly choose one red byte and compute the other four bytes such that, after the L^{-1} transformation, we get the predetermined constants at each row in a. However, the value of the four shaded blue bytes in each row of state a depends also on the three blue bytes in the rows of state b. We call these bytes relative constants because their final values cannot be determined until the backward execution starts and these

values are different for each execution iteration. Specifically, their final values are the predetermined constants XORed with the corresponding blue bytes multiplied by the L^{-1} coefficients. In the sequel, we have eight free bytes (one for each row in b) which means 2^{64} forward degrees of freedom to start the forward execution and hence 2^{64} different input values to the matching point M_4^P.

At the matching point, we match results at M_4^P from the forward chunk with the values at M_4^L from the backward chunk through the L transformation. As depicted in Figure 5.2 at the matching point, five bytes are known from the forward computation and four bytes are known from the backward computation in each row. As a result, we can form four linear equations using three unknowns and match the resulting forward and backward values through the remaining equation. More precisely, we use the following equation to compute a given output row y through the linear transformation L given an input row x.

$$\begin{bmatrix} x_7 & x_6 & \overline{x_5} & \overline{x_4} & \overline{x_3} & x_2 & x_1 & x_0 \end{bmatrix} \begin{bmatrix} l_{0,7} & l_{0,6} & l_{0,5} & l_{0,4} & l_{0,3} & l_{0,2} & l_{0,1} & l_{0,0} \\ l_{1,7} & l_{1,6} & l_{1,5} & l_{1,4} & l_{1,3} & l_{1,2} & l_{1,1} & l_{1,0} \\ l_{2,7} & l_{2,6} & l_{2,5} & l_{2,4} & l_{2,3} & l_{2,2} & l_{2,1} & l_{2,0} \\ l_{3,7} & l_{3,6} & l_{3,5} & l_{3,4} & l_{3,3} & l_{3,2} & l_{3,1} & l_{3,0} \\ l_{4,7} & l_{4,6} & l_{4,5} & l_{4,4} & l_{4,3} & l_{4,2} & l_{4,1} & l_{4,0} \\ l_{5,7} & l_{5,6} & l_{5,5} & l_{5,4} & l_{5,3} & l_{5,2} & l_{5,1} & l_{5,0} \\ l_{6,7} & l_{6,6} & l_{6,5} & l_{6,4} & l_{6,3} & l_{6,2} & l_{6,1} & l_{6,0} \\ l_{7,7} & l_{7,6} & l_{7,5} & l_{7,4} & l_{7,3} & l_{7,2} & l_{7,1} & l_{7,0} \end{bmatrix} = \begin{bmatrix} y_7 & \overline{y_6} & y_5 & \overline{y_4} & y_3 & \overline{y_2} & y_1 & \overline{y_0} \end{bmatrix}$$

In the above equation, the overline denotes the unknown bytes at a given row. More precisely, the input contains the unknown bytes x_5, x_4, and x_3 and the corresponding output contains the known bytes y_7, y_5, y_3, and y_1. Accordingly, given the $GF(2^8)$ equivalent of the Streebog binary matrix [79], we can form the following equations:

$$y_7 = t_7^{in} \oplus x_5 \cdot l_{2,7} \oplus x_4 \cdot l_{3,7} \oplus x_3 \cdot l_{4,7} \tag{5.1}$$

$$y_5 = t_5^{in} \oplus x_5 \cdot l_{2,5} \oplus x_4 \cdot l_{3,5} \oplus x_3 \cdot l_{4,5} \tag{5.2}$$

$$y_3 = t_3^{in} \oplus x_5 \cdot l_{2,3} \oplus x_4 \cdot l_{3,3} \oplus x_3 \cdot l_{4,3} \tag{5.3}$$

$$y_1 = t_1^{in} \oplus x_5 \cdot l_{2,1} \oplus x_4 \cdot l_{3,1} \oplus x_3 \cdot l_{4,1}, \tag{5.4}$$

where t_i^{in} is the total of the known input bytes in the i^{th} row multiplied by their corresponding matrix coefficients. To this end, we calculate x_5, x_4, and x_3 from equations 1, 2, and 3 and substitute their values in equation 4. Consequently, the two sides of equation 4 are all known from both input and output directions. Hence, the matching size per row is one byte and hence the matching probability for the whole state is 2^{-64}. The choice of the number forward and backward values directly affects the matching probability as their number determines the number of red and blue bytes at a given row at the matching point. If the number of blue and red bytes are not properly chosen at the initial structure, one might have no value to match at the matching point. In other words, we cannot have a MitM matching value if the total number of red and blue bytes in a given row at the matching point is less than or equal to eight. The attack can be summarized as follows:

1. Randomly choose the chaining value and the constants at the initial structure.

2. For each forward starting value fw_i in the 2^{64} forward starting values at M_2, compute the forward matching value fm_i at M_4^P and store (fw_i, fm_i) in a lookup table T.

3. For each backward starting value bw_j in the 2^{72} backward starting values in M_2^P compute the backward matching value bm_j at M_4^L and check if there exists an $fm_i = bm_j$ in T. If found, then a partial match exists and the full match should be checked using the matched starting points fw_i and bw_i. If a full match exists, then output the chaining value and the message M_0, else go to step 1.

The complexity of the MitM preimage attack is given by $2^n(2^{-r} + 2^{-b} + 2^{-m})$, where n is the state size and r, b, and m are the forward, backward, and matching bit sizes, respectively [150]. The choice of these parameters should minimize the complexity and this can be achieved by keeping r, b and m, as close as possible. In the chunk separation shown in Figure 5.2, $r = 64$, $b = 72$, and $m = 64$. To further explain the complexity of the attack, we consider the attack procedure. After step 2, we have 2^{64} forward matching values and we need 2^{64} memory to store them. At the end of step 3, we have 2^{72} backward matching values. Accordingly, we get $2^{64+72} = 2^{136}$ partial matching candidate pairs. Since the probability of a partial match is 2^{-64}, we expect 2^{72} partially matching pairs. The probability that a partial match results in a full match is $2^{64-512} = 2^{-448}$. Consequently, the expected number of

fully matching pairs is 2^{-376}. Thus we need to repeat the attack 2^{376} times to get a fully matching pair. The time complexity for one repetition of the attack is 2^{64} for the forward computation, 2^{72} for the backward computation, and 2^{72} to check that partially matching pairs fully match. Consequently, the overall complexity of the attack is $2^{376}(2^{64} + 2^{72} + 2^{72}) \approx 2^{448}$ time and 2^{64} memory

5.4 Extending the Attack to 6 Rounds

The previous 5-round attack cannot be extended to 6-rounds because at the end of each chunk execution the state has undetermined bytes at each row. Consequently, applying the linear transformation L to such state results in a fully undetermined state and no matching can be achieved. A guess and determine approach [132] can be used in one direction to guess the undetermined bytes in some rows. Thus we have some known state rows after the linear transformation L. The proposed chunk separation for the 6-round MitM attack is shown in Figure 5.4. In order to be able extend the attack by one extra round, we guess the twelve undetermined bytes (yellow bytes) in state M_4^P. As a result, we can reach state M_5^P with four determined columns where matching takes place.

Our choice of the separation and guessed parameters is based on our analysis of the attack complexity and enumerating several values. Our main objective is to maximize the attack probability by carefully selecting the forward, backward, and guessed bit values. We aim to maximize the number of forward bits and keep the backward and the matching number of bits larger than the number of guessed bits and as close as possible. For our attack, the chosen forward, backward, and guessed bit sizes are 16, 128, and 96, respectively. Setting these parameters fixes the matching bit size which is equal to 128. In what follows, we give the attack procedure and complexity based on the above chosen parameters:

1. Randomly choose the chaining value and the constants the initial structure.

2. For each forward starting value fw_i and guessed value g_i in the 2^{16} forward starting values and the 2^{96} guessed values, compute the forward matching value fm_i at M_5^P and store (fw_i, g_i, fm_i) in a lookup table T.

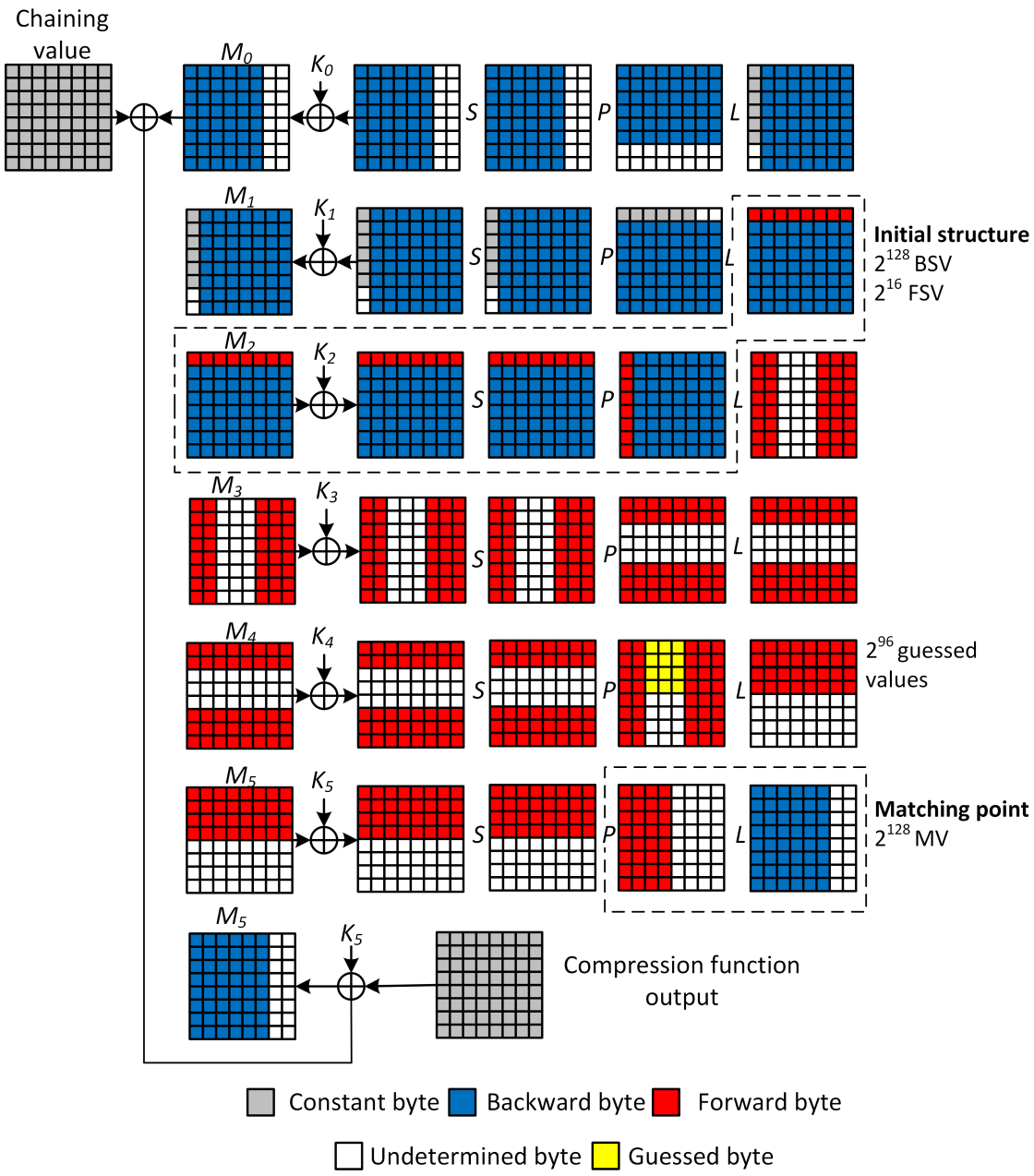

Figure 5.4: Chunk separation for a 6-round MitM pseudo preimage attack on Streebog compression function. BSV: Backward starting value, FSV: Forward starting value, MV: Matching value

3. For each backward starting value bw_j in the 2^{128} backward starting values, compute the backward matching value bm_j at M_5^L and check if there exists an $fm_i = bm_j$ in T. If found, then a partial match exists and the full match should be checked using the matched forward, guessed, and backwards values fw_i, g_i, and bw_i. If a full match exists, then output the chaining value and the message M_0, else go to step 1.

After step 2, we have $2^{16+96} = 2^{112}$ forward matching values which need 2^{112} memory for the look up table. At the end of step 3, we have 2^{128} backward matching values. Accordingly, we get $2^{112+128} = 2^{240}$ partial matching candidate pairs. Since the probability of a partial match is 2^{-128} and the probability of a correct guess is 2^{-96}, we expect $2^{240-128-96} = 2^{16}$ correctly guessed partially matching pairs. The probability that a partial match is a full match is 2^{-384}. Consequently, the expected number of fully matching pairs is 2^{-368} and hence we need to repeat the attack 2^{368} times to get a full match. The time complexity for one repetition is 2^{112} for the forward computation, 2^{128} for the backward computation, and 2^{16} to check that partially matching pairs fully match. The overall complexity of the attack is $2^{368}(2^{112} + 2^{128} + 2^{16}) \approx 2^{496}$ time and 2^{112} memory.

5.5 Preimage of the Streebog Hash Function

In this section, we show how the previously presented pseudo preimage attacks on the Streebog compression function can be utilized to produce preimages for the whole hash function. Streebog has a finalization step which is the last compression function call in the hash function. In this step, the compression function operates on the modular addition of the previously processed message blocks. At first glance, this may seem to limit the ability of turning a pseudo preimage of the compression function to a hash function preimage because inverting the last compression function call returns the sum of the message blocks and thus constraints their values. However, a preimage of the hash function can be found when we consider a large set of long messages that produce different sums and a set pseudo preimage attacks on the last compression function call. Hence, another MitM attack can be performed on both sets to find the message that corresponds to the retrieved sum [105]. As depicted in Figure 5.5, the attack is divided into four stages:

1. Given the hash function output $H(M)$, we produce 2^p pseudo preimages for the last compression function call. The output of this step is 2^p pairs of the last chaining value and the message sum $(H_{515}, \sum_o)$. We store these results in a table T.

2. In this stage, we construct a large set of equal length messages such that all of them collide at H_{512}. This structure is called a multicollision of length 512 [76]. More precisely, a multicollision

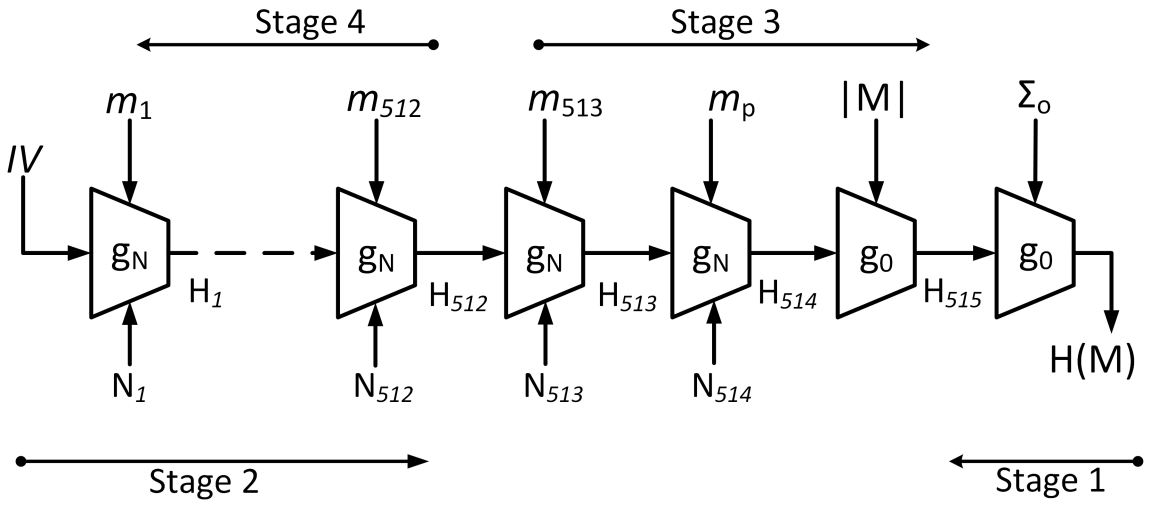

Figure 5.5: Preimage attack on the Streebog hash function.

of length t is a set of 2^t messages where each message consists of exactly t block and every application of the compression function results in the same chaining value. Consequently, all the 2^t messages lead to the same H_t value. Building a multicollision of length t is done with time complexity of $t \cdot 2^{n/2}$ and memory complexity of $t \cdot 2 \cdot n$ to store t 2-message blocks, where n is the state size. In our case, we build 2^{512} multicollision, i.e., $M_i = m_1^j \| m_2^j \| ... \| m_{512}^j$, where $i \in \{1, .., 2^{512}\}$ and $j \in \{1, 2\}$ such that all the $M_i's$ lead to the same H_{512}. To this end, we have 2^{512} different massages stored in $512 \cdot 2 \cdot 512 = 2^{19}$ memory and hence 2^{512} candidate sums $\sum_{M_i}$.

3. At this point, we try to connect the results of stages 1 and 2 using the freedom of choosing m_{513}. Specifically, since we are using messages of 513 complete blocks, then both the padding block m_p and the length block $|M|$ are known constants. We also have one known value of H_{512} produced from the previous stage. In the sequel, we randomly choose m^*_{513}, compute H^*_{515} and check if it exists in T. As T contains 2^p entries, it is expected to find a match after 2^{512-p} evaluations of the following three compression function calls:

$$H_{513} = g_N(H_{512}, m^*_{513}, N_{513})$$

$$H_{514} = g_N(H_{513}, m_p, N_{514})$$

$$H^*_{515} = g_0(H_{514}, |M|)$$

Once a matching H_{515} value is found in T, the corresponding $\sum_o$ is fixed as well. Hence the desired sum at the output of the multicollision $\sum_{M_i}$ is equal to $\sum_o - m_p - m_{513}$.

4. At the last stage of the attack, we try to find a message M_i out of the 2^{512} messages generated in stage 2 that has a sum equal to the sum $\sum_{M_i}$ acquired at the previous stage. This can be achieved by a meet in the middle attack. More precisely, we first calculate all the 2^{256} sums of the first half of all the 2^{256} messages $\sum_{M_1} = m_1^j + m_2^j + ... + m_{256}^j$ and we store them in a table. Afterwards, for each second half message we compute the sum $\sum_{M_2} = m_{266}^j + m_{267}^j + ... + m_{512}^j$ and check if $\sum_{M_i} - \sum_{M_2}$ is in the table. It is expected to find a match after 2^{256} checks. Once a match is found, the concatenation of the two message halves that correspond to the matching sums and m_{513} is the preimage of the given $H(M)$.

The time complexity of the attack is evaluated as follows: we need $2^P \times$ (complexity of pseudo preimage attack) in stage 1, 512×2^{256} to build the multicollision at stage 2, 2^{512-p} evaluations of three compression function calls at stage 3, and finally 2^{256} for the MitM attack in stage 4. The memory complexity for the four stages is as follows: 2^p 2-states to store the pseudo preimages in stage 1, 512 2-message blocks for the multicollision, and 2^{256} for the MitM table in stage 4. Since the time complexity is highly influenced by p, so we have chosen $p = 32$ for the 5-round attack and $p = 8$ for the 6-round attack to obtain the maximum gain. Accordingly, preimages for 5-round Streebog hash function can be produced with a time complexity of $2^{32+448} + 2^{9+256} + 2^{512-32} \times 3 + 2^{256} \approx 2^{481}$. The time complexity for the 6-round attack is $2^{8+496} + 2^{9+256} + 2^{512-8} \times 3 + 2^{256} \approx 2^{505}$, both attacks have a similar memory complexity of 2^{256} dominated by the MitM attack in stage 4.

5.6 Conclusion

In this chapter, we have analyzed Streebog and its compression function with respect to preimage attacks. We have shown that with a carefully balanced chunk separation, pseudo preimages for the 5-round reduced compression function are generated with time complexity of 2^{448} and memory complexity of 2^{64}. Additionally, we have adopted a guess and determine technique to obtain a 6-round chunk separation that maximizes the forward degrees of freedom and balances the backward and the

guess bit sizes. As a result, we were able to extend the 5-round attack by one more round with time complexity of 2^{496} and memory complexity of 2^{112}. Finally, using 2^{512} multicollision and another MitM attack, the compression function pseudo preimage attacks are used to produce 5 and 6-round hash function preimages with time complexity of 2^{481} and 2^{505}, respectively. The two preimage attacks have equal memory complexity of 2^{256}. Interestingly, if one considers long preimages (1024 blocks), the time complexity of the attack can be further reduced by removing the last stage MitM procedure [99]. Specifically, instead of considering 2^{512} multicollision, one can adopt a variant of the multicollision attack [58] that deals with the checksum and considers a preimage of the last compression function call. Accordingly, a 1024 block preimage message for the 6-round reduced Streebog can be generated in 2^{496} time and 2^{112} memory.

It should be noted that the Streebog compression function key whitening round K_N enhances its resistance to certain attacks that require similar diffusion of the executions of both the message and the chaining value. The guess and determine approach is more effective in reducing the complexity when similar chunk separation is performed on the key of the internal cipher to provide additional starting values in both directions [132]. However, key separation cannot be achieved because Streebog has an initial nonlinear whitening round that deviates the chaining value (key) from the message by one round. Hence, even if we were able to start from the middle and separate the chaining value execution, we lose all information when we get to the input chaining value because of the wide trail effect.

Chapter 6

Malicious Streebog

In this chapter, we investigate the new Russian cryptographic hashing standard in the context of malicious hashing and present a practical collision for a malicious version of the full hash function. In particular, we apply the rebound attack to find three solutions for three different differential paths for four rounds. Then, using the freedom of the round constants we connect them to obtain a collision for the twelve rounds of the compression function. Additionally, and due to the simple processing of the counter, we bypass the barrier of the checksum finalization step and transfer the compression function collision to the hash function output with no additional cost. The presented attack has a practical complexity and is verified by an example. While the results presented in this chapter may not have a direct impact on the security of the current Streebog hash function, they have raised concerns within the cryptographic community and presented an urge for the designers of the new standards to publish the origin of the used parameters and the rational behind their choices. Such concerns were later addressed by the designers of Streebog in a published paper [129], available on the Russian standardization agency website, explaining the origin of the adopted parameters.

6.1 Introduction

Research on malicious cryptographic primitives has always been thought of as the work of intelligence agencies. The belief that governmental spy agencies work hard to incorporate backdoors in their primitives, which enables the efficient manipulation of certain security properties, has always

been lurking in the cryptographic community. This belief was further strengthened last year after Edward Snowden exposed the existence of the NSA's Bullrun decryption project [147]. Leaked documents have shown that the NSA has deliberately inserted a backdoor in the standardized pseudorandom number generator Dual_EC_DRBG [148]. This backdoor provides the knowledge of the internal state of the generator and accordingly its subsequent outputs. Additionally, it is also speculated that NSA paid RSA Security $10 million in a secret deal to use Dual_EC_DRBG as the default pseudorandom number generator in the RSA BSAFE cryptography library [148]. With Dual_EC_DRBG being recommended by NIST at that time, these revelations have raised suspicions with respect to the NIST standards being manipulated by the NSA, particularly, after voices from the cryptographic community began suggesting the possibility of the NSA compromising the NIST's recommended elliptic curve constants [133].

Only few papers have been peer reviewed in public venues in the area of malicious cryptography. Young and Yung were among the first to address the topic of malicious cryptography through their cryptovirology project [151]. Later Rijmen and Preneel proposed malicious versions of CAST and LOKI by hiding linear relations in the used Sboxes [127]. Work related to malicious ciphers, implementations and pseudorandom generators includes [22, 27, 54, 119, 120]. Although most of the previous work focused on ciphers, just recently the concept of malicious hashing have been introduced in [5, 21]. Specifically, Albertini *et al.* proposed a malicious version of SHA-1 by which collisions can be produced in an efficient way. They have used the freedom of the round constants to satisfy a given differential path and generate one block message collisions.

Since coming to effect in 2013, Streebog has been standardized by IETF as RFC 6896 [72]. However, unlike the specifications of other standardized hash functions, the reference of the new GOST standard [2] gives no information about how or why the parameters of the function (e.g., round constants, matrix constants, and the number of rounds) have been chosen. This fact opens the door to our analysis, which makes use of exactly two parameters: the heavily random looking independent constants and the number of rounds, to present practical collisions for a malicious version of Streebog.

In this chapter, we investigate a malicious version of Streebog. We exploit the randomness of the independent round constants and take advantage of the number of rounds of the compression function to efficiently generate collisions for the compression function. More precisely, we first employ

the rebound attack technique proposed in [93] to find three pairs of messages and keys that satisfy a specific three 4-round differential paths independently. In the sequel, we use the freedom of five out of the twelve round constants to connect the three obtained solutions and obtain collisions for the twelve round compression function. Finally, we tune the last constant of the compression function to adjust its output after the feedforward to cancel the effect of the counter, N_{i-1}, addition of the following compression function call, and append another identical colliding message pair. Hence, we generate a two block messages 2^2 multicollision structure where two of them have the same modular sum and thus a collision at the output of the hash function. While previous work [21] stated that compression function collisions are not sufficient to generate hash function collision in constructions that incorporate checksum, our results prove that this is not the case for Streebog. Table 6.1 provides the six new constants used in our malicious version of Streebog. An example of the two block message collision along with its corresponding digest is provided in Table 6.3.

The rest of the chapter is organized as follows. In the next section, we provide a detailed description of the used approach, the malicious compression function attack and its corresponding complexity. In Section 6.3, we show how collisions of the malicious hash function are generated using the attack presented in Section 6.2. Finally, the chapter is concluded and a short discussion is provided in Section 6.4.

6.2 Malicious compression function collision

In Latincrypt 2014, Kölbl and Rechberger presented a practical method to find semi free-start collision for a 4-round AES-based compression function [93]. More precisely, they have proposed a way to first find a specific differential path for $1 \xrightarrow{r_i} 8 \xrightarrow{r_i+1} 64 \xrightarrow{r_{i+2}} 8 \xrightarrow{r_{i+3}} 1$ transition, then use the freedom in the key to find two messages that follow the given path. They have implemented their approach on Streebog and presented a semi free-start collision for the 4-round reduced compression function. In what follows, we show how we adapt their approach to generate collisions for a malicious version of the full Streebog compression function.

Our approach makes use of the heavily random looking independent round constants and the twelve rounds of the compression function. In fact, the specific number of rounds (12) used in Streebog enables us to find three independent solutions for the commonly known $1 \longrightarrow 8 \longrightarrow 64 \longrightarrow 8 \longrightarrow 1$ four round differential path and then, by changing five constants, we can successfully connect them and generate a collision. Our attack starts by finding the first solution which is a pair of messages and

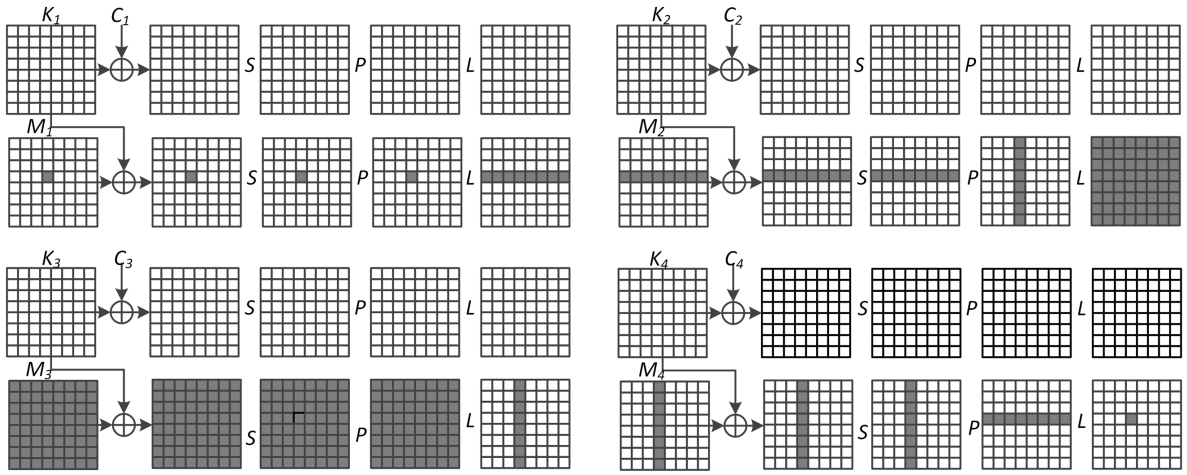

Figure 6.1: The first truncated differential path.

a key that follow the given differential path shown in Figure 6.1. In doing so, we employ the approach proposed in [93] which is composed of two procedures and is briefly described as follows:

Building the differential characteristic In this procedure, one determines the exact differential transitions of the above truncated differential trail as follows:

1. Choose a random difference at $M_4^L[3,3]$ and propagate it backward until the full active state M_3^S.
2. For each byte difference in M_3^S, save a set of all possible input differences.
3. Create a table T_L of all possible 255 byte difference values d_3 (candidates for $M_2^P[*,3]$) and their corresponding 8 byte difference values $L(d_3)$ (candidates for $M_3^X[\text{row } 3]$). These values are the result of applying the linear transformation L to a difference at column 3.
4. For each row of M_3^X, check if there is a possible match with the rows in T_L.
5. To achieve the transition from one active byte in $M_1^P[*,3]$ to eight active bytes in $M_2^X[\text{row } 3]$, steps 2 and 4 must be repeated for only one row between states M_2^S and M_1^P.

According to the Streebog Sbox differential distribution properties, finding the differential characteristic has a complexity of $\approx 2^{20}$ [93].

6.2.1 Finding a solution for the differential path

Once we have found a characteristic, we now need to find a message pair that follows it. This can be done by performing the following steps:

1. Set the message state at M_3^X with a solution that satisfies the full active state differentials from the above procedure.

2. Use K_3[col 3] to satisfy the solutions of the Sbox differentials at M_2^P[col 3]. Also use K_3[row 3] to satisfy the solutions of the Sbox differentials at M_4^X[col 3].

Since there is one byte, $K_3[3, 3]$, shared between the two solutions, one needs to repeat the above procedure 2^8 times. For more details on the specifics of the used technique, the reader is referred to [93].

6.2.2 Our proposed technique for finding collisions of the malicious compression function

To this end, we have found a solution to the first differential path with a key input different from that is produced by the standard IV. This solution gives us a specific input and output differences Δ_{in}^1 and Δ_{out}^1 at $M_1[3, 3]$ and $M_4^L[3, 3]$, respectively. In the sequel, we restart the above two procedures to search for the second differential characteristic and its solution such that this second search covers rounds five to eight and has an input difference Δ_{in}^2 at $M_5[3, 3]$ equals to the output difference Δ_{out}^1 of the first path. Since we restrict the input difference of the second path to a specific value, the complexity of the second procedure of our search is increased by a factor of 2^8. However, the overall search complexity is still dominated by the first procedure which is about 2^{20}. Finally, we search for the third and last differential path and its solution which covers rounds nine to twelve. For this path, we have to restrict its input difference Δ_{in}^3 at $M_9[3, 3]$ to be equal Δ_{out}^2 at $M_8^L[3, 3]$ and its output difference Δ_{out}^3 at $M_{12}^L[3, 3]$ to be equal Δ_{in}^1 at $M_1[3, 3]$, so that the latter cancels out after the

feedforward. Figure 6.2 depicts an overview of our technique. Similarly colored input and output differences in the states which result from the three solutions are chosen to be equal. The constants that are evaluated to connect the three solutions, and to get the desired IV and compression function output values are also shown in the figure.

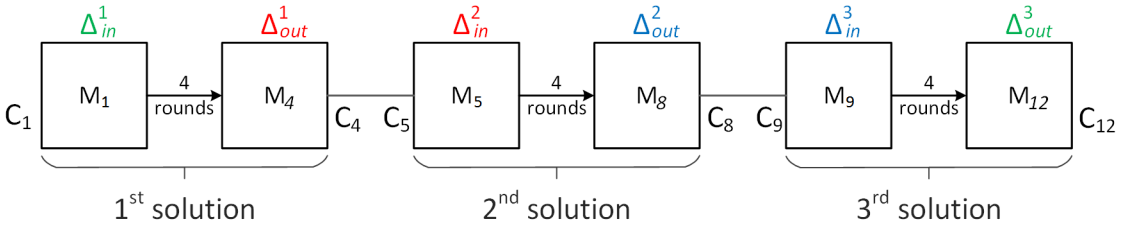

Figure 6.2: Our approach for finding collision for the full round compression function.

6.2.3 Connecting the three solutions

Now that we have the three solutions, we can start tuning specific round constants to connect them. We first work on the first solution's key output K_1, which is different than that generated by the standard IV. To solve this problem, we fix the new $C_1 = LPS(IV) \oplus (K_1^X)$. By doing this, we guarantee that the resulting new key satisfies the first differential path. Thus, the new colliding messages are $m = (M_1^X \oplus LPS(IV)$ and $m' = m \oplus \Delta_{in}^1$.

To connect the first and second solutions, we have to change K_5. However, altering K_5 affects both K_4 and K_6, which are restricted by the solutions of the first and second paths, respectively. In order to cancel the propagation of alteration to the latter two round keys, we compute the new two constants C_5 and C_4 as follows:

$$\begin{aligned} K_5 &= M_4^L \oplus M_5^X, \\ C_5 &= K_5 \oplus K_5^X, \\ C_4 &= S^{-1} \oplus P \oplus L^{-1}(K_5) \oplus K_4, \end{aligned}$$

where M_4^L and K_4 are solutions of the first path, while M_5^X and K_5^X are solutions of the second path. To connect the second and third paths, we perform the same procedure to compute the new C_8 and

C_9. Having all the new five constants in place, Table 6.3, gives an example of a colliding message pair which has the same compression function output using $IV = 0$ and $N_{i-1} = 0$.

6.3 Collision attack on the full malicious Streebog

While previous work [21] speculated that collisions of the compression function cannot be reflected at the output of the hash function when employing a checksum finalization step, in this section, we show how to turn the previous compression function collision to a hash function collision. On top of the modular checksum finalization step, Streebog incorporates a counter N_{i-1} with each compression function call. However, N_{i-1} is mixed with the chaining value with a simple XOR operation. It should be noted that once the constants of the compression function are fixed to some values, they remain the same for all successive executions of the compression function. Accordingly, it is infeasible to search for a different collision with the same constants. Our approach replicates the first collision

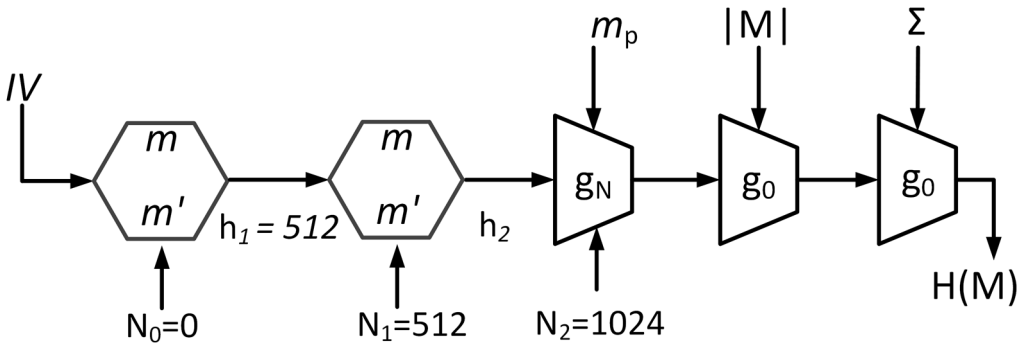

Figure 6.3: Malicious Streebog collision.

two times, thus creating a 2-block multicollision structure with the same h_2 input to the padding call $g_N(h_2, m_p, N_2)$ as depicted in Figure 6.3. By doing this, it is guaranteed that four messages collide at h_2, and only two of them collide at the output of the hash function. Namely, those two that have the same modular checksum which are $M = m||m'$ and $M' = m'||m$. However, using the same collision twice implies that the second collision should have a chaining input h_1 equal to that of the first collision which is $IV = 0$. For this, we compute a new C_{12} to enforce the output of the first collision h_1 to be 512, which is equal to the value of N_1 used in the following compression function call. The desired

value of C_{12} is evaluated as follows:

$$C_{12} = S^{-1} \circ P \circ L^{-1}(M_{12}^{L} \oplus M_1 \oplus 512) \oplus K_{12}.$$

To this end, at the input of the second compression function call h_1 cancels the effect of N_1 and the second colliding message pair has a chaining input internal state equal to that of the IV which is used at the first call.

6.4 Conclusion

In this chapter, we have investigated a malicious version of Streebog. We took advantage of the heavily random looking constants and the number of rounds of the compression function to present a 2-block message pair with the same digest. Our approach first searches for three solutions for three different 4-round differential paths, and then uses the freedom of five constants to connect them to produce a compression function collision. Finally, we employed the freedom in the last constant used in the round key generation to cancel the effect of the counter used in the second compression function call. Hence, we were able to append a second similar message pair, thus creating a 2^2 multicollision structure where only two of them have the same modular checksum and accordingly the same digest.

It should be noted that these results have no impact on the security of the original standard. Additionally, this new set of constants does not provide collision for GOST-256 as it uses a different IV. However, they are interesting in the light of the absence of the source of the used parameters of the standard. Our results also show one of the first examples of compression function collisions being sufficient to generate hash function collisions. It is interesting to mention that, due to the versatility of the used differential path where the one byte difference can virtually be anywhere in the state, we get the freedom to satisfy the magic number as well as other constraints that are needed to produce meaningful collisions for some specific file formats (cf. section 4 in [5]). In other words, as the difference is sparse and the complexity of the attack is upper bounded by 2^{20}, if one requires to find two messages that start with a specific byte value, then we need to repeat the first path search 256 times which raises the time complexity of the attack by a factor of 2^8. As a future direction, one may

investigate the applicability of the attack if the number of rounds is not a multiple of four. Also, one might try searching for a malicious adaptation that holds for the two versions of the hash function simultaneously. In response to our results, the Russian standardization body TC26 has published a note [129] explaining the origin of the employed constants.

C_1								C_4								C_5							
3b	7b	5d	ca	f1	e4	23	2f	7b	51	2e	eb	f5	f6	ab	f4	9b	b1	e8	b9	00	2f	6d	75
de	dd	27	78	d6	9b	fe	93	42	52	38	55	1b	14	c2	9d	96	d7	e3	12	a2	5c	66	9c
f7	9f	94	dd	27	02	f3	a2	6e	5b	20	23	c9	b9	8f	3d	7e	aa	0e	bf	dd	0e	04	88
4b	8e	ad	06	8d	6f	3a	fd	a5	cc	0b	e3	78	9b	9d	52	f7	30	67	e2	8c	b5	37	1e
fa	da	e2	5c	b1	2a	0f	3a	bc	30	cc	de	99	39	07	69	6b	1c	1b	28	09	6d	0d	78
0f	7d	0d	18	ba	f6	0c	e9	cb	69	60	cf	89	c9	20	cd	4c	fa	57	06	9e	da	f6	4f
27	b7	42	a3	7d	68	cd	64	e7	e6	7c	81	ef	d7	97	6e	1d	20	22	e9	ce	7e	54	3f
5b	41	e8	61	e2	cb	9d	a6	71	ac	16	c5	bf	cc	b9	c1	35	0c	56	b4	d8	a5	01	b7
C_8								C_9								C_{12}							
02	e5	04	18	6c	11	2d	01	f9	53	2e	c1	78	84	d2	6e	a3	23	32	b5	81	5e	1b	85
02	f1	f2	49	5d	d0	aa	7b	17	ae	c9	5a	a4	44	4c	8d	f4	67	4d	bc	c3	77	fd	7f
98	4c	e1	b8	08	fd	0f	60	21	8b	63	a4	c1	2a	32	b8	f8	a1	db	b5	e3	69	99	41
46	79	75	f7	37	5d	a1	8c	41	2c	9a	d0	71	20	55	30	eb	15	09	84	de	8d	22	ea
3c	b5	83	ac	90	27	38	30	fb	71	99	26	59	a8	6f	4f	9d	e6	44	d5	fd	40	7b	5d
25	af	e8	05	d1	bd	e3	34	8e	37	7a	c5	06	ad	7f	93	d1	32	45	08	e9	3d	3f	51
ea	eb	50	bf	be	39	32	9a	50	0b	be	70	04	4b	9d	5c	2a	36	ae	cc	53	97	0f	fc
61	1a	1a	22	e1	0d	ff	58	d7	aa	2c	27	6e	cd	41	01	41	a7	84	f3	44	91	24	3e

Table 6.1: The six new constants.

C_2								C_3								C_6							
6f	a3	b5	8a	a9	9d	2f	1a	f5	74	dc	ac	2b	ce	2f	c7	ae	4f	ae	ae	1d	3a	d3	d9
4f	e3	9d	46	0f	70	b5	d7	0a	39	fc	28	6a	3d	84	35	6f	a4	c3	3b	7a	30	39	c0
f3	fe	ea	72	0a	23	2b	98	06	f1	5e	5f	52	9c	1f	8b	2d	66	c4	f9	51	42	a4	6c
61	d5	5e	0f	16	b5	01	31	f2	ea	75	14	b1	29	7b	7b	18	7f	9a	b4	9a	f0	8e	c6
9a	b5	17	6b	12	d6	99	58	d3	e2	0f	e4	90	35	9e	b1	cf	fa	a6	b7	1c	9a	b7	b4
5c	b5	61	c2	db	0a	a7	ca	c1	c9	3a	37	60	62	db	09	0a	f2	1f	66	c2	be	c6	b6
55	dd	a2	1b	d7	cb	cd	56	c2	b6	f4	43	86	7a	db	31	bf	71	c5	72	36	90	4f	35
e6	79	04	70	21	b1	9b	b7	99	1e	96	f5	0a	ba	0a	b2	fa	68	40	7a	46	64	7d	6e
C_7								C_{10}								C_{11}							
f4	c7	0e	16	ee	aa	c5	ec	ab	be	de	a6	80	05	6f	52	7b	cd	9e	d0	ef	c8	89	fb
51	ac	86	fe	bf	24	09	54	38	2a	e5	48	b2	e4	f3	f3	30	02	c6	cd	63	5a	fe	94
39	9e	c6	c7	e6	bf	87	c9	89	41	e7	1c	ff	8a	78	db	d8	fa	6b	bb	eb	ab	07	61
d3	47	3e	33	19	7a	93	c9	1f	ff	e1	8a	1b	33	61	03	20	01	80	21	14	84	66	79
09	92	ab	c5	2d	82	2c	37	9f	e7	67	02	af	69	33	4b	8a	1d	71	ef	ea	48	b9	ca
06	47	69	83	28	4a	05	04	7a	1e	6c	30	3b	76	52	f4	ef	ba	cd	1d	7d	47	6e	98
35	17	45	4c	a2	3c	4a	f3	36	98	fa	d1	15	3b	b6	c3	de	a2	59	4a	c0	6f	d8	5d
88	86	56	4d	3a	14	d4	93	74	b4	c7	fb	98	45	9c	ed	6b	ca	a4	cd	81	f3	2d	1b

Table 6.2: The six unchanged (original) constants.

m								m'								Δm							
d2	d7	5d	81	b1	63	d8	cc	d2	d7	5d	81	b1	63	d8	cc	00	00	00	00	00	00	00	00
63	16	bb	de	0e	61	85	d6	63	16	bb	de	0e	61	85	d6	00	00	00	00	00	00	00	00
97	89	a3	e6	55	cf	46	e7	97	89	a3	e6	55	cf	46	e7	00	00	00	00	00	00	00	00
37	de	22	19	54	d6	01	95	37	de	22	bb	54	d6	01	95	00	00	00	a2	00	00	00	00
13	44	b8	4d	a3	4d	36	4c	13	44	b8	4d	a3	4d	36	4c	00	00	00	00	00	00	00	00
a3	50	36	27	f3	51	7f	ee	a3	50	36	27	f3	51	7f	ee	00	00	00	00	00	00	00	00
58	23	1d	88	80	1b	09	62	58	23	1d	88	80	1b	09	62	00	00	00	00	00	00	00	00
08	9d	bc	4d	aa	a1	73	2a	08	9d	bc	4d	aa	a1	73	2a	00	00	00	00	00	00	00	00

$$H(m||m') = H(m'||m)$$

94e19a2ad9252ca78d14600c20488ad66de12c72ab3aac19f7bb9e277abe973aea22f1c3fa3be180c6dd212f4b19eefed80fb114c44dfb39ffdb2cfad24c6275

Table 6.3: Example of a 2-block message collision for the malicious Streebog hash function.

Chapter 7

Differential Fault Analysis of Streebog

In this chapter, we present a fault analysis attack on the Streebog hash function. In particular, our attack considers the compression function in the secret key setting where both the input chaining value and the message block are unknown. The adopted fault model is the one in which an attacker is assumed to be able to cause a bit-flip at a random byte in the internal state of the underlying cipher of the compression function. We also consider the case where the position of the faulted byte can be chosen by the attacker. In the sequel, we propose a two-stage approach that recovers the two secret inputs of the compression function using an average number of faults that varies between 338-1640, depending on the assumptions of our employed fault model. Moreover, we show that the attack can be extended to the iterated hash function using a feasible pre-computation stage. Finally, we analyze Streebog in different MAC settings and demonstrate how our attack can be used to recover the secret key of HMAC/NMAC-GOST.

7.1 Introduction

Streebog is expected to be included in standardized cryptographic suites that support its use in the secret key setting. Thus, studying its vulnerability to fault attacks and demonstrating the complexity of the key recovery by an adversary that can manipulate the function's execution are of paramount importance. In this chapter, we present a practical differential fault analysis attack (DFA) on Streebog. The attack considers the compression function when operating with secret inputs which is the default

setting when the function is used in a message authentication code (MAC) scheme. In other words, we consider that both the input chaining value and message block are unknown and that we can only observe the output of the compression function. In the sequel, we propose a two-stage attack using the one-bit fault model where the attacker is able to cause a bit flip at a chosen or random byte in the internal state of the function. Employing a specific property of the Streebog Sbox and by observing several correct and faulty compression function outputs, the first stage of the attack bypasses the final feedforward and retrieves the state of the internal cipher. Since all inputs are unknown, the retrieved state does not allow us to invert the internal cipher of the compression function because its round keys are dependant on the input chaining value which is a secret. Accordingly, in the second stage of the attack, we recover one of the round keys which enables the recovery of both the chaining value and message block of the attacked compression function. To this end, we are restricted to the processing of the last compression function in the iterated hash function as it is the only one which we can observe both its correct and faulty outputs. For that, we employ two precomputed tables which allows us to extend the attack to the whole hash function. Finally, we analyze the GOST hash function in different MAC [25] settings and show how to use our attack to recover the secret MAC key of simple prefix and secret-IV MACs [124], HMAC, and NMAC [25].

The rest of the chapter is organized as follows. In the next section, a brief overview on fault analysis attacks is given. Afterwards, in section 7.3, we provide a detailed description of the used fault model, our two-stage approach, and show how to extend the attack from the compression function to the whole hash function. In section 7.4, we consider Streebog operating in different MAC settings and present the approaches used in the key recovery of simple prefix, secret-IV, HMAC, and NMAC. Simulation results and analysis of the number of required faults for different attack scenarios are given in section 7.5. Finally, the chapter is concluded in section 7.6.

7.2 Fault Analysis

In mathematical attacks, such as differential and linear cryptanalysis, the attacker tries to exploit any weakness in the underlying mathematical structure of the cryptographic primitive. In fault

analysis, which is an implementation dependent attack, the attacker faults the state of the primitive during its computation to deduce information about its secret material. In particular, the attacker applies some kind of physical intervention during the computation of the internal state of the primitive which corrupts random or known bits in the state. Consequently, the attacker observes the correct and the faulty outputs and performs differential fault analysis [29]. During this analysis, the attacker gains non negligible information about the secret material embedded in the hardware by comparing the correct and faulty outputs. Fault injection can be done in many ways which include power glitches, clock pulses, and laser radiation. The reader is referred to [42,136] for more details about the practical experimentation with different methods of fault injection.

Fault analysis was first introduced when Boneh *et al.* showed how the private key of the RSA-CRT-algorithm can be successfully recovered by observing the correct ciphertext and then injecting a fault and acquiring the faulty ciphertext [34]. Later on, Biham and Shamir combined fault analysis with differential cryptanalysis and presented differential fault analysis [29] against DES. Their attack works by observing the difference between the correct and faulty ciphertexts and exploiting this relation to recover the key of DES. DFA attacks have been used for the analysis of the hardware security of many ciphers (e.g., see [23,64,139]). In particular and due to its significance as a standard, AES has received a lot of attention with regards to DFA where some of the works used fault injection in the encryption process [64, 139], and others attacked the key schedule [83]. DFA attacks vary in the number of required faults depending on the employed fault model. Generally, all models assume that the attacker has access to the physical device, and is able to reset the device to the same unknown initial settings as often as needed. Furthermore, different assumptions with respect to the amount of control the attacker has over the position and the Hamming weight of the induced faults are employed.

While most of the DFA work in the literature is targeted towards block and stream ciphers, only few researchers considered hash functions. This fact might seem logical at first glance because ciphers have a secret key input. On the other hand, hash functions are usually analyzed with known inputs. However, lately, DFA attacks have been considered on hash functions with secret inputs, which is the default setting for the hash function when used in a MAC scheme. In general, adapting DFA

attacks against hash functions operating in the secret key setting is somewhat inherently more difficult than adapting it against stream and block ciphers. In fact, unlike block and stream ciphers where one assumes that only the input key material is unknown, when a hash compression function is used in a MAC setting, we consider all its inputs as secrets. Additionally, when a hash function is employed in a MAC scheme, there are usually several applications of the hash function and even a single application of the hash function uses a domain extender with occasionally a complex finalization stage.

Literature related to DFA attacks on hash functions include the analysis of SHACAL [97], which is the internal cipher of the SHA1 compression function. Later, the attack was adapted to deal with the feedforward which masks the output of the internal cipher and both the secret chaining value and message block were retrieved [70]. Afterwards, DFA was used to analyze HAS1-60 [78], and Grøstl [55]. In particular, in the analysis of Grøstl [55], the authors have used the one-bit fault model to invert the truncated output transformation, and to retrieve the input chaining value and message block of its permutation based compression function. In our attack on Streebog, we employ some of the concepts introduced in [55]. In the following section, we give the description of our two-stage attack.

7.3 Differential Fault Analysis Attack on Streebog

Our attack on the Streebog compression function aims to recover the secret input chaining value and message block. We proceed in a two-stage approach. In the first stage, given the compression function output, we recover the internal state of the last round of the internal cipher. Unlike the attack on the permutation based Grøstl, the knowledge of the internal state is not sufficient to recover the secret inputs since Streebog employs an internal cipher with secret round keys additions. Hence, we adopt a second stage for the attack where we use the knowledge of the retrieved state from stage one to successfully recover one of the secret round keys, thus inverting the cipher and acquiring both the secret inputs of the compression function. In what follows, we give the definition of the used fault model and one of the Streebog Sbox properties that we are going to use in our attack.

Fault model: In our attack, we use the one-bit fault model which is used in [55, 64]. For each fault injection, the attacker is assumed to be able to flip one bit in a given byte of the processed state whose position at row r and column c may be known or not. The practicality of this model has been demonstrated in [42], where the authors showed how tuning the laser injection parameters enables them to control with a 100% success rate the fault injection effect on a single bit: 0 to 1 or 1 to 0. Let M be a correctly computed state and M' a faulty state with a fault induced during its computation, then $M' = M \oplus \Delta$ where Δ is the error state with only one non-zero byte. Formally, the employed fault model is defined as follows:

$$\Delta[r, c] = \begin{cases} \delta \in E & \text{for only one byte position,} \\ 0 & \text{otherwise,} \end{cases}$$

where $\Delta[r, c]$ denotes the error at the byte in row r and column c, and the set

$$E = \{0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80\}.$$

For the Streebog Sbox, if x is a random input byte, $\delta x_i \in E$ for $i = 1, .., n$, $n \leq 8$ is a randomly chosen but distinct one bit faults, and

$$\delta y_i = S(x) \oplus S(x \oplus \delta x_i),$$

then x is uniquely identified by the values of δy_i only. In other words, the value of the Sbox input byte x can be recovered by observing n output differences δy_i corresponding to n one-bit distinct input faults. According to our exhaustive simulation, depending on x, the average number of fault insertions δx_i which affect different bits required to identify x varies between 2.071418 - 4.86861, and the overall average is $\overline{n} \approx 2.635$ faults per byte. For the case when fault insertions δx_i are randomly picked, the overall average is $\overline{n} \approx 3.077$ faults per byte. Another observation is that, for all x, there always exist two unique δx_i that would identify x. In what follows, we give the details of the first stage of the attack.

7.3.1 Stage One

In this stage, we recover the message state of the internal cipher of the compression function $g_N(h_{i-1}, m, N)$. We first observe the value of the correct compression function output $h_i = g_N(h_{i-1}, m, N)$. Afterwards and as depicted in Figure 7.1, we induce one-bit fault in a given byte of M_{11}, which is the input to the last round of the cipher, such that, $M'_{11} = M_{11} \oplus \Delta$, and Δ has only one non zero byte at position $[r, c]$. This fault results in a faulty h'_i that differs from the correct h_i state in one row as shown in Figure 7.1.

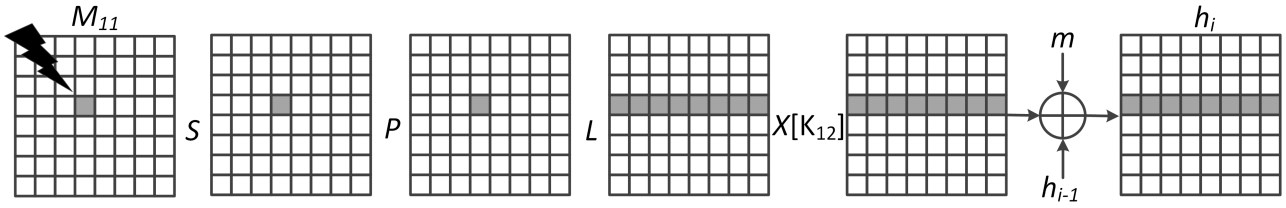

Figure 7.1: Fault injection in the first stage of the attack.

For a fault in a given byte position $M_{11}[r, c]$ we get the following two equations:

$$\begin{aligned} h_i &= (X[k_{12}] \circ L \circ P \circ S(M_{11})) \oplus h_{i-1} \oplus m, \\ h'_i &= (X[k_{12}] \circ L \circ P \circ S(M_{11} \oplus \Delta)) \oplus h_{i-1} \oplus m. \end{aligned}$$

Since, $X, L,$ and P are bijective linear functions, we can propagate the difference at h_i backwards until the state after the Sbox as follows:

$$\begin{aligned} h_i \oplus h'_i &= L \circ P \circ (S(M_{11}) \oplus S(M_{11} \oplus \Delta)), \\ P \circ L^{-1}(h_i \oplus h'_i) &= S(M_{11}) \oplus S(M_{11} \oplus \Delta). \end{aligned}$$

To this end, the difference state at the output of the Sbox of the last round of the internal cipher is given by $\Delta_{out} = S(M_{11}) \oplus S(M_{11} \oplus \Delta)$, where $\Delta_{out} = P \circ L^{-1}(h_i \oplus h'_i)$ and has only one non-zero value at row r and column c. Since, the substitution transformation operates on the state bytes independently, then the knowledge of the difference state Δ_{out} reveals the position $[r, c]$ of the induced fault. Accordingly, if we assume that we have enough faulty compression function outputs h'_i such

that we know enough Δ_{out} states for each byte position in state M_{11}^S, then using the Sbox property presented in the previous section, we can recover the value of the entire state M_{11}.

7.3.2 Stage Two

Although in stage one, we are able to bypass the effect of the feedforward and recover state M_{11} of the internal cipher, we are still not able to invert the compression function and retrieve the secret input chaining value and message block. This is due to the fact that unlike other AES-based hash functions such as Grøstl which employs an internal permutation where known round constant additions are used, the Streebog internal cipher employs round key addition. These round keys are derived from the secret input chaining value and consequently they are not known to the attacker. For that reason, the knowledge of a round state of the compression function is not sufficient to invert it.

Our strategy in this stage is to recover the value of round key k_{11}, which is the key used in the round before the last one. Once we retrieve the value of k_{11}, we invert the key schedule to compute all previous round keys and finally, using the knowledge of the compression function counter N, the secret input chaining value is recovered. The employed approach depends on the knowledge of state M_{11} which we have recovered in the first stage of the attack. To recover the value of k_{11}, we first retrieve the value of state M_{10}, then evaluate $k_{11} = L \circ P \circ S(M_{10}) \oplus M_{11}$. Since, we know the value

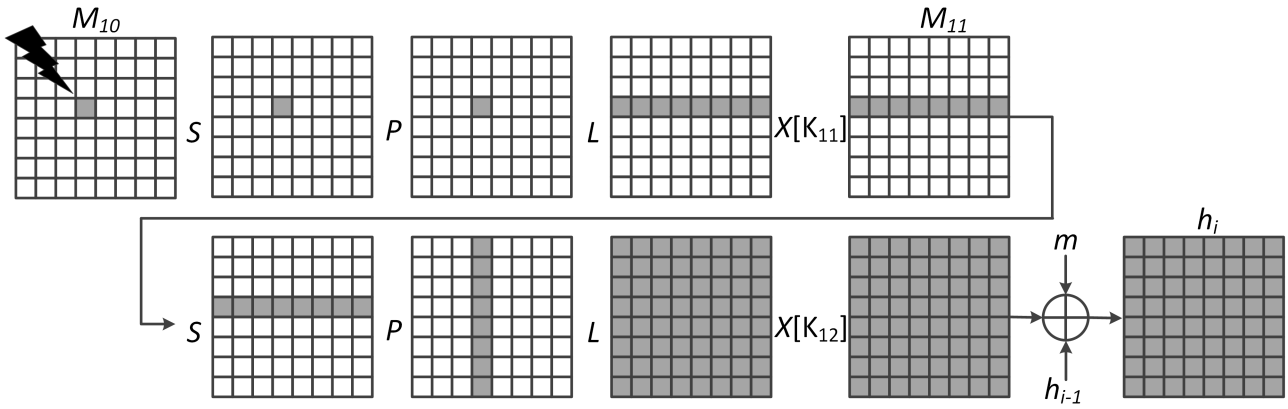

Figure 7.2: Fault injection in the second stage of the attack.

of M_{11}, we can inject one-bit faults in M_{10} and propagate the resulting differences in state h_i back to state M_{11} which is then used as h_i in stage one to recover M_{10}. As depicted in Figure 7.2, we inject

one-faults in M_{10} and acquire the corresponding faulty h'_i which differs from the correct h_i in the whole state. The difference state $h_i \oplus h'_i$ is then propagated backward through the linear transformations until state M^S_{11}. Accordingly, the value of the faulty state M'_{11} is given by:

$$M'_{11} = S^{-1}(S(M_{11}) \oplus (P \circ L^{-1}(h_i \oplus h'_i))).$$

To this end, we get the difference at M_{11} which is then propagated backward to state M^S_{10}. The difference at M^S_{10} is the output difference of the Sbox at the eleventh round corresponding to the fault that we injected at state M_{10}. Consequently, this difference has only one active byte which reveals the byte position of the injected fault. The difference at state M^S_{10} is denoted by Δ_{out} and is given by:

$$\Delta_{out} = P \circ L^{-1}(M_{11} \oplus M'_{11}).$$

Now, if we repeat stage two such that we get enough Δ_{out} values for each of the 64 positions in state M^S_{10}, we can recover the value of state M_{10}. Consequently, the value of k_{11} is computed by the following equation:

$$k_{11} = (L \circ P \circ S(M_{10})) \oplus M_{11}).$$

In the sequel, using k_{11} we invert the key schedule and acquire all the round keys. Then by utilizing the knowledge of the compression function counter N within the hash function, the input chaining value h_{i-1} is recovered. Since, we only observe the output of the last compression function call of the hash function, we always assume that we are processing $g_0(h_{i-1}, \Sigma)$ so that $N = 0$. However, the attack can work on any g_N within the hash function as described in the following subsection. Finally, with the knowledge of the round keys and state M_{10}, we invert the message encryption and recover the input message block m of $g_0(h_{i-1}, \Sigma)$.

7.3.3 Extending the Attack to the Hash Function

The two-stage attack presented in the previous section works on a compression function that one can observe the effect of the induced fault on its output. When Streebog is used in various MAC applications, full hash function application is used and, as depicted in Figure 7.3, one can only observe

the output $H(M)$ of the last compression function call $g_0(h_{t+1}, \Sigma)$ of the hash function. Accordingly,

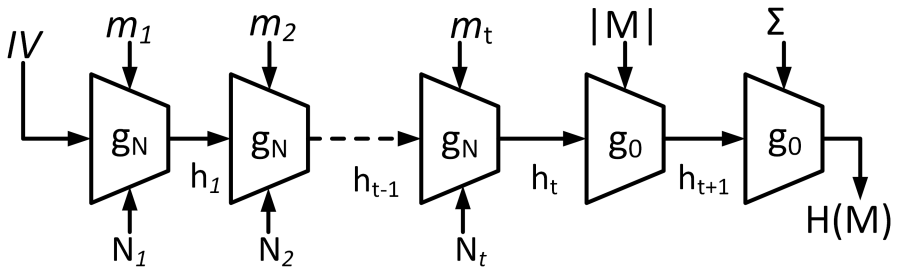

Figure 7.3: The Streebog iterated hash function.

to be able to retrieve the inputs of the previous compression function, we first launch the two-stage attack on $g_0(h_{t+1}, \Sigma)$. Because we observe both the correct and faulty values of $H(M)$, we can retrieve the values of Σ and h_{t+1}. To attack $g_0(h_t, |M|)$, the first stage of the attack requires the difference at h_{t+1} in addition to its value which cannot be deduced from observing the faulty $H'(M)$. This requirement can be fulfilled with a precomputed table T_1 for all the possible differences at h_{t+1} that result from injecting any fault at all the 64 byte positions of state M_{11} and their corresponding faulty $H'(M)$. Building this table is quite feasible for the fact that whatever the value of the induced fault at M_{11}, as depicted in Figure 7.1, each byte position at M_{11}^S may have up to 255 difference values which linearly maps to 255 one row differences Δh_{t+1} at state h_{t+1}. Accordingly, for each byte position in M_{11}^S, we linearly propagate the 255 possible differences forward to get Δh_{t+1}. Using our knowledge of the values of h_{t+1} and Σ, we evaluate the faulty $H'(M)$ corresponding to each difference. Finally, table T_1 will have 64×255 pairs of Δh_{t+1} and their corresponding $H'(M)$. Consequently, table T_1 enables us to complete the first stage of our attack because, when we inject a fault in M_{11}, the value of Δh_{t+1} corresponding to the resulting observed faulty $H'(M)$ is obtained from T_1. This step allows the recovery of the value of state M_{11} of $g_0(h_t, |M|)$.

The second stage of our attack requires the knowledge of the difference ΔM_{11} at state M_{11}. As depicted in Figure 7.2, a fault at a given position in M_{10} may have up to 255 difference after the Sbox which internally linearly map to 255 one row difference ΔM_{11} at state M_{11}. Since, we already know the value of state M_{11} from the previous step, we can get the corresponding 255 output differences after the Sbox at state M_{11}^S and linearly propagate them to get the full active state differences Δh_{t+1} at

state h_{t+1}. Similar to the previous step, we build a second table, T_2 with all the 64×255 differences Δh_{t+1} and their corresponding $H'(M)$. This table allows us to finish stage two of our DFA and recover the values of h_t and $|M|$ of $g_0(h_t, |M|)$. The knowledge of $|M|$ reveals the number of the processed message blocks and accordingly the number of compression function calls and their corresponding counter values. Finally, we repeat the previous two-steps for each compression function and hence invert all of the compression function calls within the iterated hash function and retrieve all their secret inputs. Although we consider the 512-bit version of the hash function in our 2-stage attack, it also works on the 256-bit version where the last four rows of the last compression function are truncated. We only have to add an initial stage that deals with the truncation. We utilize the fact that the position and value of a single byte difference in a given row can be uniquely identified from the knowledge of the difference in any two bytes in the same row after the linear transformation (*cf.* Lemma 3 in [55]). In the added initial stage, we retrieve half of the state of the last round. Then, in stage one of our attack, we recover the whole state in the round before the last one with the knowledge of half of the difference state after the linear transformation, then continue with the rest of the attack.

7.4 DFA on Streebog in Different MAC Settings

One of the prospective applications of the new Russian standard is using it in MAC schemes. Despite the fact that both the simple prefix and the secret-IV MACs [124] are vulnerable to length extension attacks, Streebog is by design not vulnerable to length extension attacks due to its finalization stage. This property may tempt users to adopt one of the simpler MAC constructions. Indeed, the designers of the NIST SHA-3 hash function, Keccak [26,39] state on their website that since Keccak is not vulnerable to length extension attacks, it does not need HMAC and propose that MAC computation can be done by concatenating the key with the message [80]. Accordingly, in what follows, we consider Streebog in both the simple and standardized MAC settings, and show how our attack can be used to recover the secret MAC key.

Simple prefix/Secret-IV MACs: As depicted in Figure 7.4, in the simple prefix MAC, the secret key is used as the first message block of the processed message in the iterative construction of the hash

function. More formally, $MAC(M) = H(K||M)$. On the other hand, in the secret-IV MAC, the standard initial value is replaced by the secret key in the iterative construction of the hash function. More formally, $MAC(M) = H_K(M)$, where $H_K(M)$ is the keyed hash value of the message M using the secret key K as the IV. The knowledge of the authenticated message reveals its corresponding

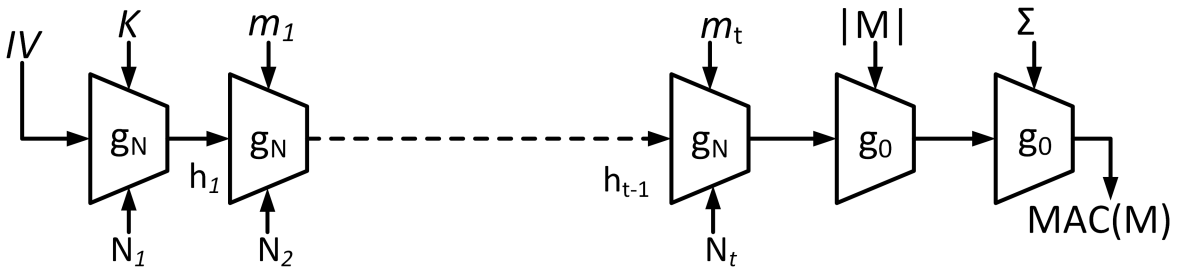

Figure 7.4: Simple prefix MAC using Streebog.

message blocks and accordingly their modular sum. We can retrieve the secret key of the simple prefix MAC using the two-stage DFA on the last compression function call. The attack recovers Σ which is the modular summation of all processed message blocks including the secret key. Accordingly, to recover the key, we simply subtract the summation of the known message blocks of the authenticated message from the retrieved Σ. As for secret-IV MAC, we use our DFA and invert the compression function calls until the first one with $N = 0$, the retrived chaining value is the secret key. In both schemes, if we do not know the authenticate message, we can easily retrieve the number of message blocks from $|M|$ and iterate the attack backwards until the compression function with $N = 0$ to recover the key.

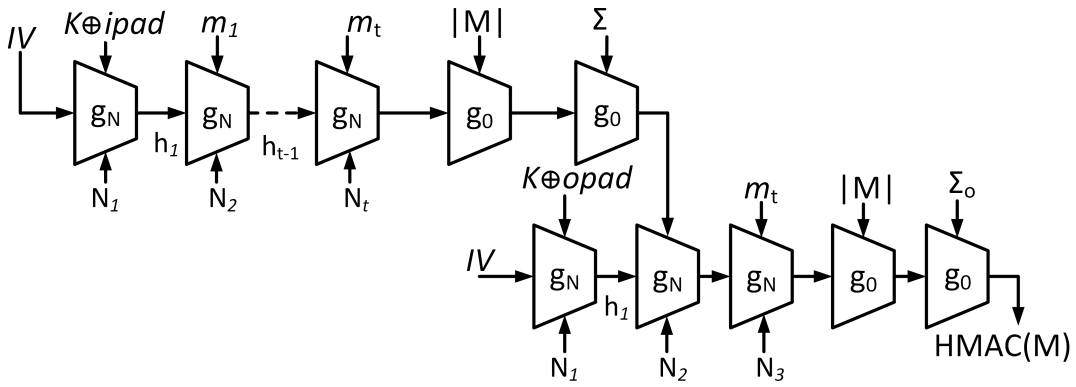

Figure 7.5: HMAC using Streebog.

HMAC/NMAC: HMAC [25] is defined as:

$$HMAC(M) = H((K \oplus opad)||H((K \oplus ipad)||M)),$$

where $opad$ and $ipad$ are known padding constants and H denotes a hash function call. The algorithm is standardized by ANSI, IETF, ISO and NIST, and is widely deployed in many Internet security protocols (e.g. SSL, SSH, IPSec). As depicted in Figure 7.5, the Streebog hash function is called twice. Our analysis works on the outer hash function call where $K \oplus opad$ is used as the first message block. Accordingly, our DFA is applied on the outer hash function and using the observed HMAC(M), we iterate the attack backwards to invert five compression function calls. The retrieved message block of the fifth backward compression function reveals the key value after xoring it with $opad$.

NMAC [25] employs two keys and is defined as:

$$NMAC(M) = H_{K_2}(H_{K_1}(M)),$$

where the keys are used as the initial values in the outer and inner hash function calls. The algorithm has a similar structure to HMAC but differs in that the first compression function call in both hash function calls in HMAC is omitted, and K_1 and K_2 are used as the IV for the following compression function call. Accordingly, if Figure 7.5 is to describe NMAC, we omit $g_N(IV, K \oplus ipad, N_1)$ and replace the resulting h_1 by K_1, and remove $g_N(IV, K \oplus opad, N_1)$ and use K_2 as the IV for the following compression function call. In the sequel, our attack works first to recover K_2 by iterating the two-stage attack backwards for four compression function calls. Afterwards, the retrieved message block corresponding to the output of the inner hash function is used to further recover K_1 from the inner hash function application.

7.5 Simulation Results

Since the attack has a very low complexity, we have simulated three scenarios of the attack on the compression function on an 4-core Intel i7 CPU running at 2.67GHz and the secret inputs were recovered in less than one minute. The scenarios vary in the assumptions of whether the attacker

can control the injection of distinct faults and if the faulted byte position can be chosen or not. The provided average fault requirements are the result of running our simulation using 1000 different inputs to the compression function. As shown by our simulations, the number of required faults to retrieve 128 bytes in both stages depends on the assumptions used during fault injections. In what follows, we give the results of our simulation:

1. When the faults are selected distinctly and the byte position $[r, c]$ is chosen by the attacker, then one needs an average of 338 faults which is equivalent to an average of 2.635 faults per byte.

2. If we randomly induce non distinct one-bit faults and select the byte positions, then the attack requires an average of 394 fault injections in total with an average of 3.077 faults per byte.

3. In the case where both the byte position and the induced one-bit faults are randomly chosen, the attack requires an average of 1640 fault injections in total, and accordingly an average 12.807 fault per byte.

7.6 Conclusion

In this chapter, we have investigated the security of the new Russian hash function standard GOST R 34.11-2012 with respect to differential fault analysis. In particular, we have proposed a two-stage approach that considers the compression function operating with secret inputs. Using one-bit faults, the first stage of our attack bypasses the final feedforward and retrieves the internal state of the cipher used in the compression function. The second stage retrieves one of the round keys used in the cipher which enables the generation of the rest of the round keys and consequently, the input chaining value and message block are recovered. We have simulated the attack on the compression function with different assumptions regarding the control of the attacker over the induced faults and the faulted position. The results show that our two-stage attack requires between 338 and 1640 faults on average, depending on what are the assumptions of the employed fault model. Moreover, we have proposed a feasible precomputation step where we require two tables of size 2^{14} state each to enable the extension of the attack to the whole hash function. Finally, we have shown how our proposed approach is used to recover the secret MAC key when Streebog is used in simple prefix, secret-IV, HMAC, and NMAC

settings. A naive approach to prevent our attack is to use spatial/temporal algorithm level redundancy and to disable the device output if the two produced MAC tags do not match. Another approach is to add parity bits to detect corruptions of the inner state registers and disable the device output if any of these parity checks is violated. Efficient fault analysis resistant implementations for Streebog, as well as for other hash functions deployed in MAC schemes, need to be addressed in future research.

Chapter 8

Preimage Analysis of the Maelstrom-0 Hash Function

Maelstrom-0 is the second member of a family of AES-based hash functions whose designs are pioneered by Paulo Baretto and Vincent Rijmen. According to its designers, the function is designed to be an evolutionary lightweight alternative to the ISO standard Whirlpool. In this chapter, we study the preimage resistance of the Maelstrom-0 hash function which employs the 3CM chaining construction. More precisely, we apply a meet-in-the-middle preimage attack on the compression function and combine it with a guess and determine approach which allows us to obtain a 6-round pseudo preimage for a given compression function output with time complexity of 2^{496} and memory complexity of 2^{112}. Then, we propose a four stage attack in which we adopt another meet-in-the-middle attack and a 2-block multicollision approach to defeat the two additional checksum chains and turn the pseudo preimage attack on the compression function into a preimage attack on the hash function. Using our approach, preimages of the 6-round reduced Maelstrom-0 hash function are generated with time complexity of 2^{505} and memory complexity of 2^{112}.

8.1 Introduction

Maelstrom-0 is an AES-based hash function that adopts a modified chaining scheme called 3CM [53]. The function is proposed by Filho, Barreto, and Rijmen as an evolutionary lighter alterna-

tive to its predecessor Whirlpool. Maelstrom-0 is considered the second member of a family of hash functions which is preceded by Whirlpool and followed by Whirlwind. The design of Maelstrom-0 is heavily inspired by Whirlpool but adopts a simpler key schedule and takes into account the recent development in hash function cryptanalysis. Particularly, the designers consider those attacks where the cryptanalytic techniques which are applicable on the compression function can be easily mapped to the hash function due to the simplicity of the Merkle-Damgård construction used by Whirlpool. In addition to adopting a simpler key schedule which makes Maelstrom-0 more robust and significantly faster than Whirlpool, the designers employ the Davis-Mayer compression mode which is the only mode among the twelve secure constructions that naturally allows the compression function to accept a message block size different from the chaining value size, thus allowing faster hashing rate [53]. Also, all the remaining eleven constructions XOR the message and the chaining value block, thus forcing either truncation or padding to cope with the different sizes, and it is unclear to what extent truncation or padding might adversely affect the security analysis.

The most important feature in the design of Maelstrom-0 is the proposal of a new chaining construction called 3CM which is based on the 3C/3C+ family [61]. This construction computes two checksums from the generated intermediate chaining values, concatenates them, and as a finalization step processes the result as a message block in the last compression function call. This finalization step aims to thwart some generic attacks on the MD construction used in Whirlpool such as long second preimage and herding attacks, and also inhibits length extension attacks. According to the designers of Maelstrom-0, the proposed finalization step mitigates the applicability of extending attacks on the compression function to the hash function. Unfortunately, this is not the case in our attack where we employ a 4-stage approach that uses a modified technique which defeats the 3CM chaining construction [56, 57, 59] and combines it with another meet-in-the-middle (MitM) attack to extend a pseudo preimage attack on the compression function to a preimage attack on the hash function.

Literature related to the cryptanalysis of Maelstrom-0 include the analysis of the collision resistance of its compression function by Kölbl and Mendel [92] where the weak properties of the key schedule were used to produce semi free-start collision for the 6 and 7 round reduced compression function and semi free-start near collision for the 8 and 10-rounds compression function. An analysis of the used chaining construction was presented by Gauravaram and Kelsey in [56, 57] along with a

long second preimage and herding attacks on the hash function. Finally, Mendel *et al.* used the rebound attack to show how employing a message block whose size is double that of the chaining state is used to present a free start collisison on the 8.5 reduced round compression function [107].

In this chapter, we investigate the security of Maelstrom-0 and its compression function, assessing their resistance to the MitM preimage attacks. Employing the partial matching and initial structure concepts [130], we present a pseudo preimage attack on the 6-round reduced compression function. In the presented attack, we employ a guess and determine approach [132] to guess parts of the state. This approach helps in maintaining partial state knowledge for an extra round when all state knowledge is lost due to the wide trail effect. The proposed 6-round execution separation maximizes the overall probability of the attack by balancing the chosen number of starting values and the guess size. Finally, we propose a four stage approach which combines a 2-block multicollision attack [56, 57] with a second MitM attack to bypass the effect of the 3CM checksum used in the finalization step. Our approach is successfully used to generate preimages of the 6-round reduced Maelstrom-0 hash function using the presented pseudo preimage attack on the last compression function. Up to our knowledge, our analysis is the first to consider the hash function and not only the compression function of Maelstrom-0.

The rest of the chapter is organized as follows. In the next section, the description of the Maelstrom-0 hash function along with the notation used throughout the chapter are given. Afterwards, in section 8.3, we provide detailed description of the pseudo preimage attack on the compression function. In section 8.4, we show how preimages of the hash function are generated using our four stage approach and the attack presented in section 8.3. Finally, the chapter is concluded in section 8.5.

8.2 Specifications of Maelstrom-0

Maelstrom-0 is an AES-based iterative hash function designed by Filho, Barreto and Rijmen [53]. Its compression function processes 1024-bit message blocks and a 512-bit chaining value. As depicted in Figure 8.1, the message M is padded by 1 followed by zeros to make the length of the last block 768. Then the remaining 256 bits are used for the binary representation of the message length

$|M|$. Hence the padded message has the form $M = m_1||m_2||\cdots||m_k$, where the last 256-bits of m_k denote $|M|$. The compression function is iterated in the 3CM chaining mode which is based on 3C/3C+

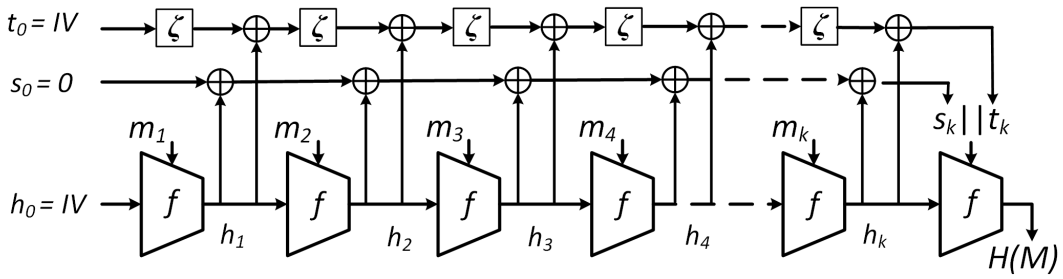

Figure 8.1: The Maelstrom-0 hash function.

family [61]. Given that h_i denotes the internal state value after processing the message block m_i, i.e., $h_i = f(m_i, h_{i-1})$ with $h_0 = IV$, this chaining mode generalizes the Merkle-Damgård construction by maintaining three chains h_i, s_i, t_i instead of only h_i. The extra two chains are transformed into an additional message block $m_{k+1} = s_k||t_k$. The second chain s_i is a simple XOR accumulation of all intermediate compression function outputs, recursively defined as $s_0 = 0$, $s_i = h_i \oplus s_{i-1}$. The third chain is recursively defined as $t_0 = IV$, $t_i = h_i \oplus \zeta(t_{i-1})$ where an LFSR is employed by ζ to update t_{i-1} by left shifting it by one byte followed by a one byte XOR. More precisely, we compute the hash value h_i in the following way:

$$\begin{aligned} h_0 &= IV, \\ h_i &= f(h_{i-1}, m_i), \text{ for } i = 1, 2, ..., k, \\ H(M) &= f(h_k, s_k||t_k). \end{aligned}$$

The compression function, f, employs a block cipher, E and uses the Davis-Mayer mode of operation. The internal cipher is based on the one used in Whirlpool where it only differs in the key schedule. The round function which operates on 8×8 byte state is initially loaded with the input chaining value. As depicted in Figure 8.2, the state is updated through 10 rounds and one key addition at the beginning. One round of the state update function consists of the application of the following four transformations:

- The nonlinear layer γ: A transformation that consists of parallel application of a nonlinear Sbox on each byte using an 8-bit Sbox. The used Sbox is the same as the one used in Whirlpool.

- The cyclical permutation π: This layer cyclically shifts each column of its argument independently, so that column j is shifted downwards by j positions, $j = 0, 1, \cdots, 7$.

- The linear diffusion layer θ: A MixRow operation where each row is multiplied by an 8×8 MDS matrix over F_{2^8}. The values of the matrix are chosen such that the branch number of MixRow is 9. Therefore the total number of active bytes at both the input and output is at least 9.

- The key addition σ: A linear transformation where the state is XORed with a round key state.

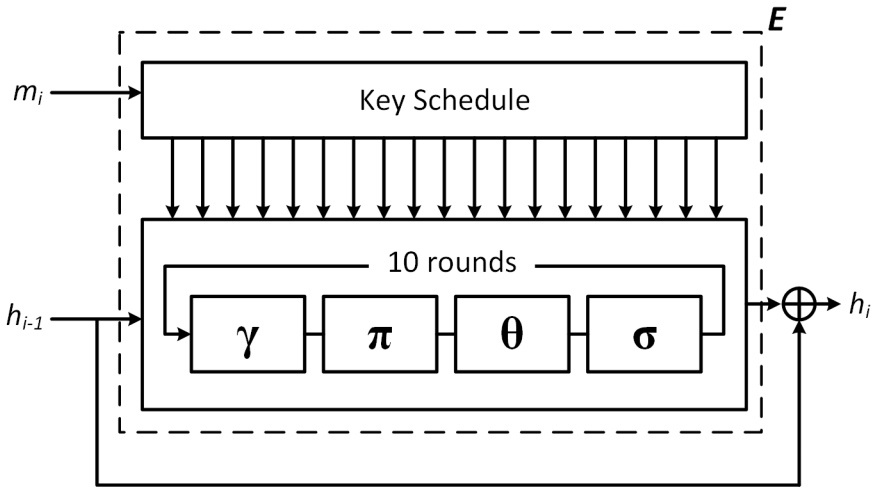

Figure 8.2: The Maelstrom-0 compression function.

The key schedule takes as input the 1024-bit message block and generates the 512-bit round keys, $K_0, K_1, \cdots, K_{10}$. Since the key scheduling process is not relevant to our attack, we do not give a detailed description of the round key generation function. For more details on the specification of Maelstrom-0, the reader is referred to [53].

- Notation: Let X be (8×8) byte state denoting the internal state of the function. The following notation is used in our attacks:

- X_i: The message state at the beginning of round i.

- X_i^U: The message state after the U transformation at round i, where $U \in \{\gamma, \pi, \theta, \sigma\}$.

- $X_i[r, c]$: A byte at row r and column c of state X_i.

- X_i[row r]: Eight bytes located at row r of state X_i.
- X_i[col c]: Eight bytes located at column c of state X_i.

8.3 Pseudo Preimage Attack on the 6-Round Reduced Compression Function

In our analysis of the compression function, we are forced to adopt a pseudo preimage attack because the compression function operates in Davis-Mayer mode. Consequently, using the cut and splice technique causes updates in the first state which is initialized by the chaining value. In our attack, we start by dividing the two execution chunks around the initial structure. More precisely, we separate the six attacked rounds into a 3-round forward chunk and a 2-round backward chunk around the starting round represented by the initial structure. The proposed chunk separation is shown in Figure 8.3. The number of the forward and backward starting values in the initial structure amounts for the complexity of the attack. Accordingly, one must try to balance the number starting values for each chunk and the number of known bytes at the matching point at the end of each chunk. The total number of starting values in both directions should produce candidate pairs at the matching point to satisfy the matching probability.

To better explain the idea, we start by demonstrating how the initial structure is constructed. The main objective of the MitM attack separation is to maximize the number of known bytes at the start of each execution chunk. This can be achieved by selecting several bytes as neutral so that the number of corresponding output bytes of the θ and θ^{-1} transformations at the start of both chunks that are constant or relatively constant is maximized. A relatively constant byte is a byte whose value depends on the value of the neutral bytes in one execution direction but remains constant from the opposite execution perspective. As depicted in Figure 8.4, we want to have six constants in the lowermost row in state a, then we need to evaluate the possible values of the corresponding red row in state b such that the values of the selected six constants in state a hold. The values of the lowermost red row in state b are the possible forward starting values. For the lowermost row in state b, we randomly choose the six constant bytes in a[row 7] and then evaluate the values of red bytes in b[row 7] so that after

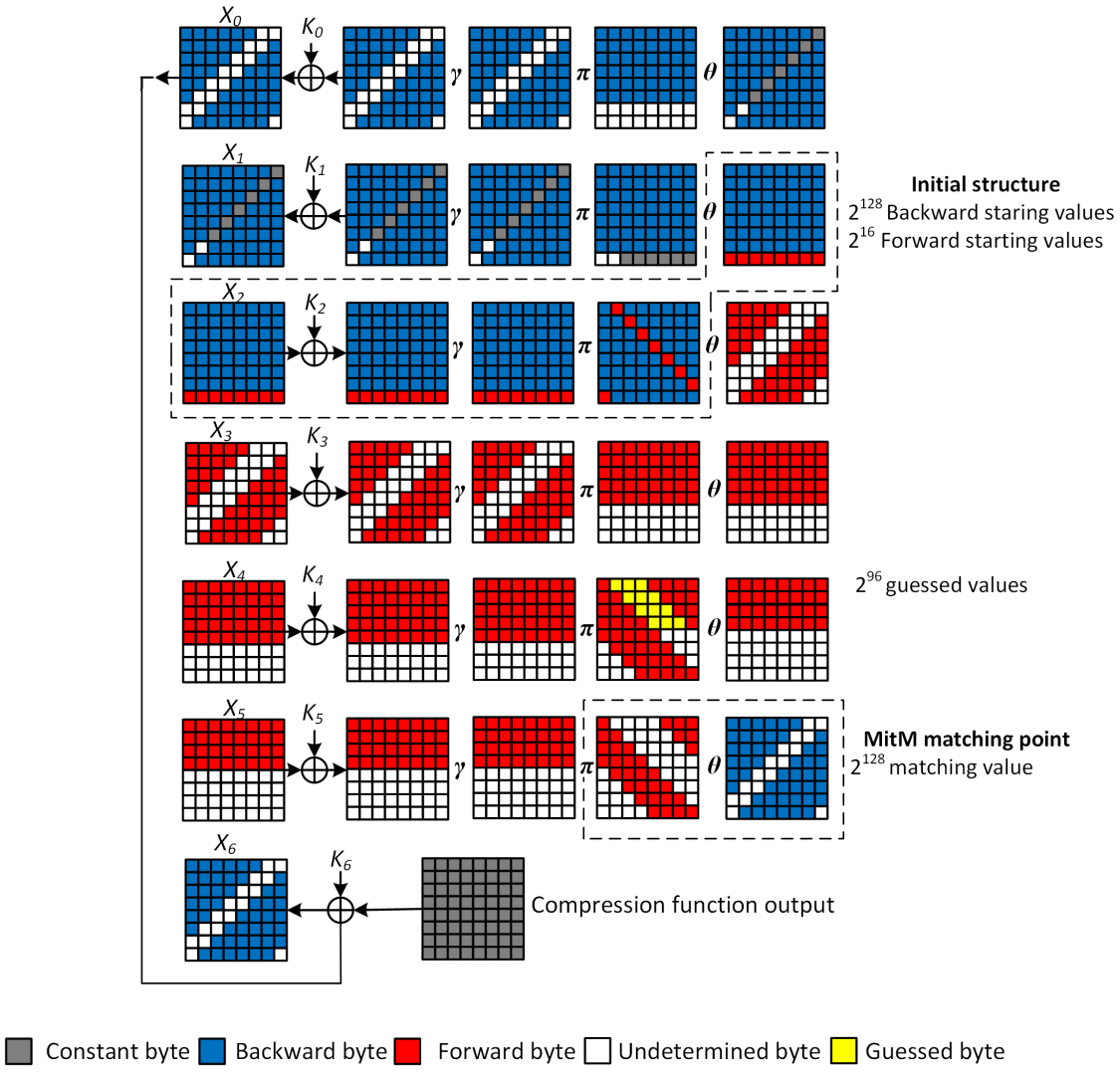

Figure 8.3: Chunk separation for the 6-round MitM pseudo preimage attack the compression function.

applying θ^{-1} on b[row 7], the chosen values of the six constants hold. Since we require six constant bytes in the lowermost row in state a, we need to maintain six variable bytes in b[row 7] in order to solve a system of six equations when the other two bytes are fixed. Accordingly, for the last row in state b, we can randomly choose any two red bytes and compute the remaining six so that the output of θ^{-1} maintains the previously chosen six constant bytes at state a. To this end, the number of forward starting values is 2^{16}. Similarly, we choose 40 constant bytes in state d and for each row in state c we randomly choose two blue bytes and compute the other five such that after the θ transformation we get the predetermined five constants at each row in d. However, the value of the five shaded red bytes in each row of state d depends also on the one red byte in the rows of state c. We call these

bytes relative constants because their final values cannot be determined until the forward execution starts and these values are different for each forward execution iteration. Specifically, their final values are the predetermined constants acting as offsets which are XORed with the corresponding red bytes multiplied by the MDS matrix coefficients. In the sequel, we have two free bytes for each row in c which means 2^{128} backward starting values.

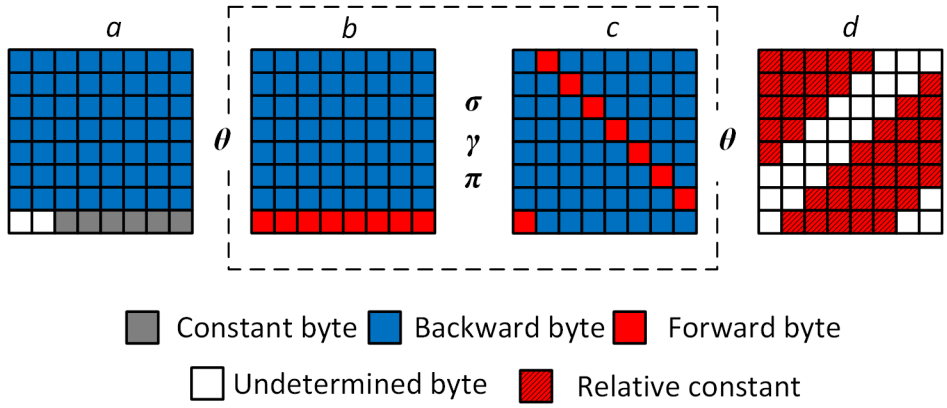

Figure 8.4: Initial structure used in the attack on the 6-round compression function.

Following Figure 8.3, due to the wide trail strategy where one unknown byte results in a full unknown state after two rounds, we lose all state knowledge after applying θ on X_4^π. To maintain partial state knowledge in the forward direction and reach the matching point at X_5^π, we adopt a guess and determine approach [132], by which, we can probabilistically guess the undetermined bytes in some rows of the state at round 4 before the linear transformation. Thus, we maintain knowledge of some state rows after the linear transformation θ which are used for matching. One have to carefully choose the number of guessed bytes and both starting values in the initial structure to result in an acceptable number of correctly guessed matching pairs. Accordingly, we guess the twelve unknown yellow bytes in state X_4^π. As a result, we can reach state X_5^π with four determined bytes in each row where matching takes place.

As depicted in Figure 8.3, the forward chunk begins at X_2^θ and ends at X_5^π which is the input state to the matching point. The backward chunk starts at X_1^π and ends after the feedforward at X_5^θ which is the output state of the matching point. The red bytes denote the bytes which are affected by the forward execution only and thus can be independently calculated without the knowledge of the blue bytes. White words in the forward chunk are the ones whose values depend on the blue bytes of

the backward chunk. Accordingly, their values are undetermined. Same rationale applies to the blue bytes of backward execution. Grey bytes are constants which can be either the compression function output or the chosen constants in the initial structure.

At the matching point, we partially match the available row bytes from the forward execution at X_5^π with the corresponding row bytes from the backward execution at X_5^θ through the linear θ transformation. In each row, we have four and six bytes from the forward and backward executions, respectively. Since the linear mapping is performed on bytes, we compose four byte linear equations in two unknown bytes. Then we evaluate the values of the two unknown bytes from two out of the four equations and substitute their values in the remaining two equations. With probability 2^{-16} the two remaining byte equations are satisfied. Hence, the matching probability for one state row is 2^{-16}. Thus, the partial matching probability for the whole state is $2^{8\times-16=-128}$.

For our attack, the chosen number for the forward and backward starting values, and the guessed values are 2^{16}, 2^{128}, and 2^{96}, respectively. Setting these parameters fixes the number of matching values to 2^{128}. The chosen parameters maximize the attack probability as we aim to increase the number of starting forward values and keep the number of backward and matching values as close as possible and larger than the number of guessed values. In what follows, we give a description of the attack procedure and complexity based on the above chosen parameters:

1. Randomly choose the constants in X_1^π and X_2^θ and the input message block value.

2. For each forward starting value fw_i and guessed value g_i in the 2^{16} forward starting values and the 2^{96} guessed values, compute the forward matching value fm_i at X_5^π and store (fw_i, g_i, fm_i) in a lookup table T.

3. For each backward starting value bw_j in the 2^{128} backward starting values, we compute the backward matching value bm_j at X_5^θ and check if there exists an $fm_i = bm_j$ in T. If found, then a partial match exists and the full match should be checked. If a full match exists, then we output the chaining value h_{i-1} and the message m_i, else go to step 1.

The complexity of the attack is evaluated as follows: after step 2, we have $2^{16+96} = 2^{112}$ forward matching values which need 2^{112} memory for the look up table. At the end of step 3, we have 2^{128}

backward matching values. Accordingly, we get $2^{112+128} = 2^{240}$ partial matching candidate pairs. Since the probability of a partial match is 2^{-128} and the probability of a correct guess is 2^{-96}, we expect $2^{240-128-96} = 2^{16}$ correctly guessed partially matching pairs. To check for a full match, we want the partially matching starting values to result in the correct values for the 48 unknown bytes in both X_5^{π} and X_5^{θ} that make the blue and red words hold. The probability that the latter condition is satisfied is $2^{48\times-8} = 2^{-384}$. Consequently, the expected number of fully matching pairs is 2^{-368} and hence we need to repeat the attack 2^{368} times to get a full match. The time complexity for one repetition is 2^{112} for the forward computation, 2^{128} for the backward computation, and 2^{16} to check that partially matching pairs fully match. The overall time complexity of the attack is $2^{368}(2^{112} + 2^{128} + 2^{16}) \approx 2^{496}$ and the memory complexity is 2^{112}.

8.4 Preimage of the Maelstrom-0 Hash Function

In this section, we propose a 4-stage approach by which we utilize the previously presented pseudo preimage attack on the Maelstrom compression function to produce a preimage for the whole hash function. The designers of Maelstrom-0 proposed the 3CM chaining scheme that computes two additional checksum chains specifically to inhibit the ability of extending attacks on the compression function to the hash function. The two additional checksums are computed from a combination of the XOR of the intermediate chaining values, then the two results are concatenated and processed as the input message block of the last compression function call in the hash function. At first instance, this construction seems to limit the scope of our attack to the compression function. Nevertheless, employing the 4-stage approach, a preimage of the hash function can be found when we consider a large set of messages that produce different combinations of intermediate chaining values and thus different checksums and combine it with a set of pseudo preimage attacks on the last compression function call. Hence, another MitM attack can be performed on both sets to find a message that correspond to the retrieved checksums. As depicted in Figure 8.5, the attack is divided into four stages:

1. Given the hash function output $H(M)$, we produce 2^p pseudo preimages for the last compression function call. The output of this step is 2^p pairs of the last chaining value and the two checksums

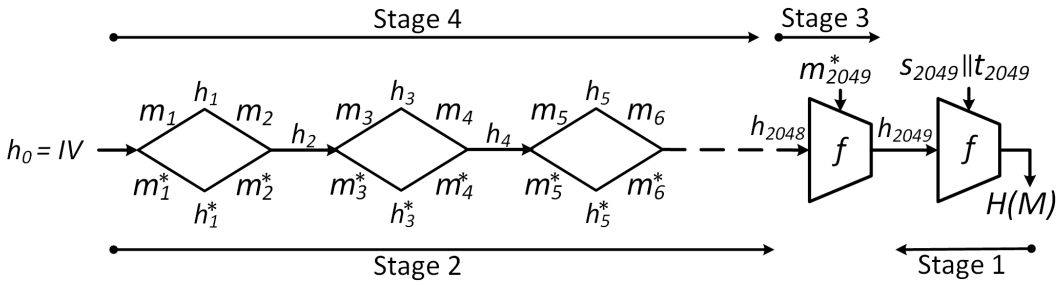

Figure 8.5: A 4-stage preimage attack on the Maelstrom-0 hash function.

$(h_{2049}, s_{2049}, t_{2049})$. We store these results in a table T.

2. In this stage, we construct a set of 2-block messages such that all of them collide at h_{2048}. This structure is called a 2-block multicollision of length 1024 [57, 76]. More precisely, an n-block multicollision of length t is a set of 2^t messages where each message consists of exactly $n \times t$ blocks and every consecutive n application of the compression function results in the same chaining value. Consequently, we have 2^t different possibilities for the intermediate chaining values and all the 2^t n-block messages lead to the same $h_{n\times t}$ value. Constructing a 2^t n-block multicollision using exhaustive collision search requires a time complexity of $t(2(n-1)+2^{b/2})$, where b is the chaining state size, and a memory complexity of $t(2 \cdot n)$ message to store t two messages of n-block each. In our case, we generate 2-block multicollision of length 1024 which gives us 2^{1024} 2-block message combinations, and each 2-block collision gives us two choices for the checksum of two consecutive chaining values. In other words, in the first 2-block collision, we either choose (h_1, h_2) or (h^*_1, h_2) and thus two choices for the checksum chains. To this end, we have 2^{1024} different 2-block massages stored in $1024 \cdot 2 \cdot 2 = 2^{12}$ memory and hence 2^{1024} candidate chaining checksums.

3. At this stage, we try to connect the resulting chaining value, h_{2048}, from stage 2 to one of 2^p chaining values, h_{2049}, stored in T which was created in stage 1, using the freedom of choosing m_{2049}. Specifically, we randomly choose 512 bit of m^*_{2049}, then properly pad it and append the message length, and using h_{2048} generated by the multicollision, we compute h^*_{2049} and check if it exists in T. As T contains 2^p entries, it is expected to find a match after 2^{512-p} evaluations of

the following compression function call:

$$h^*_{2049} = f(h_{2048}, m^*_{2049}).$$

Once a matching h^*_{2049} value is found in T, the corresponding checksums s^*_{2049}, t^*_{2049} are retrieved. Hence the desired checksums at the output of the multicollision, s_{2048} and t_{2048} are equal to $s^*_{2049} \oplus h^*_{2049}$ and $\zeta^{-1}(t^*_{2049} \oplus h^*_{2049})$, respectively.

4. At the last stage of the attack, we try to find a message M out of the 2^{1024} 2-block messages generated in stage 2 that results in checksums equal to the ones retrieved in stage 3. For this, we form a system of 1024 equations in 1024 unknowns to select one combination from the 2^{1024} different combinations of possible chaining checksums which make the retrieved two checksums hold. Note that, the algorithm proposed in [56] which employs 2^{512} 2-block multicollision and treats the two checksums independently by solving two independent systems of 512 equations cannot work on 3CM, as the two checksums are dependent on each other. This algorithm only works on the 3C chaining construction [57, 59] because it utilizes only one checksum. Accordingly, in our solution, we adopt 1024 2-block messages to find a common solution for the two checksums simultaneously, hence, having the required freedom to satisfy two bit constraints for each bit position in the two checksums. The time complexity of this stage is about $1024^3 = 2^{30}$.

The time complexity of the attack is evaluated as follows: we need $2^p\times$ (complexity of pseudo preimage attack) in stage 1, $1024 \times 2^{256} + 2048 \approx 2^{266}$ to build the 2-block multicollision at stage 2, 2^{512-p} evaluations of one compression function call at stage 3, and finally 2^{30} for stage 4. The memory complexity for the four stages is as follows: 2^p 3-states to store the pseudo preimages in stage 1 and 2^{112} for the pseudo preimage attack, and 2^{12} for the multicollision in stage 2. Since the time complexity is highly influenced by p, so we have chosen $p = 8$ to maximize the attack probability. Accordingly, preimages for the 6-round Maelstrom-0 hash function can be produced with a time complexity of $2^{8+496} + 2^{266} + 2^{512-8} + 2^{30} \approx 2^{505}$. The memory complexity of attack is dominated by the memory requirements of the pseudo preimage attack on the compression function which is given by 2^{112}.

8.5 Conclusion

In this chapter, we have investigated Maelstrom-0 and its compression function with respect to MitM preimage attacks. We have shown that with a carefully balanced chunk separation and the use of a guess and determine approach, pseudo preimages for the 6-round reduced compression function can be generated. Moreover, we have analyzed the employed 3CM chaining scheme which is designed specifically to inhibit the ability of extending attacks on the compression function to the hash function, and proposed a 4-stage approach to bypass its effect and turn the pseudo preimage attack on the compression function to a preimage attack on the hash function. Accordingly, 6-round hash function preimages are generated with a time complexity of 2^{505} and a memory complexity of 2^{112}.

Chapter 9

A Meet in the Middle Attack on Kuznyechik

In this chapter, we present a meet-in-the-middle attack on the 5-round reduced Kuznyechik cipher which has been recently chosen to be standardized by the Russian federation. Our attack is based on the differential enumeration approach, where we propose a distinguisher for the middle rounds and match a sequence of state differences at its output. However, the application of the exact approach is not successful on Kuznyechik due to its optimal round diffusion properties. Accordingly, we adopt an equivalent representation for the last round where we can efficiently filter ciphertext pairs and launch the attack in the chosen ciphertext setting. We also utilize partial sequence matching which further reduces the memory and time complexities through relaxing the error probability. The adopted partial sequence matching approach enables successful key recovery by matching parts of the generated sequence instead of the full sequence matching used in the traditional setting of this attack. For the 5-round reduced cipher, the 256-bit master key is recovered with a time complexity of $2^{140.3}$, a memory complexity of $2^{153.3}$, and a data complexity of 2^{113}.

9.1 Introduction

The Russian Federation has recently published a project for a new standard for block cipher encryption algorithm [3]. A draft for this new algorithm was presented by its designers at CTCrypt 2014 [135]. The new algorithm, Kuznyechik, (Grasshopper in Russian), is chosen [3] to accompany the current Russian encryption standard GOST 28147-89 [1]. Although the current standard is con-

sidered a lightweight cipher [122], and only theoretical attacks on the full round cipher have been presented [51, 74], it operates on 64-bit blocks of data which is not sufficient for the current requirements [135]. Hence, the need arose for a new standard with larger block length which is intended to supersede in the future the current GOST 28147-89 cipher.

The meet-in-the-middle (MitM) attack was first proposed in 1977 by Diffie and Hellman [50] for the analysis of the Data Encryption Standard (DES). Ever since, the attack has been evolving to cryptanalyze block ciphers such as Present and Prince [38], KTANTAN [33], LBlock [10], and mCrypton [68]. Additionally, MitM preimage attacks on hash functions have been presented on HAS-160 [71], Whirlpool [132], Whirlwind [13], and Streebog [12]. The first application of a non standard type of MitM attacks on AES was due to the work of Demirci and Selçuk [47], whose approach opened the door to a new line of research. They constructed a truncated differential four round distinguisher, and showed that if the input to the distinguisher has only one active byte that takes all the possible values, then each output byte can be evaluated as a function of 25 parameters. They also showed that the values of each output byte corresponding to the input byte values form an ordered sequence that can be used as a property to identify the right key guess. The main disadvantage of their technique is the high memory complexity which is required by a precomputation table to store all the sequences resulting from all the possible combinations of the 25 byte parameters. Accordingly, the approach was only valid to attack seven and eight rounds of AES-192 and AES-256, and not the 128-bit version. Afterwards, the number of parameters was reduced to 24 bytes in [48], which lowered the size of the table by a factor of 8.

Afterwards, Dunkelman *et al.* proposed the idea of multisets and differential enumeration [52] to tackle the high memory requirements of the approach of Demirci and Selçuk [47]. While the concept of multisets provides better encoding of the ordered sequence which reduces the size of the table by a factor of 4, differential enumeration can be considered the main advantage of their attack. More precisely, differential enumeration allows the ordered sequence to be generated by the knowledge of 16 byte parameters only instead of 24, which brings the number of entries of the table down from 2^{192} to 2^{128}. This gain is attributed to the use of a low probability truncated differential distinguisher where

the generated sequences at its output can only take a restricted number of values. Accordingly, one must initially search through a large amount of input data pairs to find one pair that satisfies the chosen distinguisher. Indeed, their proposal has reduced the memory complexity of the attack at the expense of its data complexity required to search for the right input data pair.

Later on, Derbez *et al.* [49] improved the attack of Dunkelman *et al.* by borrowing ideas from the rebound attack [107], and proving that not all of the sequences in the table can be verified by input data satisfying the truncated distinguisher. They have presented an efficient enumeration technique and showed that the whole set of sequences can take only 2^{80} values and not 2^{128} as with the case in the attack by Dunkelman *et al.* Accordingly, all the generated sequences require the knowledge of only 10 byte parameters, thus the number of entries of the precomputation table is further reduced to 2^{80}. A direct consequence of their improvement is that the memory complexity is not the bottleneck of the attack anymore but both the time and data complexities are. Nevertheless, their attack is considered the most efficient attack on the 7-round reduced AES-128 and 8-round reduced AES-192/256. They have also used a 5-round distingusher to attack the 9-rounds reduced AES-256.

Finally, Li *et al.* [96] employed a key-dependent sieve to further reduce the memory complexity of Derbez's attack and present an attack on 9 rounds AES-192 using a 5-round truncated differential distinguisher.

In this chapter, we present a MitM attack on Kuznyechik using the idea of efficient differential enumeration. Unlike AES, Kuznyechik employs an optimal diffusion transformation applied to the whole state, where one byte difference results in a full active state with certainty after one round. Consequently, we construct a three round distinguisher in our attack to recover 16-bytes of the master key of the reduced 5-round cipher. The direct application of the attack on Kuznyechik requires a time complexity that exceeds that of the exhaustive search for the 256-bit key, which is also attributed to the optimal round diffusion. Accordingly, we adopt an equivalent representation of the last round which allows us to efficiently select ciphertext pairs that satisfy the lower half of the differential path used in the attack with certainty. Hence, our attack is considered in the chosen ciphertext setting. This modification lowers the time complexity of the online phase by a factor of 2^{120} because we

eliminate the probabilistic propagation of the 16 to 1 transition through the linear transformation from the ciphertext side. We also present partial sequence matching, by which we generate, store, and match parts of the ordered sequence while maintaining negligible probability of error. Indeed, not only we decrease the partially encrypted/decrypted data during online matching and thus the overall time complexity of the attack is lowered, but this approach also reduces the memory requirements of the attack.

The rest of the chapter is organized as follows. In the next section, the description of the Kuznyechik block cipher along with the notation used throughout the chapter are provided. Afterwards, in section 9.3, a preliminary security analysis of Kuznyechik against well known attacks is given. In section 9.4, we provide a detailed description of the proposed distinguisher, the adopted attack procedure, our filtering approach, the proposed partial sequences idea. Finally, the chapter is concluded in section 9.5.

9.2 Specification of Kuznyechik

Kuznyechik [3, 135] is an SPN block cipher that operates on a 128-bit state. The cipher employs a 256-bit key which is used to generate ten 128-bit round keys. As depicted in Figure 9.1, the encryption procedure updates the 16-byte state by iterating the round function for nine rounds. The round function consists of:

- SubBytes (S): A nonlinear byte bijective mapping.
- Linear Transformation (L): An optimal diffusion operation that operates on a 16-byte input and has a branch number = 17.
- Xor layer (X): Mixes round keys with the encryption state.

Additionally, an initial XOR layer is applied prior to the first round. The full encryption function where the ciphertext C is evaluated from the plaintext P is given by:

$$C = (X[K_{10}] \circ L \circ S) \circ \cdots \circ (X[K_2] \circ L \circ S) \circ X[K_1](P)$$

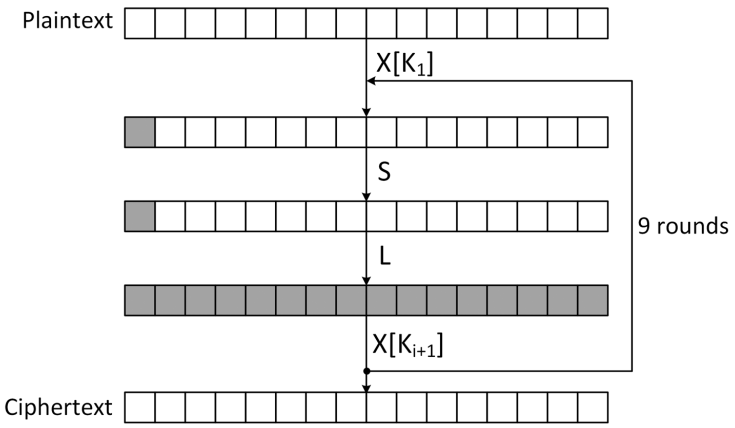

Figure 9.1: The encryption procedure of Kuznyechik

In our attack, we use an equivalent representation of the last round function. The representation exploits the fact that both the linear transformation, L, and the Xor operation, X, are linear and thus, their order can be swapped. One has to first Xor the data with an equivalent round key, then apply the linear transformation, L, to the result. We evaluate the equivalent round key after the last round r by $EK_{r+1} = L^{-1}(K_{r+1})$. We also use the following property of the Sbox:

Property 1. For a given Sbox differential $(\delta x, \delta y)$, the average number of solutions to $S(x) \oplus S(x \oplus \delta x) = \delta y$ is 1 over all x.

For further details regarding the employed Sboxes and linear transformation, the reader is referred to [135].

Key schedule: The ten 128-bit round keys are derived from the 256-bit master key by undergoing 32 rounds of a Feistel structure function. The first two round keys, K_1 and K_2, are derived directly from the master key, K, as follows: $K_1 \parallel K_2 = K$. As depicted in Figure 9.2, each pair of subsequent round keys is extracted after eight rounds of execution. During each round, the same round function used in the encryption procedure is applied to the right half of the input to the Feistel round. However, round constants are used with the X operation instead of round keys. The 128-bit round constants C_i are defined as follows: $C_i = L(i),\ i = 1, 2, \cdots, 32$. Let $F[C](a, b)$ denote $(L \circ S \circ X[C](a) \oplus b, a)$, where C, a, and b are 128-bit inputs. The rest of the round keys are derived from the first two round

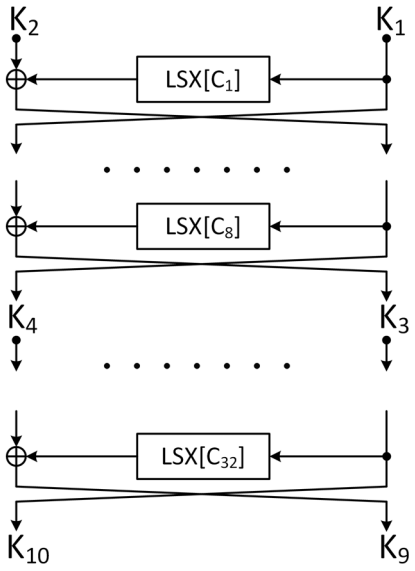

Figure 9.2: The key schedule of Kuznyechik

keys, K_1 and K_2, as follows:

$$(K_{2i+1}, K_{2i+2}) = F[C_{8(i-1)+8}] \circ \cdots \circ F[C_{8(i-1)+1}](K_{2i-1}, K_{2i}), i = 1, 2, 3, 4.$$

It is interesting to note that Kuznyechik bares a lot of resemblance with one of the AES predecessors, Khazad [24]. In particular, both ciphers employ an iterative SPN structure for updating the input block state, where the adopted linear transformation has an optimal diffusion properties. Also, they both use Fiestel network for the round keys generation. While in Kuznyechik, two round keys are generated after eight rounds of execution, only one round of execution separates consecutive round keys in Khazad. They also differ in that Khazad employs involution Sboxes and linear transformation, and it does not use a linear transformation in the last round.

- Notation: The following notation is used throughout the chapter:

- x_i, y_i, z_i: The 16-byte state after the X, S, L operation, respectively, at round i.
- x_i^j: The state at round i whose position within a set or a sequence is given by j.
- $x_i[j]$: The j^{th} byte of the state x_i, where $j = 0, 1, \cdots, 15$, and the bytes are indexed from left to right.

- Δx_i, $\Delta x_i[j]$: The difference at state x_i, and byte $x_i[j]$, respectively.
- $X[K_i]$: xor addition of the i^{th} round key K_i.

The memory complexity of our attack is given in 16-byte states and the time complexity is evaluated in reduced round Kuznyechik encryptions. In the following sections, we give a preliminary security analysis of Kuznyechik and present the details of our MitM attack.

9.3 Security Analysis of Kuznyechik

The designers of Kuznyechik did not provide any security analysis of the cipher. Accordingly, we give our analysis of the cipher against some well known attacks.

9.3.1 Differential and Linear Cryptanalysis

Since Kuznyechik employs the same Sbox as the Russian hash function standard Streebog [135], we use the Sbox properties presented in [79]. The linear transformation of the cipher has an optimal branch number of 17. The maximum Sbox differential probability $= 8/256 = 2^{-5}$. Accordingly, the maximum probability of a differential characteristic over two rounds is $(2^{-5})^{17} = 2^{-85}$. Also, with an Sbox nonlinearity of 100, there is no linear approximation over two rounds with an input-output correlation larger than $((128-100)/128)^{17} \approx 2^{-37.27}$. Furthermore, let N_r denote the minimum number of differentially or linearly active Sboxes for r rounds, $r = 1, \cdots, 9$. Then, using the mixed integer linear programming approach proposed in [115], one can show that $N_r = 1, 17, 18, 34, 35, 51, 52, 68, 69$, for $r = 1, 2, \cdots, 9$, respectively. Consequently, given the block length of 128-bits and by noting the data complexity of differential and linear cryptanalysis, there is no useful differential or linear characteristic of length more than three rounds.

9.3.2 Related-key Cryptanalysis

This attack exploits either the slow diffusion or the symmetry in the key schedule. The Kuznyechik key schedule employs a Feistel structure with the same round function used in the encryption rounds. This round function is designed to cause fast and nonlinear diffusion between round keys. Also, eight

rounds of processing are used between round keys extraction which makes it infeasible to propagate and maintain a given relation between keys from two successive extractions.

9.3.3 Integral Cryptanalysis

Using a set of 2^8 chosen plaintexts which differ in one byte that takes all the 2^8 values and the remaining fifteen bytes are equal, results in a zero sum of all 256 cipher states at every byte position after two rounds. Accordingly, one can recover the third round key by guessing the key bytes independently, decrypting the corresponding 256 ciphers, and checking for the zero sum at the respective position. Once the last round key is recovered, one can peal this round off and repeat the attack to recover the key in the round before the last. The time complexity for recovering the last round key of the three rounds reduced cipher is $16 \times 2^8 \times 2^8 \approx 2^{20}$ and for recovering the whole master key is 2^{21}. Extending the attack to four rounds can be done by guessing the last round key and then launching the previous attack on the first three rounds to recover the third round key which increases the time complexity to $2^{20+128} = 2^{148}$.

9.3.4 Higher Order Differential Cryptanalysis

Since the algebraic degree of the Sbox is 7, then the degree of two consecutive rounds of the cipher is at most $7 \times 7 = 49$ and any 50-th (or higher) order derivative must be 0. One can append an additional round before these two rounds by choosing 2^{56} plaintext inputs with seven active Sboxes. The fourth round subkey bytes are then recovered independently with a time complexity of $16 \times 2^{56+8} = 2^{68}$.

In the following section, we give the details of our MitM attack on Kuznyechik.

9.4 A MitM Attack using Differential Enumeration on Kuznyechik

Generally, our attack divides the reduced Kuznyechik block cipher, C_K, into three parts, such that $C_K = C_{k_2} \circ C^m \circ C_{k_1}$, where C^m is the middle part of the cipher which exhibits a distinguishing property. The employed property is evaluated without the knowledge of the key bits used in these

middle rounds. Hence, correct round key candidates for k_1 and k_2 are checked if they verify this distinguishing property or not. Our middle distinguisher is a truncated differential characteristic such that, when a set of input states from a δ-*set* [44] is presented as its input, the set of each byte of the output state forms an ordered sequence.

Definition 1. (δ-*set* of Kuznyechick) *is a set of 256 states where one byte at a particular state takes all the* 2^8 *possible values and the rest of the 15 bytes are constants.*

In our MitM attack, we employ a distinguisher where the δ-*set* is presented at their input from the ciphertext side, thus, after partially decrypting it, we acquire the corresponding ordered sequence. We denote the δ-*set* at state x_i resulting from changing the byte at position j by δs^j, $j = 0, 1, \cdots, 15$, where

$$\delta s^j = \{x_i^0, x_i^1, \cdots, x_i^{255}\}.$$

We also denote the set of 255 differences at byte $x_{i-r}[k]$ which form the ordered sequence for an r round distinguisher by os^k, $k = 0, 1, \cdots, 15$, where

$$os^k = \{\Delta^1 x_{i-r}[k], \Delta^2 x_{i-r}[k], \cdots, \Delta^{255} x_{i-r}[k]\},$$

and $\Delta^l x_{i-r}[k] = x_{i-r}^0[k] \oplus x_{i-r}^l[k]$, for $l = 1, 2, \cdots, 255$. The correct ordered sequence os^k is evaluated by partially decrypting the 256 bytes which are different in the δ-*set* for r rounds. However, we compute all the possible sequences so that one does not need to know the key bits involved in this encryption process because we simply compute it using all the possible values of the involved parameters.

Our proposed 5-round MitM attack employs a three round distinguisher. Figure 9.3 depicts the differential path used in the attack in which we embed a $1 \rightarrow 16 \rightarrow 16 \rightarrow 1$ distinguisher that starts at x_5 and ends at x_2. The length of the distinguisher is restricted by the properties of optimal linear transformation used in the Kuznyechik round. Unlike the MixColumn transformation used in AES which works on independent columns leading to full state diffusion after two rounds, the linear

transformation L guaranties full diffusion in one round. As depicted in Figure 9.3, our δ-*set* is the set

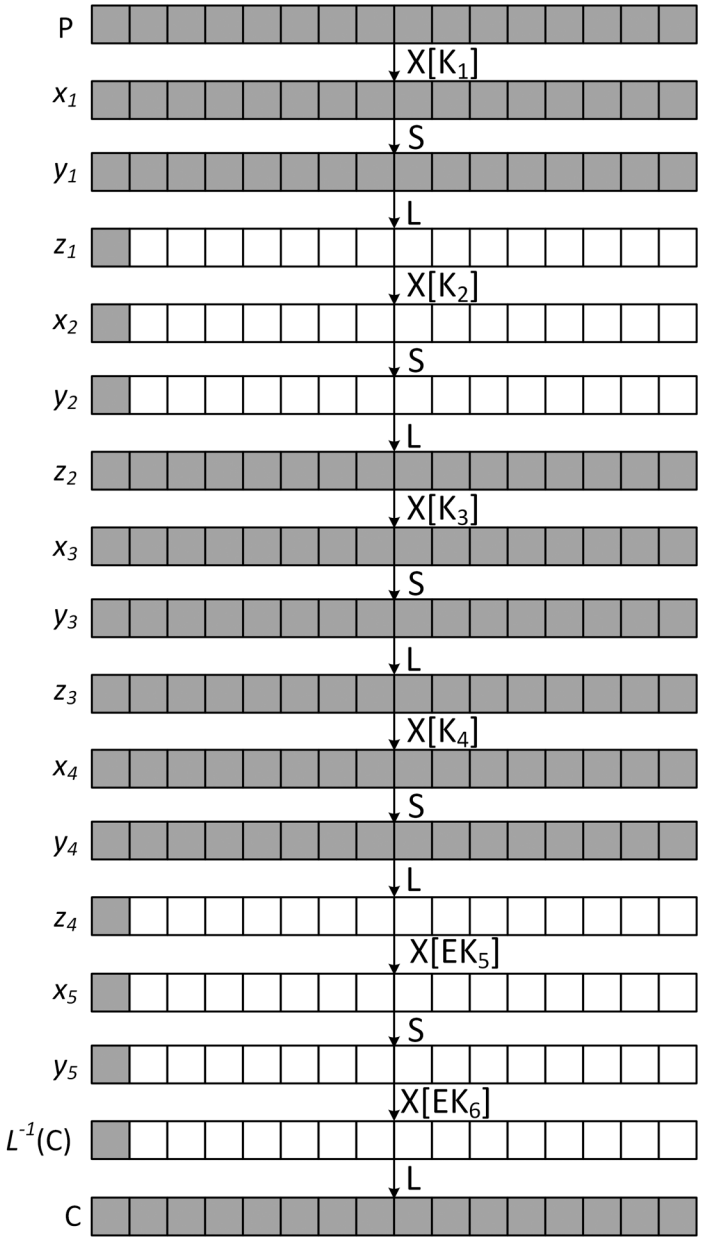

Figure 9.3: Differential path used in the 5-round attack.

of states resulting from changing the first byte at state x_5 and is given by:

$$\delta s^0 = \{x_5^0, x_5^1, \cdots, x_5^{255}\}.$$

The corresponding ordered sequence:

$$os^0 = \{\Delta^1 x_2[0], \Delta^2 x_2[0], \cdot, \Delta^{255} x_2[0]\}$$

is evaluated by the knowledge of the values of 33 bytes. More precisely, in addition to δs^0, given the values of 16 bytes at y_4, 16 bytes at y_3 and 1 byte at $y_2[0]$, the i^{th} element, $\Delta^i x_2[0]$ of the ordered sequence is computed as follows:

- Compute $\Delta^i x_5[0] = x_5^0[0] \oplus x_5^i[0]$ from δs^0.
- Linearly propagate $\Delta^i x_5[0]$ backwards and compute the value of $\Delta^i y_4$.
- Using the value of y_4 and $\Delta^i y_4$, pass the substitution layer with certainty and evaluate $\Delta^i x_4$.
- Linearly propagate $\Delta^i x_4$ backwards and compute the value of $\Delta^i y_3$.
- Using the value of y_3 and $\Delta^i y_3$, evaluate $\Delta^i x_3$
- Linearly propagate $\Delta^i x_3$ backwards through both $X[K_3]$ and L to evaluate $\Delta^i y_2[0]$.
- Using $\Delta^i y_2[0]$ and $y_2[0]$, compute $\Delta^i x_2[0]$.

However, by employing the rebound based differential enumeration technique [49], we deduce that if x_5^0 of δs^0 belongs to a pair of messages that follows the differential path in Figure 9.3, then the corresponding ordered sequence os^0 can have only 2^{152} values. Accordingly, a given ordered sequence can be computed by the knowledge of 19 byte parameters only. These parameters are $\Delta x_5[0]$, y_4, $y_2[0]$, and $\Delta y_2[0]$, where $\Delta x_5[0]$ and $\Delta y_2[0]$ denote the differences generated by a conforming message pair. In what follows, we give the details of the attack steps and explain how we evaluate the 2^{152} sequences from these 19 parameters.

9.4.1 Attack Procedure

The attack recovers the 128-bit first round key K_1 and one byte of $EK_6 = L^{-1}(K_6)$. The fact that K_1 is half the master key, K, enables us to recover the whole master key by exhaustively searching for the other half. The benefit of the extra knowledge of the recovered byte of EK_6 is limited to making the exhaustive search for the rest of the master key more efficient by early aborting the round keys generation process if the corresponding byte in $EK_6[0]$ does not match the one recovered by our attack. More precisely, the key schedule employs a large number of rounds between the generation

of sequential round keys, which leads to a very complex relation between them and renders any key bridging approaches useless.

The attack is composed of precomputation and online phases. In the precomputation phase, we iterate on all the values of the parameters required for evaluating the ordered sequence, and for each value, we deduce its corresponding 33 bytes values which are then used to generate the ordered sequence. We store all the sequences in a hash table. The online phase is further divided into data collection and filtration, and key recovery phases. In the data collection phase, we collect many pairs such that one of them satisfies the 5-round differential characteristics given in Figure 9.3. However, given the fact that our required ciphertext pairs are fully active, we employ an equivalent representation of the last round to enable efficient filtering by which we are certain that the obtained ciphertext pairs satisfy the lower two rounds of the differential characteristic. Finally, in the key recovery phase, for each of the obtained pairs, we compute the ordered sequences by deducing the first round key K_1 and guessing the first byte of EK_6. We then search for a match between the online computed sequence and the ones stored in the precomputed table, which enables the recovery of K_1 and $EK_6[0]$.

Precomputaion phase: In this phase, we construct a lookup table that contains the 2^{152} ordered sequences of the resulting 255 difference,

$$os^0 = \{\Delta^1 x_2[0], \Delta^2 x_2[0], \cdots, \Delta^{255} x_2[0]\},$$

from the $\delta s^0 = \{x_5^0, x_5^1, \cdots, x_5^{255}\}$. This stage is done by first iterating on the 2^{152} possible values for the 19 bytes $\Delta x_5[0]$, y_4, $y_2[0]$, and $\Delta y_2[0]$, and for each one of them, we deduce the possible values of the 33 original parameters using the rebound approach. Then, for each of them, we construct the ordered sequence of 255 differences. The procedure can be summarized as follows:

1. For each of the 2^{152} possible values of $\Delta x_5[0] \parallel y_4 \parallel y_2[0] \parallel \Delta y_2[0]$, evaluate the values of the 33 bytes required to compute the ordered sequence, which are $y_2[0]$, y_3, and y_4 as follows:

   - Linearly propagate $\Delta x_5[0]$ backwards to evaluate Δy_4.
   - Using Δy_4 and y_4, evaluate Δy_3.

- Compute Δx_3 by linearly propagating $\Delta y_2[0]$ through the linear transformation.
- Find x_3, such that $S(x_3) \oplus S(x_3 \oplus \Delta x_3) = \Delta y_3$. According to property 1 in Section 2, not all the 2^{152} differentials are possible, but the ones that are possible result in about 2^{16} solutions so we get one solution on average.
- Evaluate $y_3 = S(x_3)$.

2. The additional knowledge of the evaluated value of y_3 provides us with the values of the 33 bytes required to compute the 255 differences $\Delta^l x_2[0]$, $l = 1, 2, \cdots, 255$, of the ordered sequence as described at the beginning of this section.

3. Store all the generated sequences in a hash table.

Online phase: In this phase, we first find enough pairs of messages such that one of them conforms to the truncated differential characteristic in Figure 9.3. In this step we introduce a modification to the default process of data collection [49, 52]. More precisely, instead of collecting enough random pairs with full active states so that one of them satisfies the two $16 \rightarrow 1$ transitions through the linear transformation in rounds 1 and 4, we start data collection from the ciphertext side and employ an equivalent representation of the last round. During this stage, we commence by composing structures of the inverse linear transformation of ciphertext that have all the 2^8 possible values in one byte while the other bytes are constants. Accordingly, even though their corresponding ciphertext pairs have full active state, these differences guaranty the $16 \rightarrow 1$ transition through the linear transformation. Hence, we have to repeat this filtration stage enough times so that we satisfy only the probabilistic transition in round 1. A direct consequence of our modification is that instead of requiring 2^{240} pairs, the attack is applicable with 2^{120} pairs only, thus both the data and time requirements of the attack are lowered by this difference. The second step uses the found pairs to create a set of sequences and test them against the precomputed table to identify the correct K_1.

Data collection and filtration: In this step, we query the decryption oracle with structures of chosen ciphertexts to get enough pairs such that one conforms to the whole truncated differential path. Each structure is composed of 256 ciphertext, where the first byte after applying the inverse linear

transformation on them takes all the 256 values and the remaining fifteen bytes are equal. The process is described as follows:

1. To get one ciphertext structure, randomly pick the value of the rightmost fifteen bytes of $L^{-1}(C)$ and let the first byte take all the possible 256 values. This structure generates about $\frac{2^8 \times (2^8-1)}{2} \approx 2^{15}$ pairs. This step guaranties that all the corresponding (C, C') pairs in the structure conform to the $16 \rightarrow 1$ transition in round 5.

2. Query the decryption oracle for the plaintext pairs (P, P') corresponding to the ciphertext pairs generated in step 1. These pairs are not necessarily going to conform to the $16 \rightarrow 1$ transition in round 1, which happens with probability 2^{-120}.

3. Store the 2^8 plaintexts and their corresponding ciphertexts in a hash table.

4. To get one pair of plaintexts (P, P') that satisfy the $16 \rightarrow 1$ probabilistic transition, we need to try approximately 2^{120} pairs. Since, each structure provides 2^{15} pairs, one requires about 2^{105} structures, and hence the above steps are repeated 2^{105} times.

All in all, we ask for the decryption of $2^{105} \times 2^8 = 2^{113}$ chosen ciphertexts to get the required 2^{120} pairs.

Key recovery: The previous steps results in 2^{120} candidate pairs (P_i, C_i) and (P'_i, C'_i), for $i = 0, 1, \cdots, 2^{120} - 1$, with a plaintext difference, $\Delta P_i = P_i \oplus P'_i$, and a predetermined ciphertext difference, $\Delta C_i = C_i \oplus C'_i$. For each pair, we deduce the possible values of K_1 and guess the value of $EK_6[0]$ to compute a candidate ordered sequence and match it against the precomputed table, and thus determine the value of the right K_1. The following process describes the method adopted for the recovery of the first round key, and it is repeated for each plaintext pair (P_i, P'_i) and their corresponding ciphertext pair (C_i, C'_i).

1. Guess a value for $\Delta x_2[0]$, and linearly propagate it backwards to get the value of Δy_1.

2. Using the fact that $\Delta x_1 = \Delta P_i$, find the value of x_1 which provides a solution for $\Delta y_1 = S(x_1) \oplus S(x_1 \oplus \Delta x_1)$. According to property 1 in section 2, we get one solution on average.

3. Evaluate $K_1 = P_i \oplus x_1$. By repeating the previous two steps for all the possible guesses of $\Delta x_2[0]$, we get 2^8 candidate values for K_1.

4. For each candidate of the 2^8 values of K_1 and for each guess of the 2^8 guesses of $EK_6[0]$, use C_i to get the rest of the 255 ciphertexts C_i^j for $j = 1, 2, \cdots, 255$, corresponding to the δs^0 generated by C_i as follows:

   - The value of $x_5[0]$ which is the first byte of the first state in δs^0 is evaluated as follows:

   $$x_5[0] = S^{-1}(L^{-1}(C_i)[0] \oplus EK_6[0]).$$

   - The set of different values of $x_5[0]$ in the states of δs^0 has the following structure:

   $$\{x_5[0], x_5[0] \oplus \Delta_1, x_5[0] \oplus \Delta_2, \cdots, x_5[0] \oplus \Delta_{255}\},$$

   and $\Delta_j = j$ for $j = 1, 2, \cdots, 255$. Accordingly, we can evaluate the 255 values of $L^{-1}(C_i^j)[0]$ corresponding to the values of $x_5[0] \oplus \Delta_j$ by

   $$L^{-1}(C_i^j)[0] = S(x_5[0] \oplus \Delta_j) \oplus EK_6[0].$$

   - Get the difference $\Delta^j L^{-1}(C_i)[0] = L^{-1}(C_i^j)[0] \oplus L^{-1}(C_i)[0]$, and propagate it through the linear transformation to get the corresponding difference $\Delta^j C_i$. Finally, the required 255 values of C_i^j are evaluated by $C_i^j = C_i \oplus \Delta^j C_i$.

5. Get the 256 plaintexts $(P_i, P_i^1, \cdots, P_i^{255})$ corresponding to the ciphertexts generated in the previous step from the currently stored structure.

6. Using K_1, partially encrypt the plaintexts $(P_i, P_i^1, \cdots, P_i^{255})$ to get the 256 values of $\Delta^j x_2[0]$, which form the ordered sequence os^0.

7. Check if there is a match between the computed os^0 and the 2^{152} ordered sequences stored in the precomputed table. If there is a match, then exit with the candidate K_1 and $EK_6[0]$ as the right key, else we discard it with certainty.

The probability of a wrong key producing a valid 255 byte ordered sequence is given by $2^{152+120+16-2040} = 2^{-1752}$, which is negligible and can be relaxed. This fact allows us to present our partial sequence matching idea.

9.4.2 Complexity Analysis

The memory complexity of the attack is attributed to the precomputed table required for the storage of 2^{152} sequences of size 2040 bits each. Thus the memory requirements of the attack is given by $2^{152} \times 2040/128 \approx 2^{156}$ 128-bit states. That memory complexity can be reduced by a factor of 4 using the multiset encoding idea (cf. Appendix A in [49]), where 512-bits are used to store the required information of the 255 bytes in a sequence. The data complexity of the attack is due to the data collection step where we query the decryption oracle with 2^{113} chosen ciphertexts. The time complexity for recovering the first round key is dominated by the time required for partially encrypting the 256 values in a δ-*set* with all the 2^{16} key candidates for all the 2^{120} collected pairs. Accordingly, the time complexity of the attack $\approx 2^{(120+16+8)} \times 2/5 \approx 2^{143}$.

As it is fairly complex to deduce any relation between the recovered K_1 and $EK_6[0]$ that can aid us in the recovery of K_2, which is the second half of the master key, we are left with two options. First, with the knowledge of the recovered K_1, we can remove one round from the beginning, and repeat the attack on the following four rounds to recover K_2. Otherwise, our second option is to exhaustively search for K_2. Comparing the complexities of both options, we opt for the second one. Thus, the memory, data, and time complexities required for the recovery of the 256-bit KuznyechiK key are given by 2^{154}, 2^{113} and $2^{143} + 2^{128} \approx 2^{143}$, respectively. In what follows, we present the idea of partial sequence matching by which we reduce both the memory and time complexities of the attack.

9.4.3 Partial sequence matching

Our proposed 5-round attack has a time complexity of 2^{143}, which is affected by the number of partial encryptions/decryptions required to generate the $2^8 - 1$ differences in the ordered sequence from the δ-*set*. Accordingly, instead of partially encrypting the 2^8 values of the δ-*set* to get their corresponding ciphertexts, and then encrypting 2^8 plaintexts to evaluate the ordered sequence, we can

reduce the number of encryption/decryption operations to b, where $b < 2^8$ and denotes the number of differences stored in the ordered sequence. In other words, since the probability of error is so small, it can be relaxed such that we match b bytes of the 2^8 of the ordered sequence to identify the right key. More precisely, if we accept the error probability to be 2^{-32}, which is still negligible, the required number of bytes, b, is evaluated by $2^{-32} = 2^{120+16+152-8b}$. Hence, it is enough to match 40 bytes of the ordered sequence to identify a right key with an error probability of 2^{-32}. In the sequel, the memory complexity of the attack is reduced to $2^{152} \times (320/128) \approx 2^{153.3}$ states, and the time complexity is evaluated by $2^{120+16} \times 2^{5.3} \times 2/5 \approx 2^{140.3}$.

9.5 Conclusion

In this chapter, we have presented a MitM attack on the new Russian encryption standard, also known as Kuznyechik, using the idea of efficient differential enumeration. We have proposed an initial filtration stage which lowers the time complexity of the basic approach by a factor of 2^{120}. Instead of trying random data pairs such that the truncated differential path is satisfied probabilistically, we carefully compose ciphertext pairs so that the lower half of the path is conformed to with certainty. Additionally, we have adopted partial sequence matching, by which we store and match parts of the ordered sequences while maintaining a negligible probability of error which reduces both the memory and time complexities of the attack. Our attack on the 5-round reduced cipher has a memory complexity of $2^{153.3}$, a time complexity of $2^{140.3}$, and a data complexity of 2^{113} chosen ciphertext.

It should be noted that several improvements like key bridging techniques [49] for this class of attacks were possible on AES because of its relatively simple key schedule. This is unlikely to be the case for Kuznyechik, given the large number of rounds used in the generation of the round keys, which despite its conceptual simplicity leads to a very complex relation between successive round keys. Also, we note that 4 rounds of Kuznyechik can be broken using the integral attack (cf. section 5.6 and 5.7 in [24]) which was applied on Khazad.

Chapter 10

Summary and Future Research Directions

10.1 Summary of contributions

In what follows we briefly summarize the contributions of this thesis in the analysis of the Russian cryptographic hash function Streebog and block cipher Kuznyechik, and the Maelstrom-0 hash function.

In chapter 3, we have investigated the compression function of the Russian standardized hash function Streebog and its internal cipher with respect to rebound attacks. First, we analyzed the differential properties of the Streebog Sbox differential distribution table and showed how these properties affect the complexity of the rebound attack. As for the internal cipher, we have introduced differences in both the key schedule and message encryption and proposed a new message differential path such that a local collision is enforced every two rounds. Accordingly, the Sbox matching complexity which is caused by its differential bias is bypassed. As a result, a free-start 5-round collision and 7-round near collision examples for the internal cipher have been generated. Moreover, the compression function was investigated and we noted that the Streebog compression function key whitening round, which shifts the flow of the key generation process from the message encryption by one round enhances its resistance to free-start collision attacks. However, our results have demonstrated that the Streebog compression function is vulnerable to semi free-start 7.75 round collision, 8.75 and 9.75 round near collision attacks and an example for a 4.75 round 50-byte near colliding message pair has been presented.

In our second analysis of Streebog in chapter 4, we have investigated the structural integral properties of reduced-round versions of the Streebog compression function and its internal permutation. Specifically, we presented forward and backward higher order integrals that can be used to distinguish 4 and 3.5 rounds, respectively. Using the start from the middle approach, we combined the two proposed integrals to get 6.5-round and 7.5-round distinguishers for the internal permutation and 6-round and 7-round distinguishers for the compression function. Moreover, following the simplified representation of AES [62], we have extended our original work in [11] to 8 rounds by considering a new representation of the 12-round Streebog internal cipher. In this representation, the internal cipher is viewed as a sequence of six *super rounds* proceeded and followed by a transpose operation where each *super round* replaces two consecutive regular rounds.

In chapter 5, we have analyzed the security of Streebog and its compression function, assessing their resistance to the meet-in-the-middle preimage attacks. Specifically, we presented a pseudo preimage attack on the compression function reduced to 5 out of 12 rounds by employing the partial matching and initial structure concepts. In particular, we proposed an execution separation for the compression function that balances the degrees of freedom in both execution directions with their corresponding matching probability. Furthermore, we extended the attack by one round using a guess and determine approach, which allows us to guess parts of the state that belong to one execution. Finally, using a multicollision attack, the compression function pseudo preimage attacks were used to produce 5 and 6-round hash function preimages.

Continuing with the rebound attacks, in chapter 6, we proposed a malicious version of Streebog. By exploiting the randomness of the independent round constants and the number of rounds of the compression function, we were able to efficiently generate collisions for the compression function. Specifically, we first employed the rebound attack approach to find three pairs of messages and keys that satisfy a specific three 4-round differential paths independently. Then using the freedom of five out of the twelve round constants, we connected the three obtained solutions to generate collisions for the twelve round compression function. Finally, we tuned the last constant of the compression function to adjust its output after the feedforward to cancel the

effect of the counter addition of the following compression function call, and appended another identical colliding message pair. Hence, we were able to generate a two block messages 2^2 multicollision structure where two of them have the same modular sum and thus a collision at the output of the hash function. While previous works have stated that compression function collisions are not sufficient to generate hash function collision in constructions that incorporate a checksum, our results proved that this is not the case with Streebog. Our attack has a practical complexity and is verified by example.

In chapter 7, we have presented a differential fault analysis attack on Streebog. The attack considers the compression function when operating with secret inputs which is the default setting when the function is used in a message authentication code (MAC) scheme. In our analysis, we have proposed a two-stage attack using the one-bit fault model where the attacker is able to cause a bit flip at a chosen or random byte in the internal state of the function. Employing a specific property of the Streebog Sbox and by observing several correct and faulty compression function outputs, the first stage of the attack bypasses the final feedforward and retrieves the state of the internal cipher. The second stage of the attack recovers one of the round keys which enables the recovery of both the chaining value and message block of the attacked compression function. Lastly, we analyzed the Streebog hash function in different MAC settings and showed how to use our DFA attack to recover the secret MAC key of simple prefix and secret-IV MACs, HMAC, and NMAC.

In chapter 8, we have investigated the security of Maelstrom-0 and its compression function, assessing their resistance to the meet-in-the-middle preimage attacks. The Maelstrom-0 hash function is proposed as a lighter alternative to its predecessor the ISO standard Whirlpool. Firstly, we have presented a pseudo preimage attack on the 6-round reduced compression function. Then, we proposed a four stage approach which combines a 2-block multicollision attack [56, 57] with a second meet-in-the-middle attack to bypass the effect of the 3CM checksum used in the finalization step, and generate preimages of the 6-round reduced Maelstrom-0 hash function.

In chapter 9, we have presented a meet-in-the-middle attack on the new Russian encryption standard Kuznyechik using the idea of efficient differential enumeration. Unlike AES, Kuznyechik

employs an optimal diffusion transformation applied to the whole state, where one byte difference results in a full active state with certainty after one round. Consequently, we constructed a three round distinguisher to recover 16-bytes of the master key of the reduced 5-round cipher. In our analysis, we adopted an equivalent representation of the last round which allows us to efficiently select ciphertext pairs that satisfy the lower half of the differential path used in the attack with certainty. Hence, our attack is considered in the chosen ciphertext setting. Additionally, we have presented partial sequence matching, by which we generate, store, and match parts of the ordered sequence while maintaining negligible probability of error.

10.2 Future work

In what follows, we propose some avenues for possible extension of our work:

With the ongoing CAESAR competition [41] which is scheduled to announce a final portfolio of authenticated encryption schemes in 2017, it is interesting to investigate how the cryptanalytic approaches presented in this thesis can be used to analyze the CAESAR submissions. Indeed, most of the remaining submissions are not fully provably secure. Hence, their security arguments are completely attributed to their ongoing cryptanalysis. Additionally, the area of integrated authenticated encryption algorithms is considered relatively new as authenticated ciphers are less developed than other cryptographic primitives such as block ciphers and hash functions.

The Streebog compression function key whitening round K_N creates asymmetry in the key and message flows which limits our proposed approach of creating multiple local collisions for the compression function. In our work [7], we maintain a sparse differential path by enforcing local collisions through keeping similar differential patterns in both the message and key states. Due to the key whitening round K_N, our approach is only successful on the internal cipher. Accordingly, a possible research direction in the context of the collision analysis of Streebog is to study the extension of the internal cipher attack to generate collisions of the hash function.

An interesting extension to our malicious adaptation of Streebog is to investigate other malicious notions such as preimage and second preimage backdoors. To begin, the definitions of preimage

and second preimage backdoors should be clarified. More precisely, our proposed collision backdoor is based on having one colliding message block pair that is acquired at the design stage of the hash function by carefully choosing the constants. Later, this colliding message pair can be used to generate hash function collisions of various messages by inserting it in these messages. However, in the context of a preimage backdoor, how can one benefit from designing the hash function such that the preimage of a specific digest is known? and can one design a malicious hash function that allows the recovery of the message from its digest with a complexity lower than the generic attack? Answering these questions remains an open problem and provides research challenges for future work. From the other perspective, given the fact that we were able to generate nothing up my sleeve constants in our backdoored Streebog, probing hash functions for the existence of such backdoors is an interesting topic for research, which is significantly important especially after the recent Snowden revelations.

Bibliography

[1] GOST 28147-89. Information Processing Systems. Cryptographic Protection. Cryptographic Transformation Algorithm. *(In Russian)*.

[2] The National Hash Standard of the Russian Federation GOST R 34.11-2012. Russian Federal Agency on Technical Regulation and Metrology report, 2012. `http://www.tc26.ru/en/standard/gost/GOST_R_34_11-2012_eng.pdf`.

[3] The National Standard of the Russian Federation GOST R 34._-20_. Russian Federal Agency on Technical Regulation and Metrology report, 2015. `http://www.tc26.ru/en/standard/draft/ENG_GOST_R_bsh.pdf`.

[4] ABDELKHALEK, A., ALTAWY, R., TOLBA, M., AND YOUSSEF, A. M. Meet-in-the-middle attacks on reduced-round Hierocrypt-3. In *LATINCRYPT* (2015), vol. 9230 of *Lecture Notes in Computer Science*, Springer, pp. 187–203.

[5] ALBERTINI, A., AUMASSON, J.-P., EICHLSEDER, M., MENDEL, F., AND SCHLÄFFER, M. Malicious hashing: Eve's variant of SHA-1. In *SAC* (2014), A. Joux and A. Youssef, Eds., vol. 8781 of *Lecture Notes in Computer Science*, Springer, pp. 1–19.

[6] ALTAWY, R., DUMAN, O., AND YOUSSEF, A. M. Fault analysis of Kuznyechik. In *CTCrypt* (2015), pp. 302–3017. Available at: http://eprint.iacr.org/2015/347.

[7] ALTAWY, R., KIRCANSKI, A., AND YOUSSEF, A. M. Rebound attacks on Stribog. In *ICISC* (2013), H.-S. Lee and D.-G. Han, Eds., vol. 8565 of *Lecture Notes in Computer Science*, Springer, pp. 175–188.

[8] ALTAWY, R., KIRCANSKI, A., AND YOUSSEF, A. M. Second order collision for the 42-step reduced DHA-256 hash function. *Information Processing Letters 113* (2013), 764–770.

[9] ALTAWY, R., TOLBA, M., AND YOUSSEF, A. M. A higher order key partitioning attack with application to LBlock. In *Codes, Cryptology, and Information Security* (2015), S. E. Hajji, A. Nitaj, C. Carlet, and E. M. Souidi, Eds., vol. 9084 of *Lecture Notes in Computer Science*, Springer, pp. 215–227.

[10] ALTAWY, R., AND YOUSSEF, A. M. Differential sieving for 2-step matching meet-in-the-middle attack with application to LBlock. In *LightSec* (2014), T. Eisenbarth and E. Öztürk, Eds., vol. 8898 of *Lecture Notes in Computer Science*, Springer, pp. 126–139.

[11] ALTAWY, R., AND YOUSSEF, A. M. Integral distinguishers for reduced-round Stribog. *Information Processing Letters 114*, 8 (2014), 426 – 431.

[12] ALTAWY, R., AND YOUSSEF, A. M. Preimage attacks on reduced-round Stribog. In *AFRICACRYPT* (2014), D. Pointcheval and D. Vergnaud, Eds., vol. 8469 of *Lecture Notes in Computer Science*, Springer, pp. 109–125.

[13] ALTAWY, R., AND YOUSSEF, A. M. Second preimage analysis of Whirlwind. In *Insrypt* (2014), D. Lin, M. Yung, and J. Zhou, Eds., vol. 8957 of *Lecture Notes in Computer Science*, Springer, pp. 311–328.

[14] ALTAWY, R., AND YOUSSEF, A. M. Differential fault analysis of Streebog. In *Information Security Practice and Experience* (2015), J. Lopez and Y. Wu, Eds., vol. 9065 of *Lecture Notes in Computer Science*, Springer, pp. 35–49.

[15] ALTAWY, R., AND YOUSSEF, A. M. A meet in the middle attack on reduced round Kuznyechik. *IEICE Transactions 98-A*, 10 (2015), 2194–2198.

[16] ALTAWY, R., AND YOUSSEF, A. M. Preimage analysis of the Maelstrom-0 hash function. In *Security, Privacy, and Applied Cryptography Engineering* (2015), R. S. Chakraborty, P. Schwabe, and J. A. Solworth, Eds., vol. 9354 of *Lecture Notes in Computer Science*, Springer, pp. 113–126.

[17] ALTAWY, R., AND YOUSSEF, A. M. Watch your constants: malicious Streebog. *IET Information Security 9*, 6 (2015), 328–333.

[18] AOKI, K., GUO, J., MATUSIEWICZ, K., SASAKI, Y., AND WANG, L. Preimages for step-reduced SHA-2. In *ASIACRYPT* (2009), M. Matsui, Ed., vol. 5912 of *Lecture Notes in Computer Science*, Springer, pp. 578–597.

[19] AOKI, K., AND SASAKI, Y. Meet-in-the-middle preimage attacks against reduced SHA-0 and SHA-1. In *CRYPTO* (2009), S. Halevi, Ed., vol. 5677 of *Lecture Notes in Computer Science*, Springer, pp. 70–89.

[20] AOKI, K., AND SASAKI, Y. Preimage attacks on one-block MD4, 63-step MD5 and more. In *SAC* (2009), R. M. Avanzi, L. Keliher, and F. Sica, Eds., vol. 5381 of *Lecture Notes in Computer Science*, Springer, pp. 103–119.

[21] AUMASSON, J.-P. Eve's SHA3 candidate: malicious hashing. Online article, 2011. `https://131002.net/data/papers/Aum11a.pdf`.

[22] AUMASSON, J.-P. Cryptographic backdooring, 2014. `https://131002.net/data/talks/backdooring_nsc14.pdf`.

[23] BANIK, S., MAITRA, S., AND SARKAR, S. A differential fault attack on the Grain family of stream ciphers. In *CHES* (2012), E. Prouff and P. Schaumont, Eds., vol. 7428 of *Lecture Notes in Computer Science*, Springer, pp. 122–139.

[24] BARRETO, P., AND RIJMEN, V. The Khazad Legacy-Level Block Cipher. In First Open NESSIE Workshop, KU-Leuven, 2000. Submission to NESSIE.

[25] BELLARE, M., CANETTI, R., AND KRAWCZYK, H. Keying hash functions for message authentication. In *Advances in Cryptology CRYPTO 96* (1996), N. Koblitz, Ed., vol. 1109 of *Lecture Notes in Computer Science*, Springer, pp. 1–15.

[26] BERTONI, G., DAEMEN, J., PEETERS, M., AND VAN ASSCHE, G. Keccak sponge function family main document. *Submission to NIST (Round 2)* (2009).

[27] BIHAM, E., CARMELI, Y., AND SHAMIR, A. Bug attacks. In *CRYPTO* (2008), D. Wagner, Ed., vol. 5157 of *Lecture Notes in Computer Science*, Springer, pp. 221–240.

[28] BIHAM, E., AND SHAMIR, A. Differential cryptanalysis of DES-like cryptosystems. In *CRYPTO* (1990), A. Menezes and S. A. Vanstone, Eds., vol. 537 of *Lecture Notes in Computer Science*, Springer, pp. 2–21.

[29] BIHAM, E., AND SHAMIR, A. Differential fault analysis of secret key cryptosystems. In *CRYPTO* (1997), J. Kaliski, BurtonS., Ed., vol. 1294 of *Lecture Notes in Computer Science*, Springer, pp. 513–525.

[30] BIRYUKOV, A., KHOVRATOVICH, D., AND NIKOLIC, I. Distinguisher and related-key attack on the full AES-256. In *CRYPTO* (2009), S. Halevi, Ed., vol. 5677 of *Lecture Notes in Computer Science*, Springer, pp. 231–249.

[31] BIRYUKOV, A., NIKOLIĆ, I., AND ROY, A. Boomerang attacks on BLAKE-32. In *FSE* (2011), A. Joux, Ed., vol. 6733 of *Lecture Notes in Computer Science*, Springer, pp. 218–237.

[32] BOGDANOV, A., KHOVRATOVICH, D., AND RECHBERGER, C. Biclique cryptanalysis of the full AES. In *ASIACRYPT* (2011), D. H. Lee and X. Wang, Eds., vol. 7073 of *Lecture Notes in Computer Science*, Springer, pp. 344–371.

[33] BOGDANOV, A., AND RECHBERGER, C. A 3-subset meet-in-the-middle attack: Cryptanalysis of the lightweight block cipher KTANTAN. In *SAC* (2011), A. Biryukov, G. Gong, and D. Stinson, Eds., vol. 6544 of *Lecture Notes in Computer Science*, Springer, pp. 229–240.

[34] BONEH, D., DEMILLO, R., AND LIPTON, R. On the importance of checking cryptographic protocols for faults. In *EUROCRYPT* (1997), W. Fumy, Ed., vol. 1233 of *Lecture Notes in Computer Science*, Springer, pp. 37–51.

[35] BOUILLAGUET, C., DERBEZ, P., DUNKELMAN, O., FOUQUE, P.-A., KELLER, N., AND RIJMEN, V. Low-data complexity attacks on AES. *IEEE Transactions on Information Theory 58*, 11 (2012), 7002–7017.

[36] CANETTI, R., GOLDREICH, O., AND HALEVI, S. The random oracle methodology, revisited. *Journal of the ACM (JACM) 51* (July 2004), 557–594.

[37] CANTEAUT, A., FUHR, T., NAYA-PLASENCIA, M., PAILLIER, P., REINHARD, J.-R., AND VIDEAU, M. A unified indifferentiability proof for permutation- or block cipher-based hash functions. Cryptology ePrint Archive, Report 2012/363, 2012. http://eprint.iacr.org/2012/363.

[38] CANTEAUT, A., NAYA-PLASENCIA, M., AND VAYSSIRE, B. Sieve-in-the-middle: Improved MITM attacks. In *CRYPTO* (2013), R. Canetti and J. Garay, Eds., vol. 8042 of *Lecture Notes in Computer Science*, Springer, pp. 222–240.

[39] CHANG, S.-J., PERLNER, R., BURR, W. E., TURAN, M. S., KELSEY, J. M., PAUL, S., AND BASSHAM, L. E. *Third-round report of the SHA-3 cryptographic hash algorithm competition*. 2012.

[40] CHANG, S., PERLNER, R., BURR, W.E., TURAN, M., KELSEY, J., PAUL, S. AND BASSHAM, L.E. Third-round report of the SHA-3 cryptographic hash algorithm competition. `http://nvlpubs.nist.gov/nistpubs/ir/2012/NIST.IR.7896.pdf`.

[41] COMPETITIONS.CR.YP.TO. CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness, 2013.

[42] COURBON, F., LOUBET-MOUNDI, P., FOURNIER, J. J., AND TRIA, A. Adjusting laser injections for fully controlled faults. In *Constructive Side-Channel Analysis and Secure Design* (2014), E. Prouff, Ed., Lecture Notes in Computer Science, Springer, pp. 229–242.

[43] DAEMEN, J., KNUDSEN, L., AND RIJMEN, V. The block cipher SQUARE. In *FSE* (1997), E. Biham, Ed., vol. 1267 of *Lecture Notes in Computer Science*, Springer, pp. 149–165.

[44] DAEMEN, J., AND RIJMEN, V. AES proposal: Rijndael, 1998.

[45] DAEMEN, J., AND RIJMEN, V. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer, 2002.

[46] DAMGÅRD, I. B. A Design Principle for Hash Functions. In *CRYPTO* (1990), G. Brassard, Ed., vol. 435 of *Lecture Notes in Computer Science*, Springer, pp. 416–427.

[47] DEMIRCI, H., AND SELÇUK, A. A meet-in-the-middle attack on 8-round AES. In *FSE* (2008), K. Nyberg, Ed., vol. 5086 of *Lecture Notes in Computer Science*, Springer, pp. 116–126.

[48] DEMIRCI, H., TAŞKN, I., ÇOBAN, M., AND BAYSAL, A. Improved meet-in-the-middle attacks on AES. In *INDOCRYPT* (2009), B. Roy and N. Sendrier, Eds., vol. 5922 of *Lecture Notes in Computer Science*, Springer, pp. 144–156.

[49] DERBEZ, P., FOUQUE, P.-A., AND JEAN, J. Improved key recovery attacks on reduced-round AES in the single-key setting. In *EUROCRYPT* (2013), T. Johansson and P. Nguyen, Eds., vol. 7881 of *Lecture Notes in Computer Science*, Springer, pp. 371–387.

[50] DIFFIE, W., AND HELLMAN, M. Exhaustive cryptanalysis of the NBS Data Encryption Standard. *Computer 10*, 6 (1977), 74–84.

[51] DINUR, I., DUNKELMAN, O., AND SHAMIR, A. Improved attacks on full GOST. In *FSE* (2012), A. Canteaut, Ed., vol. 7549 of *Lecture Notes in Computer Science*, Springer, pp. 9–28.

[52] DUNKELMAN, O., KELLER, N., AND SHAMIR, A. Improved single-key attacks on 8-round AES-192 and AES-256. In *ASIACRYPT* (2010), M. Abe, Ed., vol. 6477 of *Lecture Notes in Computer Science*, Springer, pp. 158–176.

[53] FILHO, D., BARRETO, P., AND RIJMEN, V. The Maelstrom-0 hash function. In *VI Brazilian Symposium on Information and Computer Systems Security* (2006).

[54] FILIOL, E. Malicious cryptography techniques for unreversable (malicious or not) binaries. *CoRR abs/1009.4000* (2010).

[55] FISCHER, W., AND REUTER, C. Differential fault analysis on Grøstl. In *IEEE workshop on Fault Diagnosis and Tolerance in Cryptography* (2012), pp. 44–54.

[56] GAURAVARAM, P., AND KELSEY, J. Cryptanalysis of a class of cryptographic hash functions. Cryptology ePrint Archive, Report 2007/277, 2007. `http://eprint.iacr.org/`.

[57] GAURAVARAM, P., AND KELSEY, J. Linear-XOR and additive checksums dont protect Damgård-Merkle hashes from generic attacks. In *CT-RSA* (2008), T. Malkin, Ed., vol. 4964 of *Lecture Notes in Computer Science*, Springer, pp. 36–51.

[58] GAURAVARAM, P., AND KELSEY, J. Linear-XOR and additive checksums dont protect damgård-merkle hashes from generic attacks. In *Topics in Cryptology CT-RSA 2008* (2008), T. Malkin, Ed., vol. 4964 of *Lecture Notes in Computer Science*, Springer, pp. 36–51.

[59] GAURAVARAM, P., KELSEY, J., KNUDSEN, L. R., AND THOMSEN, S. On hash functions using checksums. *International Journal of Information Security 9*, 2 (2010), 137–151.

[60] GAURAVARAM, P., KNUDSEN, L. R., MATUSIEWICZ, K., MENDEL, F., RECHBERGER, C., SCHLÄFFER, M., AND THOMSEN, S. S. Grøstl a SHA-3 candidate. *NIST submission* (2008).

[61] GAURAVARAM, P., MILLAN, W., DAWSON, E., AND VISWANATHAN, K. Constructing secure hash functions by enhancing Merkle-Damgård construction. In *ACISP* (2006), L. Batten and R. Safavi-Naini, Eds., vol. 4058 of *Lecture Notes in Computer Science*, Springer, pp. 407–420.

[62] GILBERT, H. A simplified representation of AES. In *ASIACRYPT* (2014), P. Sarkar and T. Iwata, Eds., vol. 8873 of *Lecture Notes in Computer Science*, Springe, pp. 200–222.

[63] GILBERT, H., AND PEYRIN, T. Super-Sbox Cryptanalysis: Improved attacks for AES-like permutations. In *FSE* (2010), S. Hong and T. Iwata, Eds., vol. 6147 of *Lecture Notes in Computer Science*, Springer, pp. 365–383.

[64] GIRAUD, C. DFA on AES. In *AES* (2005), H. Dobbertin, V. Rijmen, and A. Sowa, Eds., vol. 3373 of *Lecture Notes in Computer Science*, Springer, pp. 27–41.

[65] GUO, J., JEAN, J., LEURENT, G., PEYRIN, T., AND WANG, L. The usage of counter revisited: Second-preimage attack on new Russian standardized hash function. In *SAC* (2014), A. Joux and A. Youssef, Eds., vol. 8781 of *Lecture Notes in Computer Science*, Springer, pp. 195–211.

[66] GUO, J., LING, S., RECHBERGER, C., AND WANG, H. Advanced meet-in-the-middle preimage attacks: First results on full Tiger, and improved results on MD4 and SHA-2. In *ASIACRYPT* (2010), M. Abe, Ed., vol. 6477 of *Lecture Notes in Computer Science*, Springer, pp. 56–75.

[67] HALEVI, S., AND MICALI, S. Practical and provably-secure commitment schemes from collision-free hashing. In *CRYPTO* (1996), N. Koblitz, Ed., vol. 1109, pp. 201–215.

[68] HAO, Y., BAI, D., AND LI, L. A meet-in-the-middle attack on round-reduced mCrypton using the differential enumeration technique. In *Network and System Security* (2014), M. Au, B. Carminati, and C.-C. Kuo, Eds., vol. 8792 of *Lecture Notes in Computer Science*, Springer, pp. 166–183.

[69] HARBERT, T. New king of security algorithms crowned. *IEEE Spectrum 49* (2012), 12–13.

[70] HEMME, L., AND HOFFMANN, L. Differential fault analysis on the SHA1 compression function. In *IEEE workshop on Fault Diagnosis and Tolerance in Cryptography* (2011), pp. 54–62.

[71] HONG, D., KOO, B., AND SASAKI, Y. Improved preimage attack for 68-step HAS-160. In *ICISC* (2009), D. Lee and S. Hong, Eds., vol. 5984 of *Lecture Notes in Computer Science*, Springer, pp. 332–348.

[72] IETF. GOST R 34.11-2012: Hash Function, 2013. (RFC6896).

[73] INDESTEEGE, S. The Lane hash function. Submission to NIST (2008). Avalabile at: http://www.cosic.esat.kuleuven.be/publications/article-1181.pdf.

[74] ISOBE, T. A single-key attack on the full GOST block cipher. In *FSE* (2011), A. Joux, Ed., vol. 6733 of *Lecture Notes in Computer Science*, Springer, pp. 290–305.

[75] JEAN, J., AND FOUQUE, P.-A. Practical near-collisions and collisions on round-reduced ECHO-256 compression function. In *FSE* (2011), A. Joux, Ed., vol. 6733 of *Lecture Notes in Computer Science*, Springer, pp. 107–127.

[76] JOUX, A. Multicollisions in iterated hash functions. application to cascaded constructions. In *CRYPTO* (2004), M. Franklin, Ed., vol. 3152 of *Lecture Notes in Computer Science*, Springer, pp. 306–316.

[77] KALISKI, B. Pkcs# 1: Rsa encryption version 1.5.

[78] KANG, J., JEONG, K., SUNG, J., AND HONG, S. Differential fault analysis on HAS-160 compression function. In *Computer Science and its Applications* (2012), S.-S. Yeo, Y. Pan, Y. S. Lee, and H. B. Chang, Eds., vol. 203 of *Lecture Notes in Electrical Engineering*, Springer, pp. 97–105.

[79] KAZYMYROV, O., AND KAZYMYROVA, V. Algebraic aspects of the russian hash standard GOST R 34.11-2012. In *CTCrypt* (2013), pp. 160–176. Available at: http://eprint.iacr.org/2013/556.

[80] KECCAK TEAM. "Strengths of Keccak - Design and security". `http://keccak.noekeon.org/`. Last Accessed: 2014-12-2.

[81] KERCKHOFFS, A. La cryptographie militaire. *Journal des sciences militaires IX* (January 1883), 5–83.

[82] KERRY, C. F. Digital Signature Standard (DSS).

[83] KIM, C., AND QUISQUATER, J.-J. New differential fault analysis on AES key schedule: Two faults are enough. In *Smart Card Research and Advanced Applications* (2008), G. Grimaud and F.-X. Standaert, Eds., vol. 5189 of *Lecture Notes in Computer Science*, Springer, pp. 48–60.

[84] KIRCANSKI, A., ALTAWY, R., AND YOUSSEF, A. M. A heuristic for finding compatible differential paths with application to HAS-160. In *ASIACRYPT* (2013), K. Sako and P. Sarkar, Eds., vol. 8270 of *Lecture Notes in Computer Science*, Springer, pp. 464–483.

[85] KIRCANSKI, A., AND YOUSSEF, A. M. Differential Fault Analysis of Rabbit. In *Selected Areas in Cryptography* (2009), M. J. J. Jr., V. Rijmen, and R. Safavi-Naini, Eds., vol. 5867 of *Lecture Notes in Computer Science*, Springer, pp. 197–214.

[86] KNUDSEN, L. Non-random properties of reduced-round Whirlpool. NESSIE public report, 2002. NES/DOC/UIB/WP5/017/1.

[87] KNUDSEN, L., AND RIJMEN, V. Known-key distinguishers for some block ciphers. In *ASIACRYPT* (2007), K. Kurosawa, Ed., vol. 4833 of *Lecture Notes in Computer Science*, Springer, pp. 315–324.

[88] KNUDSEN, L., AND WAGNER, D. Integral cryptanalysis. In *FSE* (2002), J. Daemen and V. Rijmen, Eds., vol. 2365 of *Lecture Notes in Computer Science*, Springer, pp. 112–127.

[89] KNUDSEN, L. R. Truncated and higher order differentials. In *FSE* (1995), B. Preneel, Ed., vol. 1008 of *Lecture Notes in Computer Science*, Springer, pp. 196–211.

[90] KNUDSEN, L. R., AND RIJMEN, V. Known-key distinguishers for some block ciphers. In *ASIACRYPT* (2007), K. Kurosawa, Ed., vol. 4833 of *Lecture Notes in Computer Science*, Springer, pp. 315–324.

[91] KNUDSEN, L. R., ROBSHAW, M. J. B., AND WAGNER, D. Truncated differentials and skipjack. In *CRYPTO* (1999), M. J. Wiener, Ed., vol. 1666 of *Lecture Notes in Computer Science*, Springer, pp. 165–180.

[92] KÖLBL, S., AND MENDEL, F. Practical attacks on the Maelstrom-0 compression function. In *ACNS* (2011), J. Lopez and G. Tsudik, Eds., vol. 6715 of *Lecture Notes in Computer Science*, Springer, pp. 449–461.

[93] KÖLBL, S., AND RECHBERGER, C. Practical attacks on AES-like cryptographic hash functions. In *Latincrypt* (2014), D. F. Aranha and A. Menezes, Eds., vol. 8895 of *Lecture Notes in Computer Science*, Springer, pp. 259–273.

[94] LAI, X., AND MASSEY, J. Hash function based on block ciphers. In *EUROCRYPT* (1992), R. A. Rueppel, Ed., vol. 658 of *Lecture Notes in Computer Science*, Springer, pp. 55–70.

[95] LAMBERGER, M., MENDEL, F., RECHBERGER, C., RIJMEN, V., AND SCHLÄFFER, M. Rebound distinguishers: Results on the full Whirlpool compression function. In *ASIACRYPT* (2009), M. Matsui, Ed., vol. 5912 of *Lecture Notes in Computer Science*, Springer, pp. 126–143.

[96] LI, L., JIA, K., AND WANG, X. Improved single-key attacks on 9-round AES-192/256. In *FSE* (2014), C. Cid and C. Rechberger, Eds., vol. 8540 of *Lecture Notes in Computer Science*, Springer.

[97] LI, R., LI, C., AND GONG, C. Differential fault analysis on SHACAL-1. In *IEEE workshop on Fault Diagnosis and Tolerance in Cryptography* (2009), pp. 120–126.

[98] LUCKS, S. The saturation attack a bait for Twofish. In *FSE* (2002), M. Matsui, Ed., vol. 2355 of *Lecture Notes in Computer Science*, Springer, pp. 1–15.

[99] MA, B., LI, B., HAO, R., AND LI, X. Improved cryptanalysis on reduced-round GOST and Whirlpool hash function. In *Applied Cryptography and Network Security* (2014), I. Boureanu, P. Owesarski, and S. Vaudenay, Eds., vol. 8479 of *Lecture Notes in Computer Science*, Springer, pp. 289–307.

[100] MATSUI, M. Linear cryptoanalysis method for DES cipher. In *EUROCRYPT* (1993), T. Helleseth, Ed., vol. 765 of *Lecture Notes in Computer Science*, Springer, pp. 386–397.

[101] MATUSIEWICZ, K., NAYA-PLASENCIA, M., NIKOLI, I., SASAKI, Y., AND SCHLÄFFER, M. Rebound attack on the full lane compression function. In *ASIACRYPT* (2009), M. Matsui, Ed., vol. 5912 of *Lecture Notes in Computer Science*, Springer, pp. 106–125.

[102] MATYUKHIN, D., RUDSKOY, V., AND SHISHKIN, V. A perspective hashing algorithm. In *RusCrypto* (2010). *(In Russian)*.

[103] MATYUKHIN, D., AND SHISHKIN, V. Some methods of hash functions analysis with application to the GOST P 34.11-94 algorithm. *Mat. Vopr. Kriptogr 3* (2012), 71–89. *(In Russian)*.

[104] MENDEL, F., PEYRIN, T., RECHBERGER, C., AND SCHLÄFFER, M. Improved cryptanalysis of the reduced Grøstl compression function, ECHO permutation and AES block cipher. In *Selected Areas in Cryptography* (2009), M. J. Jacobson Jr, V. Rijmen, and R. Safavi-Naini, Eds., vol. 5867 of *Lecture Notes in Computer Science*, Springer, pp. 16–35.

[105] MENDEL, F., PRAMSTALLER, N., AND RECHBERGER, C. A (second) preimage attack on the GOST hash function. In *FSE* (2008), K. Nyberg, Ed., vol. 5086 of *Lecture Notes in Computer Science*, Springer, pp. 224–234.

[106] MENDEL, F., PRAMSTALLER, N., RECHBERGER, C., KONTAK, M., AND SZMIDT, J. Cryptanalysis of the GOST hash function. In *CRYPTO* (2008), D. Wagner, Ed., vol. 5157 of *Lecture Notes in Computer Science*, Springer, pp. 162–178.

[107] MENDEL, F., RECHBERGER, C., SCHLFFER, M., AND THOMSEN, S. S. The rebound attack: Cryptanalysis of reduced Whirlpool and Grøstl. In *FSE* (2009), O. Dunkelman, Ed., vol. 5665 of *Lecture Notes in Computer Science*, Springer, pp. 260–276.

[108] MENDEL, F., RECHBERGER, C., SCHLFFER, M., AND THOMSEN, S. S. Rebound attacks on the reduced Grøstl hash function. In *CT-RSA* (2010), J. Pieprzyk, Ed., vol. 5985 of *Lecture Notes in Computer Science*, Springer, pp. 350–365.

[109] MENDEL, F., RIJMEN, V., AND SCHLFFER, M. Collision attack on 5 rounds of Grøstl. In *FSE* (2014), C. Cid and C. Rechberger, Eds., vol. 8540 of *Lecture Notes in Computer Science*, Springer, pp. 509–521.

[110] MENEZES, A. J., VAN OORSCHOT, P. C., AND VANSTONE, S. A. *Handbook of applied cryptography*. CRC press, 2010.

[111] MERKLE, R. C. One Way Hash Functions and DES. In *CRYPTO* (1990), G. Brassard, Ed., vol. 435 of *Lecture Notes in Computer Science*, Springer, pp. 428–446.

[112] MINIER, M., PHAN, R., AND POUSSE, B. Integral distinguishers of some SHA-3 candidates. In *CANS* (2010), S.-H. Heng, R. N. Wright, and B.-M. Goi, Eds., vol. 6467 of *Lecture Notes in Computer Science*, Springer, pp. 106–123.

[113] MINIER, M., PHAN, R. C., AND POUSSE, B. Distinguishers for ciphers and known key attack against Rijndael with large blocks. In *AFRICACRYPT* (2009), B. Preneel, Ed., vol. 5580 of *Lecture Notes in Computer Science*, Springer, pp. 60–76.

[114] MINIER, M., AND THOMAS, G. Integral distinguisher on Grøstl-512 v3. In *Indocrypt* (2013), To appear in Springer LNCS.

[115] MOUHA, N., CANNIÈRE, C. D., INDESTEEGE, S., AND PRENEEL, B. Finding collisions for a 45-step simplified HAS-V. In *WISA* (2009), H. Y. Youm and M. Yung, Eds., vol. 5932 of *Lecture Notes in Computer Science*, Springer, pp. 206–225.

[116] NATIONAL BUREAN OF STANDARDS. Data Encryption Standard, U.S. Department of Commerce, Federal Information Processing Standards 46 (1977).

[117] NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. Advanced Encryption Standard (AES) (FIPS PUB 197), 2001.

[118] NIST. Announcing request for candidate algorithm nominations for a new cryptographic hash algorithm (SHA-3) family. In *Federal Register* (November 2007), vol. 72(212). Available at: http://csrc.nist.gov/groups/ST/hash/documents/FR_Notice_Nov07.pdf.

[119] PATARIN, J., AND GOUBIN, L. Trapdoor one-way permutations and multivariate polynomials. In *ICICS* (1997), Y. Han, T. Okamoto, and S. Qing, Eds., vol. 1334 of *Lecture Notes in Computer Science*, Springer, pp. 356–368.

[120] PATERSON, K. G. Imprimitive permutation groups and trapdoors in iterated block ciphers. In *FSE* (1999), L. Knudsen, Ed., vol. 1636 of *Lecture Notes in Computer Science*, Springer, pp. 201–214.

[121] PEYRIN, T. Improved differential attacks for ECHO and Grøstl. In *CRYPTO* (2010), T. Rabin, Ed., vol. 6223 of *Lecture Notes in Computer Science*, Springer, pp. 370–392.

[122] POSCHMANN, A., LING, S., AND WANG, H. 256 bit standardized crypto for 650 GE GOST revisited. In *CHES* (2010), S. Mangard and F.-X. Standaert, Eds., vol. 6225 of *Lecture Notes in Computer Science*, Springer, pp. 219–233.

[123] PRENEEL, B., GOVAERTS, R., AND VANDEWALLE, J. Hash functions based on block ciphers: A synthetic approach. In *CRYPTO* (1993), D. R. Stinson, Ed., vol. 773 of *Lecture Notes in Computer Science*, Springer, pp. 368–378.

[124] PRENEEL, B., AND VAN OORSCHOT, P. C. On the security of iterated message authentication codes. *IEEE Transactions on Information Theory 45*, 1 (1999), 188–199.

[125] RESCORLA, E. Diffie-Hellman key agreement method.

[126] RIJMEN, V., AND BARRETO, P. S. L. M. The Whirlpool hashing function. *NISSIE submission* (2000).

[127] RIJMEN, V., AND PRENEEL, B. A family of trapdoor ciphers. In *FSE* (1997), E. Biham, Ed., vol. 1267 of *Lecture Notes in Computer Science*, Springer, pp. 139–148.

[128] RIJMEN, V., TOZ, D., AND VARC, K. Rebound attack on reduced-round versions of JH. In *FSE* (2010), S. Hong and T. Iwata, Eds., vol. 6147 of *Lecture Notes in Computer Science*, Springer, pp. 286–303.

[129] RUDSKOY, V. Note on Streebog constants origin, 2015. `http://tk26.ru/en/ISO_IEC/streebog/streebog_constants_eng.pdf`.

[130] SASAKI, Y. Meet-in-the-middle preimage attacks on AES hashing modes and an application to Whirlpool. In *FSE* (2011), A. Joux, Ed., vol. 6733 of *Lecture Notes in Computer Science*, Springer, pp. 378–396.

[131] SASAKI, Y., AND WANG, L. Comprehensive study of integral analysis on 22-round LBlock. In *ICISC2012* (2013), T. Kwon, M.-K. Lee, and D. Kwon, Eds., vol. 7839 of *Lecture Notes in Computer Science*, Springer, pp. 156–169.

[132] SASAKI, Y., WANG, L., WU, S., AND WU, W. Investigating fundamental security requirements on Whirlpool: Improved preimage and collision attacks. In *ASIACRYPT* (2012), X. Wang and K. Sako, Eds., vol. 7658 of *Lecture Notes in Computer Science*, Springer, pp. 562–579.

[133] SCHNEIER, B. The NSA is breaking most encryption on the internet. `https://www.schneier.com/blog/archives/2013/09/the\_nsa\_is\_brea.html`, [Online; published September-2013].

[134] SHANNON, C. E. Communication theory of secrecy systems. *Bell System Tech. J. 28* (1949), 656–715.

[135] SHISHKIN, V., DYGIN, D., LAVRIKOV, I., MARSHALKO, G., RUDSKOY, V., AND TRIFONOV, D. Low-Weight and Hi-End: Draft Russian Encryption Standard. In *CTCrypt* (2014), pp. 183–188.

[136] SKOROBOGATOV, S., AND ANDERSON, R. Optical fault induction attacks. In *CHES* (2003), B. Kaliski, e. Ko, and C. Paar, Eds., vol. 2523 of *Lecture Notes in Computer Science*, Springer, pp. 2–12.

[137] STEVENS, M., LENSTRA, A., AND DE WEGER, B. Chosen-Prefix Collisions for MD5 and Colliding X.509 Certificates for Different Identities. In *EUROCRYP* (2007), M. Naor, Ed., vol. 4515 of *Lecture Notes in Computer Science*, Springer, pp. 1–22.

[138] TC26. Provision on the open research competition on hash function GOST R 34.11-2012, 2013. `http://www.tc26.ru/en/research/polozhenie/`.

[139] TUNSTALL, M., MUKHOPADHYAY, D., AND ALI, S. Differential fault analysis of the Advanced Encryption Standard using a single fault. In *Information Security Theory and Practice* (2011), C. Ardagna and J. Zhou, Eds., vol. 6633 of *Lecture Notes in Computer Science*, Springer, pp. 224–233.

[140] WAGNER, D. The boomerang attack. In *Fast Software Encryption* (1999), L. Knudsen, Ed., vol. 1636 of *Lecture Notes in Computer Science*, Springer, pp. 156–170.

[141] WAGNER, D. A generalized birthday problem. In *CRYPTO* (2002), M. Yung, Ed., vol. 2442 of *Lecture Notes in Computer Science*, Springer, pp. 288–304.

[142] WANG, G. L. Collision attack on the full extended MD4 and pseudo-preimage attack on RIPEMD. *Journal of Computer Science and Technology 28* (2013), 129–143.

[143] WANG, X., YIN, Y. L., AND YU, H. Finding collisions in the full SHA-1. In *CRYPTO* (2005), V. Shoup, Ed., vol. 3621 of *Lecture Notes in Computer Science*, Springer, pp. 17–36.

[144] WANG, X., AND YU, H. How to break MD5 and other hash functions. In *EUROCRYPT* (2005), R. Cramer, Ed., vol. 3494 of *Lecture Notes in Computer Science*, Springer, pp. 19–35.

[145] WANG, Z., YU, H., AND WANG, X. Cryptanalysis of GOST R hash function. *Information Processing Letters 114*, 12 (2014), 655–662.

[146] WAYNER, P. *Digital Cash (2nd Ed.): Commerce on the Net*. Academic Press Professional, Inc., 1997.

[147] WIKIPEDIA. Bullrun (decryption program) — Wikipedia, the free encyclopedia, 2014. [Online; accessed 22-October-2014].

[148] WIKIPEDIA. Dual_EC_DRBG — Wikipedia, the free encyclopedia, 2014. [Online; accessed 22-October-2014].

[149] WU, H. The hash function JH, 2011. Avalabile at:http://www3.ntu.edu.sg/home/wuhj/research/jh/jh-round3.pdf.

[150] WU, S., FENG, D., WU, W., GUO, J., DONG, L., AND ZOU, J. (Pseudo) preimage attack on round-reduced Grøstl hash function and others. In *FSE* (2012), A. Canteaut, Ed., vol. 7549 of *Lecture Notes in Computer Science*, Springer, pp. 127–145.

[151] YOUNG, A., AND YUNG, M. *Malicious cryptography: Exposing cryptovirology*. John Wiley & Sons, 2004.

[152] ZOU, J., WU, W., AND WU, S. Cryptanalysis of the round-reduced GOST hash function. In *Information Security and Cryptology* (2014), D. Lin, S. Xu, and M. Yung, Eds., Lecture Notes in Computer Science, Springer, pp. 309–322.