Cryptanalysis and Design of Symmetric Primitives

Inauguraldissertation
zur Erlangung des akademischen Grades
Doctor rerum naturalium (Dr.rer.nat.)

der Bauhaus-Universität Weimar
an der Fakultät Medien

vorgelegt von
Michael Gorski

Weimar, Dezember 2010

Gutachter:	Prof. Dr. Stefan Lucks Bauhaus-Universität Weimar, Weimar (Deutschland)
	Dr. Orr Dunkelman Weizmann Institute of Science, Rehovot (Israel)

Ehrenwörtliche Erklärung

Ich erkläre hiermit ehrenwörtlich, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen direkt oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet.

Weitere Personen waren an der inhaltlich-materiellen Erstellung der vorliegenden Arbeit nicht beteiligt. Insbesondere habe ich hierfür nicht die entgeltliche Hilfe von Vermittlung- bzw. Beratungsdiensten (Promotionsberater oder anderer Personen) in Anspruch genommen. Niemand hat von mir unmittelbar oder mittelbar geldwerte Leistungen für Arbeiten erhalten, die im Zusammenhang mit dem Inhalt der vorgelegten Dissertation stehen.

Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form einer anderen Prüfungsbehörde vorgelegt.

Ich versichere, dass ich nach bestem Wissen die reine Wahrheit gesagt und nichts verschwiegen habe.

Ort, Datum Unterschrift

Acknowledgements

First of all I want to thank all the people who helped me to make this thesis successful. I would like to thank my PhD advisors Prof. Dr. Stefan Lucks and Dr. Orr Dunkelman for their support and comments they gave to me helping to force my understanding in cryptography. They inspired and supported me while exploring the world of symmetric cryptography. I also want to thank Prof. Dr. Johannes Blömer for giving me an interesting topic for my diploma thesis that initialized my research on symmetric cryptography and forced my interest in this topic.

I also want to thank Ewan Fleischmann who discussed and developed many ideas with me many times. He always spent his time to enjoy discussions of different aspects in cryptography with me. Furthermore, I thank Christian Forler, who joined the research group of Prof. Dr. Stefan Lucks in 2009. He had huge impact on the success of our TWISTER hash function proposal as well as a fast and efficient implementation of the algorithm. A special thank to Thomas Peyrin for interesting discussions and motivations for my thesis. Thanks to all the people in the crypto community that have given me helpful comment on the papers I made. Last but not least I want to thank my parents for their everlasting support and encouragement in all the things I have done, the things I do and all the things that I will do.

Michael Gorski

Abstract

This thesis focuses on the cryptanalysis and the design of block ciphers and hash functions. The thesis starts with an overview of methods for cryptanalysis of block ciphers which are based on differential cryptanalysis. We explain these concepts and also several combinations of these attacks. We propose new attacks on reduced versions of ARIA and AES. Furthermore, we analyze the strength of the internal block ciphers of hash functions. We propose the first attacks that break the internal block ciphers of Tiger, HAS-160, and a reduced round version of SHACAL-2.

The last part of the thesis is concerned with the analysis and the design of cryptographic hash functions. We adopt a block cipher attack called slide attack into the scenario of hash function cryptanalysis. We then use this new method to attack different variants of GRINDAHL and RADIOGATÚN.

Finally, we propose a new hash function called TWISTER which was designed and proposed for the SHA-3 competition. TWISTER was accepted for round one of this competition. Our approach follows a new strategy to design a cryptographic hash function. We also describe several attacks on TWISTER and discuss the security issues concerning these attack on TWISTER.

Zusammenfassung

Der Schwerpunkt dieser Dissertation liegt in der Analyse und dem Design von Blockchiffren und Hashfunktionen. Die Arbeit beginnt mit einer Einführung in Techniken zur Kryptoanalyse von Blockchiffren. Wir beschreiben diese Methoden und zeigen wie man daraus neue Techniken entwickeln kann, welche zu stärkeren Angriffen führen.

Im zweiten Teil der Arbeit stellen wir eine Reihe von Angriffen auf eine Vielzahl von Blockchiffren dar. Wir haben dabei Angriffe auf reduzierte Versionen von ARIA und dem AES entwickelt. Darüber hinaus präsentieren wir im dritten Teil Angriffe auf interne Blockchiffren von Hashfunktionen. Wir entwickeln Angriffe, welche die internen Blockchiffren von Tiger und HAS-160 auf volle Rundenanzahl brechen. Die hier vorgestellten Angriffe sind die ersten dieser Art. Ein Angriff auf eine reduzierte Version von SHACAL-2 welcher fast keinen Speicherbedarf hat, wird ebenfalls vorgestellt.

Der vierte Teil der Arbeit befasst sich mit den Design und der Analyse von kryptographischen Hashfunktionen. Wir habe einen Slide Angriff, eine Technik welche aus der Analyse von Blockchiffren bekannt ist, im Kontext von Hashfunktionen zur Anwendung gebracht. Dabei präsentieren wir verschiedene Angriffe auf GRINDAHL und RADIOGATÚN. Aufbauend auf den Angriffen des zweiten und dritten Teils dieser Arbeit stellen wir eine neue Hashfunktion vor, welche wir TWISTER nennen. TWISTER wurde für den SHA-3 Wettbewerb entwickelt und ist bereits zur ersten Runde angenommen.

Contents

List of Tables

List of Figures

Notations

The following notations are used in this thesis:

$\oplus$: bitwise XOR operation

$\wedge$: bitwise AND operation

$\vee$: bitwise OR operation

$\neg$: bitwise complement operation

$X^{\ll i}$: shift of the word X by i bits to the left

$X^{\gg i}$: shift of the word X by i bits to the right

$X^{\lll k}$: bit-rotation of the word X by k positions to the left.

$X^{\ggg k}$: bit-rotation of the word X by k positions to the right.

$\boxplus$: addition modulo 2^b, where b is the word size in bits

$\boxminus$: subtraction modulo 2^b, where b is the word size in bits

$\boxtimes$: multiplication modulo 2^b, where b is the word size in bits

e_i : a b-bit word with zeros in all bits except for bit i, $(0 \leq i \leq b-1)$

$e_{i_1,\ldots,i_l}$: $e_{i_1} \oplus \cdots \oplus e_{i_l}$

$a \ll b$: a is much smaller than b

$f^{-1}(\cdot)$: is the inverse function to $f(\cdot)$

The bit positions follow the little endian convention, i.e., for a b-bit word bit $b-1$ is the most significant bit and bit 0 is the least significant bit. The bit positions are labeled as $b-1, b-2, \ldots, 1, 0$.

Chapter 1

Introduction

1.1 Cryptography

During the last years, the amount of data transferred through the internet follows an exponential growth. The wide spreading of the world wide web in more and more parts of society leads to an increase in the possibility of accessing data from everywhere. The barriers for data transportation disappear, meaning that it gets easier to share, contribute, and access data due to the increasing number of connections that support these actions.

For example, today it is quite common to use online-banking from all over the world to manage bank accounts everywhere. Having access to our personal bank accounts whenever and wherever we want, raises the need for assuring that this data can only be used by us. Thus, data is often encrypted using methods designed to give rights of handling only to the persons of desire. The creator of some data, therefore, applies a cryptographic algorithm to the data to protect them against modification, or access by a third party. He then shares some kind of secret information with the receiver in an encrypted way, which only the receiver should be able to decrypt. An adversary which observes an encrypted communication between the bank and the customer should not be able to gain any relevant information concerning the customer's bank account. Hence, cryptography must guarantee on the one hand that the adversary will not succeed in doing so and on the other hand that the encryption does not reduce the usability of the system while increasing the customers' access barriers to their accounts.

Cryptography always faces this tradeoff between security and speed. It is quite easy to design a slow cryptographic system that offers a high degree of security, i.e., an adversary cannot gain any useful information from observing a large set of transactions. Such a system will be unusable in practice and thus will never be used. It is also easy to design a fast insecure cryptographic system. The difficulty is to find the best equilibrium between both extremes, which depends heavily on the applications, needs for security, and the duration the security must be maintained.

Cryptography deals with a broad range of security applications such as protocols, signatures, and encryptions. One of the most known cryptographic applications is encryption. Two types of encryption can be addressed, *public key* encryption and *secret key* encryption. In public key encryption, a public and a private key are being used, where the public key is accessible to anyone for encryption, whereas the private key is only attainable to the person that is allowed to decrypt the data. In contrast, secret key encryption uses only one key, which is accessible to all authorized persons which share the secret key.

The most important symmetric primitives which are regarded in this theses can be grouped into four main primitives which are block ciphers, stream ciphers, hash functions and message authentication codes. A block cipher is a keyed permutation that processes plaintexts of fixed length with a secret key and encrypts them into corresponding ciphertexts. To decrypt the ciphertexts, one has to apply the inverse of the encryption algorithm using the same secret key as for the encryption. Stream ciphers are divided into two types synchronous and self-synchronizing stream ciphers. In a synchronous stream cipher, a stream of pseudo-random characters is generated independently of the plaintext and the ciphertext. Then, these characters are combined with the plaintext or the ciphertext. In a self-synchronizing stream cipher, the receiver will automatically synchronize with the key stream generator after receiving some ciphertext characters which makes it easier to recover if characters are dropped or added to the message stream due to transmission errors. A hash function is a primitive that maps an arbitrary sized input value to a digest of fixed size. A message authentication code computes a fingerprint of a given message using a secret key.

The Data Encryption Standard (DES) [117], which was the American block cipher standard and widely accepted by everybody since 1977, was replaced in a competition due to some security and performance concerns. During this process, initiated by the U.S. National Institute of Standards and Technology (NIST) in 1997, block ciphers received much attention. NIST accepted 15 candidates for the first round of the competition which were designed by researchers as well as from companies. In 1999, five finalists were selected for a more thorough analysis RC6, Rijndael, Twofish, MARS and Serpent. The competition ended in 2001, when the Rijndael block cipher was selected to be the Advanced Encryption Standard (AES).

Block ciphers have a fixed sized input and output length. In order to process messages of arbitrary length, block ciphers are used in *modes of operation*. A mode of operation specifies how to process concatenated blocks of the message. Well known modes of operation are *electronic codebook* (ECB), *cipher-block chaining* (CBC), *cipher feedback* (CFB), *output feedback* (OFB), and *counter mode* (CTR).

A hash function takes an input of arbitrary length and generates an output of fixed length. It can be seem as a digital fingerprint of the data. Hash functions are used in a broad range of applications, e.g., digital signatures, databases, or for checksums. Nowadays, hash functions gained much attention due to recent breaks of the some widely used hash functions such as MD5, SHA-0 and SHA-1. NIST announced a competition for a new Secure Hash Algorithm (SHA-3), which started at the end of 2008. Compared to a block cipher, the design of a good hash function is much more complicated since it contains no secret key which is unknown to the adversary. Every part of the computation is known and can be observed and controlled in some way by the adversary.

1.2 Outline

The contribution of this thesis is outlined in the following, were we give a summary of each chapter.

Chapter 2 treats the preliminaries that are needed for the remainder of this thesis. We explain attacks on block ciphers like differential cryptanalysis, related-key attacks, as well as some of their extensions.

Chapter 3 focuses on the boomerang and the rectangle attacks on reduced versions of ARIA and the Advanced Encryption Standard (AES). *The results of this chapter have been published in [55, 60, 64, 71].*

Chapter 4 focuses on boomerang and rectangle attacks on the internal block ciphers of hash functions. We present an attacks on SHACAL-2 for reduced version and we also propose attacks which break the Tiger block cipher and 77-round HAS-160 in encryption mode. *The results of this chapter have been published in [47, 61, 62].*

Chapter 5 addresses a new technique for attacking hash functions called slide attacks. This technique was originally proposed for block cipher cryptanalysis and we adopted it to a class of hash function which are designed using the sponge framework. We also present a candidate hash function for the SHA-3 competition called TWISTER. The TWISTER hash function family was designed to resist cryptanalytic techniques presented in Chapter 2 and Chapter 4 and 5. It offers a simple and easy to analyze framework for building a hash function. *The results of this chapter have been published in [50, 51, 52, 53, 56, 72].*

Chapter 6 concludes the thesis and discusses open problems and further research directions.

As presented above, preliminary versions of the results obtained in this thesis were published before in [47, 50, 51, 52, 53, 55, 56, 60, 64, 61, 62, 71, 72]. Other results in symmetric cryptography that are not considered in this thesis and have been published during my studies, can be found in [54, 57, 58, 59, 63, 65].[1]

[1] These papers do not fit in the context of this thesis, so we do not include them. Also some of them were published after finishing the thesis.

Chapter 2

Preliminaries

In the first parts of this chapter we describe what block ciphers are and methods for cryptanalysis of block ciphers. These techniques are used to analyze the security of such ciphers in the later parts of the thesis. In the second part of this chapter we introduce hash functions and show how they can be built from a block cipher.

2.1 A Block Cipher

We start with a description of a block cipher, which is a symmetric key primitive. A block cipher accepts data strings of n bits and keys of k bits. When the data is a plaintext, the block cipher encrypts the data into a ciphertext, and when the data is the ciphertext, the block cipher decrypts the data back into a plaintext. In other words a block cipher, $E : \{0,1\}^n \times \{0,1\}^k \to \{0,1\}^n$ is a keyed permutation of n-bit where the key K is of length k bits. A plaintext P is encrypted under K as

$$E_K(P) = C,$$

where C is the corresponding ciphertext. Most block ciphers follow an iterative design, which repeats the same round transformation many times. A *Substitution Permutation Network (SPN)* is a common form of building such a block cipher. Usually a round of a SPN consists of three different layers substitution, permutation, and key mixing in each round.

Substitution Layer The substitution layer consists of so called *S-boxes*. An S-box is a non linear transformation, i.e., the output bits cannot be represented as a linear function of the input bits. It can be implemented by a lookup table.

Permutation Layer The permutation layer simply permutes the bit or byte positions or applies a linear transformation (e.g., MixColumns which is part of the permutation layer of AES, see Section 3.2.1). These operations can be applied bit-wise, byte-wise or word-wise depending on the block cipher.

Key Mixing Layer In the key mixing layer a *subkey*, which is obtained from the *key schedule algorithm*, is inserted into the internal state (e.g., via bitwise $\oplus$ or $\boxplus$ mod 2^n). A key schedule is an algorithm which is used for the derivation of the subkeys from the key.

These layers are often combined in a round function $f_{K_i}(\cdot)$ (under the subkey K_i) which is applied several times, depending on the number of rounds, r, of the block cipher. Let $f_{K_i}^{-1}(\cdot)$ be the inverse round function. Furthermore, let $g_{K_i}(\cdot)$ (under the subkey K_i) be a key mixing layer and $g_{K_i}^{-1}(\cdot)$ its inverse. The key is usually used before the first round to guarantee that the first round is applied to values unknown to the adversary while assuming that the key mixing layer is the last operation in $f_{K_i}(\cdot)$. Thus, the encryption can be written as

$$E_K(P) = f_{K_r} \circ f_{K_{r-1}} \circ \cdots \circ f_{K_2} \circ f_{K_1} \circ g_{K_0}(P) = C,$$

where $K_0, K_1, K_2, \ldots, K_r$ are the subkeys derived from the key K, by the *key schedule* algorithm. The decryption of C into P is performed by the inversion of the encryption process as

$$D_K(C) = g_{K_0}^{-1} \circ f_{K_1}^{-1} \circ f_{K_2}^{-1} \circ \cdots \circ f_{K_{r-1}}^{-1} \circ f_{K_r}^{-1}(C) = P.$$

2.2 Preliminaries

For a given small number, e.g., $\lceil (k+1)/n \rceil$, of plaintext-ciphertext pairs which are encrypted under K, the key K can be recovered in average time of 2^{k-1} trial encryptions using *exhaustive search*, i.e., trying to encrypt all $\lceil (k+1)/n \rceil$ plaintexts under all the possible keys and verifying that the ciphertexts correspond to the given respective ciphertexts.

Generally, an attack can be used for a key recovery or to distinguish the cipher from an ideal cipher [139]. The ideal cipher is a model, where we assume the existence of a publicly accessible block cipher with an n-bit input and a k-bit key. This block cipher is a set of randomly chosen permutations from n bits to n bits. Using an ideal cipher the adversary cannot predict any information on the key observing the plaintexts and ciphertexts. Thus, cryptographers try to design their block ciphers in a way that best imitates the ideal cipher. In the following we discuss possible attack scenarios that can be used for this showing that the cipher is not ideal.

Attacking a block cipher depends heavily on the assumptions made on the adversary and his knowledge of the cipher. In the standard assumptions it is common that the adversary has access to all the data transmitted over the insecure channel. Due to Kerckhoffs' principle [89] we also assume that he knows all the details of the underlying encryption function except for the secret key. From a standard point of view this seems to be a bit unusual, since one would expect not to make the cryptosystem public. Kerckhoffs' principle is applied in modern cryptography since it is hard to hide the cryptosystem. Furthermore, a lack of security could be found more easily in a publicly known algorithm, since more people have the capability to analyze the algorithm. Historically, this was always proved to be the case. Assume that the key of a publicly known algorithm gets public. In such a system changing the key might be very cheap compared with the scenario where a hidden algorithm gets publicly known and weaknesses are found. So it would be much more expensive to replace the whole algorithm instead of just changing the key in a well studied algorithm.

There are several types of attacks:

- **ciphertext only attack** – only the ciphertexts are available to the adversary.
- **known plaintext attack** – plaintext-ciphertext pairs are available for some plaintexts, which are usually assumed to be random. The adversary has no control over the plaintext pairs.
- **chosen plaintext attack** – the adversary may choose plaintexts and obtain their corresponding ciphertexts.
- **adaptive chosen plaintext attack** – the adversary may choose plaintexts *adaptively* and obtain their corresponding ciphertexts.
- **chosen ciphertext attack** – the adversary may choose ciphertexts and obtain their corresponding plaintexts.
- **adaptive chosen ciphertext attack** – the adversary may choose ciphertexts *adaptively* and obtain their corresponding plaintexts.
- **related-key attack** – the adversary has some knowledge about the relation between different keys that are used in the encryption or he may be able to choose the relation. (see Section 2.4)

Related-key attack models are always combined with one of the other scenarios. Each scenario defines the access the adversary has to the cipher while performing the attack.

Usually, an attack can be separated into three steps which are *data acquisition*, *data filtering*, and *analysis*.

- **Data Acquisition Step** In this step the data, which is needed to mount the attack is generated through the scenarios listed above.
- **Data Filtering Step** In this step the adversary reduces the amount of data which is analyzed to reduce the complexity of the analysis and improve the success probability. This might improve the success probability, since some useless data is usually discarded.
- **Analysis Step** The data remaining after the data filtering step is used either to distinguish the cipher from an ideal cipher or for key recovery.

2.3 Differential Cryptanalysis

Differential cryptanalysis [20, 22] was introduced by Biham and Shamir in 1990 as a technique for block cipher cryptanalysis. It can also be used to attack stream ciphers and hash functions as well. Biham and Shamir have demonstrated the strength of the attack by breaking reduced versions of DES. They later extended the attack to break DES faster than exhaustive search [21]. Many other block ciphers were shown to be vulnerable to this attack.

Differential cryptanalysis is a method used for key recovery and distinguishing. It uses pairs of plaintexts or ciphertexts, which have a certain relation in the pair. The propagation of the pairs' relation, which is called a *difference* is computed through the components of the cipher. The adversary predicts this propagation and uses it to find the correct keys.

It is quite common that the subkeys are injected by an XOR function during each round. In the following we assume this kind of injection. Assume that X and X' are some intermediate values during the encryption process of the plaintexts P and P'. We assume that each plaintext is encrypted under the same key K, where K_i indicates the corresponding subkey of round i due to the key schedule. Furthermore, let $Y = X \oplus K_i$ and $Y' = X' \oplus K_i$, respectively, be the intermediate encryption values after the subkey injection. In the following we often write ΔX for $X \oplus X'$ and ΔY for $Y \oplus Y'$, respectively. The difference between two intermediate encryption values after the subkey injection is

$$\begin{aligned} \Delta Y &= Y \oplus Y' \\ &= (X \oplus K_i) \oplus (X' \oplus K_i) \\ &= X \oplus X' = \Delta X. \end{aligned}$$

For sake of simplicity we only discuss the case where the subkeys are inserted by an

XOR operation.[1] While regarding the difference between two intermediate values, this difference is independent of the subkey K_i as long as the same subkeys are used in both encryptions. Therefore, the adversary can predict the value of ΔY, but he cannot compute the actual values of Y and Y' due to the unknown subkey.

Let A be a linear transformation, an let $Z = A \cdot X$, and X' is processed as $Z' = A \cdot X'$, respectively. The difference ΔZ of the pair of intermediate values, Z and Z', can then be computed as

$$\Delta Z = A \cdot \Delta X.$$

When a nonlinear operation is performed, i.e., $Z = F(X)$, for a nonlinear F, then most of the time $\Delta X \neq 0$ prevents determining the difference ΔZ with probability 1. When $\Delta X = 0$, then $\Delta Z = 0$ as well. We note that for some specific (and rare) combinations of $\Delta X \neq 0$ and F, one can predict the output difference ΔZ with probability 1.

Assuming that X is chosen uniformly over all possible plaintexts and for a given ΔX, the adversary can compute the distribution of all possible output differences ΔZ, which are listed in a *difference distribution table*. A difference distribution table displays how often a certain pair of input and output differences occurs for the S-box. It is therefore used to compute the probability of these transitions.

The adversary uses this table to form a *differential characteristic*. A one-round differential characteristic is an input and an output differences to one round of the cipher with the probability that the input difference is transformed to the output difference. One can concatenate one-round differential characteristics to form differential characteristics over many rounds.

Finally, the differential characteristic ends in a ciphertext difference (or some difference before the last round) ΔC for a given plaintext difference ΔP. Assuming that the propagations of the differences for each non-linear component are independent of each other, the overall probability p of a differential characteristic is given by the product of each one-round differential characteristic it consists of.

A *differential* is an input and output difference linked with its probability. The probability of a differential can be computed as the sum of all the differential characteristics which share the same input and output difference.

Given a plaintext difference $\Delta P = \alpha$ which is transformed into a ciphertext difference $\Delta C = \beta$ with probability p (or in short we just write $\alpha \to \beta$ for the differential) it is defined as

$$\Pr_{P,K}[E_K(P) \oplus E_K(P \oplus \alpha) = \beta] = p,$$

[1] We note that differential cryptanalysis can be applied if the subkeys are injected with an operation different than $\oplus$.

where $\Pr_{P,K}[\cdot]$ is an average probability over the choices of P and K.

Given a differential with probability p, the adversary needs about $O(p^{-1})$ plaintext pairs which all have difference α for a successful attack. The larger the probability p is, the fewer pairs are needed for the attack to succeed. The adversary then often counts the number of pairs which produce the predicted output difference β. Given N pairs of plaintexts the number of expected matches is $p \cdot N$. For an ideal cipher the expected number is about $2^{-n} \cdot N$. If $2^{-n} \ll p$ then the analyzed cipher can be distinguished from an ideal cipher successfully.

An S-box is called *active* if its input difference is non-zero, otherwise the S-box is called *non-active* (or passive). In order to keep the data complexity low, the adversary tries to maximize the probability of the differential characteristic. Therefore, he chooses the differential characteristic such that the probability of the differential is as high as possible.

We now give an example of a differential characteristic for a simple block cipher as shown in Figure 2.1. The block cipher processes strings of size 16 bits. The S-boxes are labeled with $S_{i,j}$ and the rectangles before each S-box layer represent the key mixing layers. In our example the bold lines represent the positions where a bit is one in the difference, where the remaining bits are zero. The example shows how a certain plaintext difference propagates through three rounds of the sample cipher.

After the introduction of differential cryptanalysis, many extensions were proposed such as truncated differentials [94, 95] (considering differences that are only partially determined), the higher-order differential attack [95] (differences of differences), the boomerang attack [144] (combining two differentials each covering half of the cipher in an adaptive chosen plaintext and ciphertext attack scenario), the amplified boomerang attack [86] (transformation of the boomerang attack in an adaptive chosen plaintext attack), the rectangle attack [16] (combining a large amount of differentials in an amplified boomerang attack scenario), and the impossible differential attack [13] (a combination of input and output differences that cannot occur simultaneously).

In our later attacks we often use a special form a differentials which are called *truncated differentials*. These truncated differentials consider differences that are only partially determined. For example, differences where only a part of the whole difference is regarded or differences where it is only of interest if certain words or bytes have a zero or a non-zero difference.

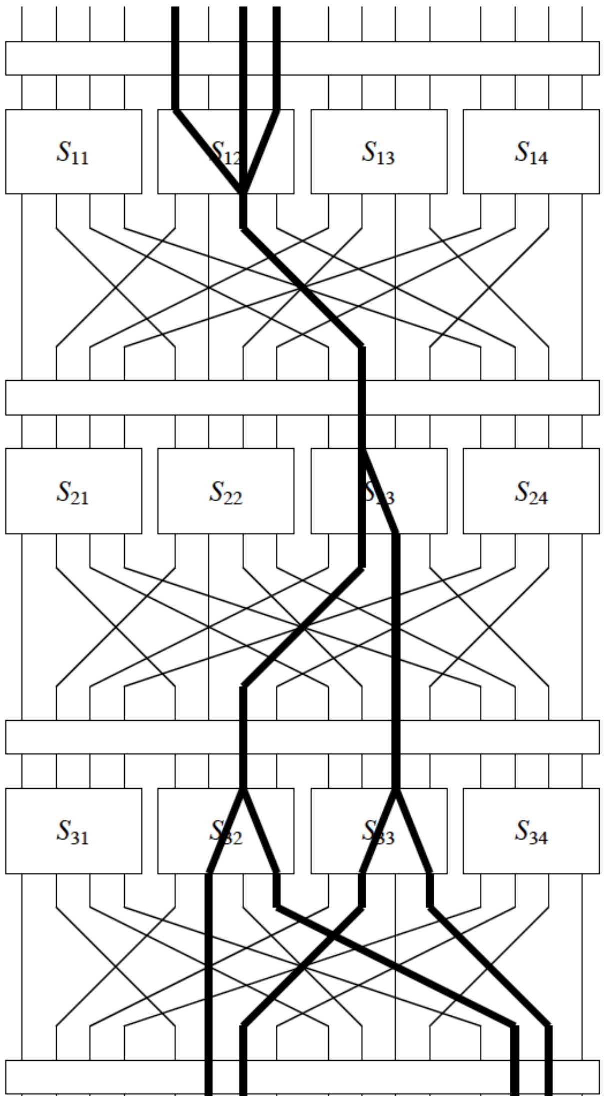


Figure 2.1: A differential characteristic over three rounds

2.4 Related-Key Differential Attack

Related-key attack was introduced independently by Biham [12] and Knudsen [93]. In this model the adversary can choose or observe the relations between different keys and exploit this additional knowledge to perform his attack. Block ciphers with weak key schedules, i.e., with a high degree of linear operations, are often more vulnerable to related-key attacks than block ciphers with strong key schedules.

Related-keys can be used to enhance differential attacks, i.e., exploiting properties of the keys the differential probability can be increased (or decreased in the case of related-key impossible differential attacks [18, 81]).

Let $\Delta K = K \oplus K'$ be the XOR difference between two keys K and K'. A related-key differential attack uses a related-key differential $\alpha \to \beta$ that holds with probability p under a key differential ΔK. Formally, the probability for such a differential is written as

$$\Pr_{P,K}\left[E_K(P) \oplus E_{K\oplus\Delta K}(P \oplus \alpha) = \beta\right] = p.$$

It is assumed that this probability is independent of P and K. As in the original differential attack the probability p of the related-key differential must be greater than 2^{-n} (or exactly zero in the case of impossible differentials) for the attack to succeed. Forming a differential characteristic with a high probability through the cipher becomes more likely if the differences between the subkeys lead to a few active S-boxes. This might increase the overall probability of the differential characteristic.

2.5 The Boomerang Attack

The *boomerang attack* was introduced by Wagner [144]. It is an extension of differential cryptanalysis which uses two differentials, each covering a part of the cipher instead of one differential for the whole cipher. Generally speaking, the less rounds a differential covers, the higher its probability. Using two short high probability differentials instead of a long low probability differential might be better if the total probability is higher in the first case.

The boomerang attack treats the cipher as a cascade of two *subcipher*s

$$E_K(P) = E1_K(E0_K(P)),$$

where K is the key used for the encryption and $E0(\cdot)$ and $E1(\cdot)$ are two subciphers, respectively. It is assumed that $E0(\cdot)$ and $E1(\cdot)$ are independent of each other. In the reminder of this subsection we discuss attacks on one key and thus we omit the key

K and write $E(P) = E1(E0(P))$ instead. Whenever we use only one key we use this notation.

We assume that the differential $\alpha \to \beta$ for $E0$ has probability p, and that the differential $\gamma \to \delta$ for $E1$ has probability q, where α, β, γ and δ are XOR differences. The backward direction (i.e., decryption) $E0^{-1}$ and $E1^{-1}$ of the differential for $E0$ and $E1$ are denoted by $\alpha \leftarrow \beta$ and $\gamma \leftarrow \delta$ and which have probabilities p and q, respectively. The boomerang attack includes a *data acquisition step*, a *data filtering step* and an *analysis step*.

Data Acquisition Step

To generate the amount of data needed for the attack the adversary performs the following steps:

1. The adversary chooses a pool of s (s depends on the probability of the differentials) plaintexts P_i, $i \in \{1, \ldots, s\}$ uniformly at random and computes a pool $P'_i = P_i \oplus \alpha$.
2. He asks for the encryption of P_i, i.e., $C_i = E(P_i)$, and he asks for the encryption of P'_i, i.e., $C' = E(P'_i)$.
3. He computes new ciphertexts $D_i = C_i \oplus \delta$ and $D'_i = C'_i \oplus \delta$.
4. He asks for the decryption of D_i, i.e., $O_i = E^{-1}(D_i)$, and he asks for the decryption of D'_i, i.e., $O'_i = E^{-1}(D'_i)$.

This step generates a certain amount of *quartets* (P_i, P'_i, O_i, O'_i) which are used in the following steps.

Data Filtering Step

The data filtering step reduces the number of quartets which are used to recover some key bits afterwards. It works as follows:

> For each pair (O_i, O'_i), $i \in \{1, \ldots, s\}$ check if $O_i \oplus O'_i$ is equal to α and store the quartet (P_i, P'_i, O_i, O'_i) into a set ϕ if so.

The number of quartets, which passes the filtering can be estimated in the following way. A pair (P, P') with difference α satisfies the differential $\alpha \to \beta$ with probability p. Let the output after the encryption with the subcipher $E0$ be A and A', respectively, i.e., $E0(P) = A$ and $E0(P') = A'$. These intermediate encryption values have difference

$\beta = A \oplus A'$ with probability p. The encryption of A and A' under the second subcipher $E1$ leads to the ciphertexts $C = E1(A)$ and $C' = E1(A')$, respectively. Using C and C', the new ciphertexts, $D = C \oplus \delta$ and $D' = C' \oplus \delta$ can be computed. The decryption under $E1$ leads to the intermediate encryption values $B = E1^{-1}(D)$ and $B' = E1^{-1}(D')$, respectively. A pair of ciphertexts with difference δ has a difference γ after passing $E1^{-1}$ with probability q, due to the differential $\delta \to \gamma$ which has probability q. Since we regard two such pairs of ciphertexts, the difference $\gamma = A \oplus B$ and $\gamma = A' \oplus B'$ has probability q^2. Since $A \oplus A' = \beta$ occurs with probability p, it follows that

$$\begin{aligned} (A \oplus B) \oplus (A \oplus A') \oplus (A' \oplus B') &= \\ \gamma \oplus \beta \oplus \gamma &= \beta = (B \oplus B') \end{aligned}$$

holds with probability $p \cdot q^2$ under the assumption that both differentials are independent of each other. A β difference of two intermediate encryption values leads to an α difference after passing the differential $\beta \to \alpha$ with probability p. Thus, a pair of plaintexts (P, P') with $P \oplus P' = \alpha$ generates a new pair of plaintexts (O, O') having a plaintext difference $\alpha = O \oplus O'$ with probability $p^2 \cdot q^2$. These two pairs form a quartet which is called a *right quartet* or a right boomerang quartet. A right quartet containing these two pairs is defined as:

Definition 1 *A quartet (P, P', O, O') which satisfies*

$$P \oplus P' = \alpha = O \oplus O',$$

$$A \oplus A' = \beta = B \oplus B',$$

$$A \oplus B = \gamma = A' \oplus B',$$

$$C \oplus D = \delta = C' \oplus D',$$

is called a ***right quartet****. A random quartet has probability $Pr_c = p^2 \cdot q^2$ to be a right one.*

In other words, a right quartet satisfies all the differential characteristics. Figure 2.2 displays the structure of such a right boomerang quartet. The adversary discards all quartets which do not satisfy the condition $P \oplus P' = \alpha = O \oplus O'$ and stores the remaining quartets in the set ϕ.

Analysis Step

From now on, the adversary operates on the remaining quartets stored in the set ϕ. The quartets can now be used in a distinguishing attack or to recover some (or all) key

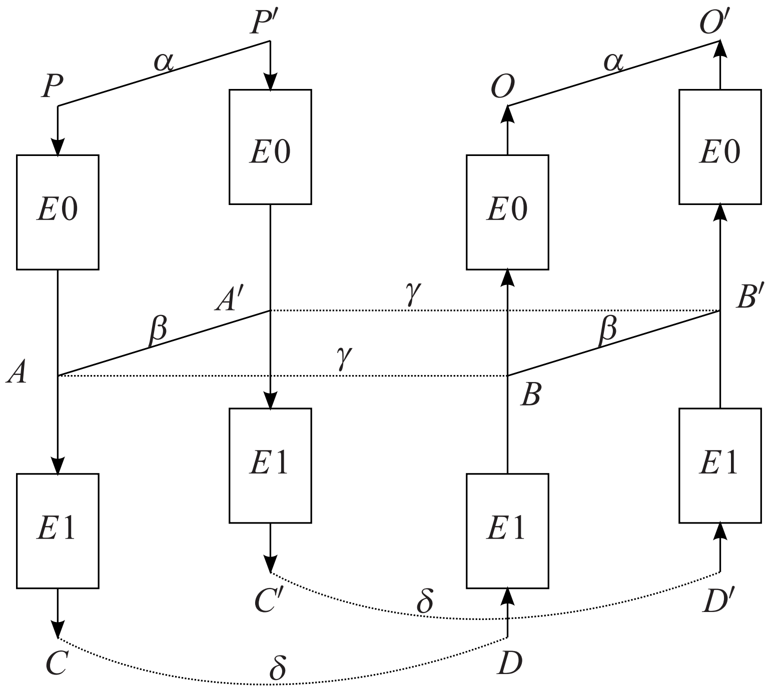


Figure 2.2: A right boomerang quartet

bits from either the first or the last rounds of the cipher (or both simultaneously). We describe the last round analysis. The other cases work similarly to this case. Let k be some key bits of the last subkeys derived from the keys K. Let $d_k(C)$ be the one round partial decryption of C under the key k. The key recovery step can be done with the following simple algorithm:

- For each candidate k

  1. Initialize a counter to zero.

  - For all quartets (P_i, P'_i, O_i, O'_i) stored in ϕ, where $i \in \{1, 2, \ldots, |\phi|\}$.

    2. For the plaintext quartet (P_i, P'_i, O_i, O'_i) and the respective ciphertext quartet (C_i, C'_i, D_i, D'_i), decrypt the ciphertext quartets (C_i, C'_i, D_i, D'_i), i.e., $\overline{C_i} = d_k(C_i), \overline{C'_i} = d_k(C'_i), \overline{D_i} = d_k(D_i)$ and $\overline{D'_i} = d_k(D'_i)$.
    3. Test whether the differences $\overline{C_i} \oplus \overline{D_i}$ and $\overline{C'_I} \oplus \overline{D'_i}$ have the desired difference an adversary expects depending on the differential characteristic $\delta \to \gamma$. Increase the counter if the difference is satisfied in both pairs.

4. Output the candidate k with the highest counter as the right one and perform an exhaustive key search on the remaining key bits.

Four cases can be differentiated in Step 3, since ϕ contains right quartets and quartets which do not satisfy all the differential characteristics. The key-bit combination k can either be right or wrong. A right quartet encrypted with the right key bits has the desired difference needed to pass the test in Step 3 with probability 1. Hence, the counter for the right key bits is increased. The three other cases are: a right quartet is used with false key bits (cK_f), a wrong quartet is used with the correct key-bits (fK_c) or a wrong quartet is used with a false key-bit combination (fK_f), where the second case also increases the counter for the correct key bits. Usually, this additional increment is negligibly small compared to the case where the right quartets are encountered with the right key bits, and thus, we usually disregard it. The probabilities in the two other cases of incrementing the counter for the wrong key bits is

$$Pr_{filter} = Pr_{cK_f} + Pr_{fK_f}.$$

The differential characteristics have to be chosen such that the counter for the correct key bits is significantly higher. If the differential characteristics have a high probability the analysis step outputs the correct candidate key in Step 4 with a high success rate. This depends on several factors, e.g., the number of subkeys which are guessed and the number of wrong keys suggested by the quartets. Assuming that the adversary wants to recover key bits at the last round of the cipher, then it could occur that no active S-boxes are available in the last round. Then he has to guess some subkeys from the last round and additionally from a previous round at the positions where active S-boxes are available.

We need at least one right quartet to mount the boomerang attack. Thus, the amount of data needed for the attack is $s = O(p^{-2}q^{-2})$ adaptive chosen plaintexts and ciphertexts such that we can expect one right quartet.

2.6 The Amplified Boomerang Attack

The *amplified boomerang attack* [86] is a descendant of the boomerang attack. This attack transforms the boomerang attack into a chosen plaintext attack. We treat the cipher E as a cascade of two subciphers $E(P) = E1(E0(P))$, as in the boomerang attack. The differentials being used are $\alpha \to \beta$ for $E0$ which has probability p and $\gamma \to \delta$ for $E1$ which has probability q.

In this attack the adversary uses quartets of plaintexts (P_1, P_2, P_3, P_4), such that $P_1 \oplus P_2 = \alpha$ and $P_3 \oplus P_4 = \alpha$. Each pair satisfies the differential $\alpha \to \beta$ with probability p in $E0$. Thus, we obtain

$$E0(P_1) \oplus E0(P_2) = \beta = E0(P_3) \oplus E0(P_4)$$

with probability p^2. If this condition holds, then $E0(P_1) \oplus E0(P_3)$ equals $E0(P_2) \oplus E0(P_4) = (E0(P_1) \oplus \beta) \oplus (E0(P_3) \oplus \beta)$. If $E0(P_1) \oplus E0(P_3)$ equals to γ, then $E0(P_2) \oplus E0(P_4)$ equals to γ as well. Under the assumption that the intermediate encryption values are distributed uniformly over all possible values, with probability 2^{-n}, the condition $E0(P_1) \oplus E0(P_3) = \gamma$ is satisfied. A pair of intermediate encryption values with difference γ satisfies the second differential $\gamma \to \delta$ for $E1$ with probability q. Finally, for two ciphertext pairs with $C_1 \oplus C_3 = \delta = C_2 \oplus C_4$ the probability is q^2 to obtain a δ difference after $E1$ in both pairs. A *right quartet* satisfies all these conditions.

Having N pairs with difference α, about pN of them are expected to satisfy the first differential for $E0$. From these pN pairs with difference β after $E0$ we can construct about $(pN)^2/2$ quartets that consist of two such pairs. As stated above with probability 2^{-n}, a difference γ occurs in the two pairs of the quartet after $E0$ under the assumption that the intermediate encryption values are distributed uniformly over all possible values. Thus, the remaining number of quartets is about $(pN)^2/2^{-n}$ which have a certain difference γ in two pairs after $E0$. Each pair with intermediate difference γ satisfies the second differential for $E1$ with probability q. The expected number of right quartets after $E1$ is about

$$\binom{pN}{2} \cdot 2^{-n} \cdot q^2 \approx N^2 \cdot 2^{-n-1} \cdot (pq)^2.$$

For an ideal cipher the expected number of quartets is about $N^2 \cdot 2^{-2n} (\approx (N^2/2) \cdot 2^{-2n+1})$, since there are N pairs that can be combined approximately to N^2 candidate quartets. For each pair (C_1, C_3), (C_2, C_4) the probability of having a specific difference in the output is 2^{-n}. Thus, the amplified boomerang attack can be mounted successfully if $2^{-n/2} < pq$ holds and if N is sufficiently large. In this way we can distinguish between the block cipher E and an ideal cipher and recover some key bits.

2.7 The Rectangle Attack

The *rectangle attack* [16] is an improvement of the amplified boomerang attack. As in the amplified boomerang attack for a *right quartet* (P_1, P_2, P_3, P_4) the following properties hold:

$$P_1 \oplus P_2 = \alpha = P_3 \oplus P_4 \quad \text{and} \quad C_1 \oplus C_3 = \delta = C_2 \oplus C_4,$$

where α and δ are the differences as before.

In the amplified boomerang attack we assumed that we have a differential $\alpha \to \beta$ for $E0$ which has probability p and another differential $\gamma \to \delta$ for $E1$ which has probability q. Instead of using a specific γ we count over all possible γ_i for which $\gamma_i \to \delta$ for $E1$ exists. A second improvement can be made by counting over all possible β_i for which

$\alpha \to \beta_i$ for $E0$ holds. This can be done since the differences β_i and γ_i do not matter, only α and δ are of interest for the attack.

The rectangle attack can be mounted for all possible values of β_i and γ_i in the following way. Firstly, using all possible values of β_i simultaneously. Let $p_i^2 = \Pr^2[\alpha \to \beta_i]$ be the probability that the two pairs of the quartet satisfies the differential $\alpha \to \beta_i$ for $E0$ for one fixed value of β_i, i.e., both pairs of the quartet satisfy the first differential. Hence, counting over the probabilities for all possible differential $\alpha \to \beta_i$ for all possible values of β_i we write

$$\hat{p} = \sqrt{\sum_{\beta_i} p_i^2} = \sqrt{\sum_{\beta_i} \Pr^2[\alpha \to \beta_i]}.$$

Thus, we obtain the probability of a right quartet as

$$2^{-n} \cdot \left(\sum_{\beta_i} \Pr^2[\alpha \to \beta_i] \right) \cdot q^2 = 2^{-n} \cdot (\hat{p}q)^2 .$$

Secondly, we use all possible differentials $\gamma_i \to \delta$ in the attack. Let $q_i^2 = \Pr^2[\gamma_i \to \delta]$ be the probability that a pair satisfy the differential $\gamma_i \to \delta$ for $E1$ for one fixed value of γ_i. Now, we can count over all the probabilities for all the differentials $\gamma_i \to \delta$ which is

$$\hat{q} = \sqrt{\sum_{\gamma_i} q_i^2} = \sqrt{\sum_{\gamma_i} \Pr^2[\gamma_i \to \delta]}.$$

The probability for a right quartet can now be written as

$$2^{-n} \cdot \left(\sum_{\beta_i} \Pr^2[\alpha \to \beta_i] \right) \cdot \left(\sum_{\gamma_i} \Pr^2[\gamma_i \to \delta] \right) = 2^{-n} \cdot (\hat{p}\hat{q})^2 .$$

Thirdly, in the boomerang attack we know the exact counterpart of the ciphertext C_1, which is $C_3 = C_1 \oplus \delta$, and C_4 is the counterpart of C_2. This is not the case in the rectangle attack, since for every pair of pairs (P_1, P_2) and (P_3, P_4) there are two possible quartets that can be built as $((P_1, P_2), (P_3, P_4))$ or $((P_1, P_2), (P_4, P_3))$. Now we can check whether a quartet $((P_1, P_2), (P_3, P_4))$ satisfies the rectangle conditions $C_1 \oplus C_2 = \delta = C_2 \oplus C_4$. If the test fails, i.e., the quartet $((P_1, P_2), (P_3, P_4))$ is not a right quartet, then the quartet $((P_1, P_2), (P_4, P_3))$, might still be a right quartet. Therefore, given N plaintext pairs with difference α the expected number of quartets that satisfy the differential is about

$$N^2 \cdot 2^{-n} \cdot (\hat{p}\hat{q})^2 . \tag{2.1}$$

Equation (2.1) gives a lower bound on the expected number of quartets. One can construct quartets that satisfy the rectangle conditions but are not counted by Equation (2.1). Assume that we have a quartet of the following form. For the intermediate differences $E0(P_1) \oplus E0(P_2) = a'$, $E0(P_2) \oplus E0(P_4) = b'$, and $E0(P_1) \oplus E0(P_3) =$

γ' and where $E0(P_2) \oplus E0(P_4) = a' \oplus b' \oplus \gamma'$ it can be possible that the differential $a' \oplus b' \oplus \gamma' \to \delta$ has a probability greater than zero. We can take this into consideration and count over the differentials for which this holds. Thus, the probability for a right quartet can be written as

$$2^{-n} \cdot \sum_{a_i, b_j} \left(\Pr[\alpha \to a_i] \cdot \Pr[\alpha \to b_j] \cdot \sum_{\gamma_k} \Pr[\gamma_k \to \delta] \cdot \Pr[a_i \oplus b_j \oplus \gamma_k \to \delta] \right). \quad (2.2)$$

Equation (2.2) counts over all quartets. However, it is infeasible to compute the exact probability for a quartet to occur since one has to count over all the differentials, which is hard. Thus, the expected lower bound given by Equation (2.1) is used instead.

The main difference between this attack and the amplified boomerang attack is that the probability for a right quartet is higher which results in a lower data and time complexities for the attack. A cipher, which has a key length much bigger than its block size is likely to be more vulnerable to the amplified boomerang or the rectangle attacks, because we always have to ”pay” the price of getting two equal internal states between both subciphers, i.e., they have a γ difference.

2.8 The Related-Key Boomerang Attack

The *related-key boomerang attack* [17] is the adaption of the boomerang attack to the related-key model. Instead of using the same key for the encryption and decryption process, in the related-key boomerang attack the adversary uses at least two different keys.

The cipher is treated as a cascade of two subciphers $E_K(P) = E1_K(E0_K(P))$, where K is the key used for encryption and decryption. We assume that the related-key differential $\alpha \to \beta$ for $E0$ has the probability p under the key difference ΔK^*, while the related-key differential $\gamma \to \delta$ for $E1$ has the probability q under the key difference $\Delta K'$. In the case of four key they are related as:

$$\begin{aligned} \Delta K^* &= K^a \oplus K^b = K^c \oplus K^d \text{ and} \\ \Delta K' &= K^a \oplus K^c = K^b \oplus K^d. \end{aligned}$$

We denote P^l as the plaintext P encrypted under the key K^l, $l \in \{a,b,c,d\}$. Let s be the amount of plaintexts needed for the attack. The attack works as follows:

Data Acquisition Step

- For $i = 1, 2, \ldots, s$ (s to be determined later) do

1. Choose a plaintext P_i^a at random and let $P_i^b = P_i^a \oplus \alpha$.
2. Ask for the encryption of P_i^a under K^a, i.e., $C_i^a = E_{K^a}(P_i^a)$, and of P_i^b under K^b, i.e., $C_i'^b = E_{K^b}(P_i^b)$.
3. Compute the new ciphertexts $D_i^c = C_i^a \oplus \delta$ and $D_i^d = C_i^b \oplus \delta$.
4. Ask for the decryption of D_i^c under K^c, i.e., $O_i^c = E_{K^c}^{-1}(D_i^c)$, and of D_i^d, under K^d, i.e., $O_i^d = E_{K^d}^{-1}(D_i^d)$.

Data Filtering Step

- For $i = 1, 2, \ldots, s$ do

  If $O_i^c \oplus O_i^d = \alpha$ store the quartet $(P_i^a, P_i^b, O_i^c, O_i^d)$ in the set ϕ, where ϕ is an empty set at the beginning.

Assume that a pair (P_i^a, P_i^b), $i \in \{1, \ldots, s\}$ with difference α satisfies the differential $\alpha \to \beta$ with probability p. Given P_i^a and P_i^b we denote the output of $E0$ by A_i^a and A_i^b, respectively, i.e., $A_i^a = E0_{K^a}(P_i^a)$ and $A_i^b = E0_{K^b}(P_i^b)$. A_i^a and A_i^b have difference $\beta = A_i^a \oplus A_i^b$ with probability p. The encryption of A_i^a and A_i^b with $E1$ leads to the ciphertexts C_i^a and C_i^b, respectively, i.e., $C_i^a = E1(A_i^a)$ and $C_i^b = E1(A_i^b)$. Using the ciphertexts C_i^a and C_i^b, the new ciphertexts $D_i^c = C_i^a \oplus \delta$ and $D_i^d = C_i^b \oplus \delta$ can be computed. Let $B_i^c = E1_{K^c}^{-1}(D_i^c)$ and $B_i^d = E1_{K^d}^{-1}(D_i^d)$ be the decryption of D_i^c and D_i^d under $E1$. Two ciphertexts with difference δ have a difference γ with probability q after the decryption with $E1$ and key difference $\Delta K'$. Since we have two ciphertext pairs with difference $\delta = C_i^a \oplus D_i^c = C_i^b \oplus D_i^d$, the intermediate differences $A_i^a \oplus B_i^c = \gamma$ and $A_i^b \oplus B_i^d = \gamma$ occur with probability q^2. As discussed earlier, the difference $\beta = A_i^a \oplus A_i^b$ occurs with the probability p. Thus, $(B_i^c \oplus B_i^d) = (B_i^c \oplus A_i^a) \oplus (A_i^a \oplus A_i^b) \oplus (A_i^b \oplus B_i^d) = \gamma \oplus \beta \oplus \gamma = \beta$ holds with probability pq^2. An input difference β leads to an output difference α through $E0^{-1}$ (under key difference ΔK^*) with probability p. Thus, a pair of plaintexts (P_i^a, P_i^b) with $P_i^a \oplus P_i^b = \alpha$ generates a new pair of plaintexts (O_i^c, O_i^d) where $O_i^c \oplus O_i^d = \alpha$ with probability $p^2 \cdot q^2$. The definition of right quartets is equivalent to that defined in the boomerang attack. Figure 2.3 displays the structure of a right related-key boomerang quartet.

After filtering step the set ϕ contains the remaining quartets which is used in the analysis step.

Analysis Step From now on, the adversary operates on the remaining quartets in ϕ. We now explain a key recovery variant which recovers some key bits from the first round of the cipher. Retrieving key bits from the last round of $E1$ works very similarly.

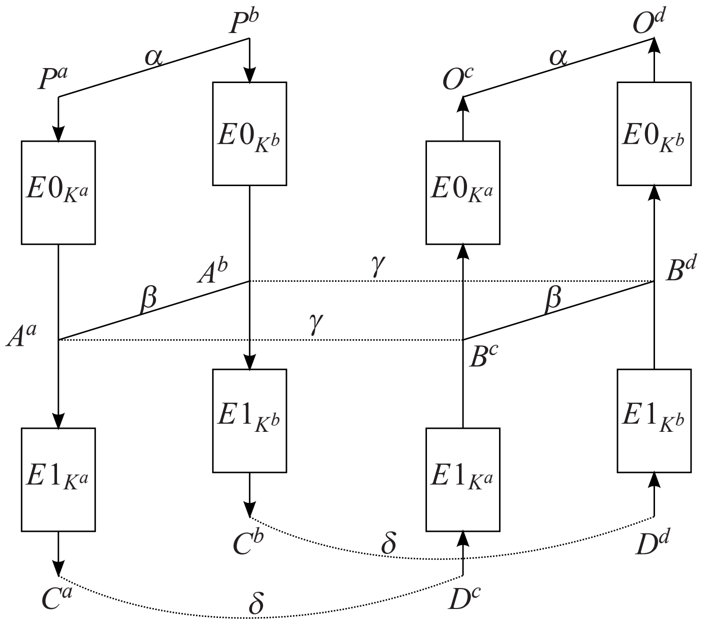


Figure 2.3: A right related-key boomerang quartet

Let $K_0^a, K_0^b, K_0^c, K_0^d$ be the 0-th subkeys derived from the keys K^a, K^b, K^c, K^d, i.e., the key which enters the cipher initially. The subkeys are related as

$$\begin{aligned}\Delta K_0^* &= K_0^a \oplus K_0^b = K_0^c \oplus K_0^d,\\ \Delta K_0' &= K_0^a \oplus K_0^c = K_0^b \oplus K_0^d,\end{aligned}$$

where ΔK_0^l, $l \in \{a,b,c,d\}$ is the key difference of the 0-th subkeys. The key recovery step works as follows:

- For each K_0^a and the respective keys K_0^b, K_0^c, K_0^d do
  1. Initialize a counter with zero.
  - For all quartets $(P_i^a, P_i'^b, O_i^c, O_i^d)$, $(i, j \in \{1,2,\ldots,s\})$ stored in ϕ do
    2. Ask for the encryption of the plaintext quartet $(P_i^a, P_i^b, O_i^c, O_i^d)$ one round under the guessed subkeys K_0^a and the keys K_0^b, K_0^c, K_0^d, respectively, i.e., $P_{1,i}^a = enc_{K_0^a}(P_i^a)$, $P_{1,i}^b = enc_{K_0^b}(P_i^b), O_{1,i}^c = enc_{K_0^c}(O_i^c)$ and $O_{1,i}^d = enc_{K_0^d}(O_i^d)$.
    3. Test whether the differences $P_{1,i}^a \oplus P_{1,i}^b$ and $P_{1,i}^c \oplus P_{1,i}^d$ have a desired difference an adversary expects depending on the related-key differential being used. Increase a counter for the used key-bits if the difference is fulfilled in both pairs.
4. Output the subkeys K_0^a, K_0^b, K_0^c and K_0^d with the highest counter as the correct one.

The analysis is the same as for the boomerang attack as well as the complexity for the attack. The data complexity of the attack is $s = O(p^2q^2)$ adaptive chosen plaintexts and ciphertexts (where the exact value of s depends on several parameters, e.g., the differentials in use, the cipher.

2.9 The Related-Key Rectangle Attack

The related-key rectangle attack [17, 78, 91] is an adaption of the rectangle attack to the related-key model. The analysis of the attack is very similar to the one described for the rectangle attack and the related-key boomerang attack, so we skip some of the details in this section.

The block cipher is treated as $E_{K^i}(P) = E1_{K^i}(E0_{K^i}(P))$, where P is a plaintext encrypted under the key K^i. It is assumed that there exist a related-key differential $\alpha \rightarrow \beta$ which holds with probability p for $E0$ under a key difference ΔK^*, i.e., $\Pr[E0_{K^a}(P^a) \oplus E0_{K^b}(P^b) = \beta | P^a \oplus P^b = \alpha] = p$, where K^a and $K^b = K^a \oplus \Delta K^*$ are two related keys, ΔK^* is a chosen key difference (the same holds for $\Pr[E0_{K^c}(O^c) \oplus E0_{K^d}(O^d) = \beta | O^c \oplus O^d = \alpha] = p$, where K^c and $K^d = K^c \oplus \Delta K^* = K^a \oplus \Delta K' \oplus \Delta K^*$ are two related keys). Note that these keys are related in the same way as we described in the realted-key boomerang attack. The related-key differential $\gamma \rightarrow \delta$ for $E1$ holds with probability q under the key difference $\Delta K'$. Given two intermediate encryption values A^a and B^c with difference $\gamma = A^a \oplus B^c$, we expect a ciphertext difference $\delta = C^a \oplus D^c$ with probability $\Pr[E1_{K^a}(A^a) \oplus E1_{K_c}(B^c) = \delta | A^a \oplus B^c = \gamma] = q$. The same argument can be applied for the other pair A^b and B^d, i.e., $\Pr[E1_{K^b}(A^b) \oplus E1_{K_d}(B^d) = \delta | A^b \oplus B^d = \gamma] = q$.

The expected number of quartets satisfying both conditions $E1_{K^a}(C^a) \oplus E1_{K^c}(D^c) = \delta$ and $E1_{K^b}(C^b) \oplus E1_{K^d}(D^d) = \delta$ is about

$$\sum_{\beta',\gamma'} N^2 \cdot 2^{-n} \cdot \hat{p}_\beta^2 \cdot \hat{q}_\gamma^2 \approx N^2 \cdot 2^{-n} \cdot (\hat{p} \cdot \hat{q})^2,$$

where $\hat{p} = \sqrt{\sum_{\beta'} (\Pr[\alpha \rightarrow \beta'])^2}$ and $\hat{q} = \sqrt{\sum_{\gamma'} (\Pr[\gamma' \rightarrow \delta])^2}$ if we count over all possible differentials. For an ideal cipher, the expected number of quartets is about $N^2 \cdot 2^{-2n}$. Therefore, if $p \cdot q > 2^{-n/2}$ and N is sufficiently large, the related-key rectangle distinguisher can distinguish between E and an ideal cipher and recover some key bits. Figure 2.4 displays the structure of a related-key rectangle quartet.

So far we have used differences where all values are specified. In our attacks we often use truncated differentials. This means that not all the bytes in a difference have to have a specific value. We only regard some bytes to be zero or non-zero. We use this concept where the specific differences are either hard to predict or are not of interest

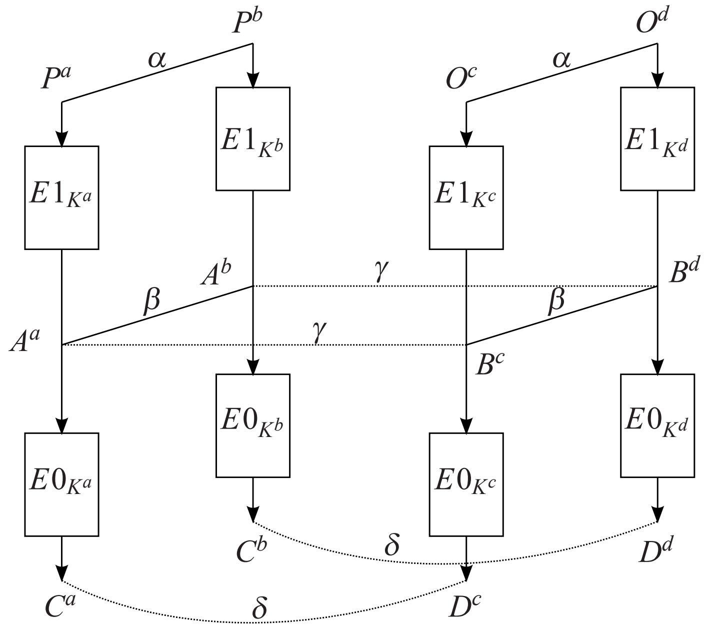


Figure 2.4: A right related-key rectangle quartet

for the attack. In this case a differential might have different probabilities in forward and backward direction.

2.10 A Hash Function

A cryptographic hash function $H : \{0,1\}^* \to \{0,1\}^n$ is used to compute an n-bit digest from an arbitrarily-sized input. Cryptographic hash functions should satisfy the following security requirements:

- *Collision resistance* – It is hard to find $x, x' \in \{0,1\}^*$ such that $x' \neq x$ and $H(x') = H(x)$,
- *2nd preimage resistance* – Given a value $x \in \{0,1\}^*$, it is hard to find $x' \in \{0,1\}^*$ such that $x' \neq x$ and $H(x') = H(x)$.
- *Preimage resistance* – Given a hash value $y \in \{0,1\}^n$, it is hard to find $x \in \{0,1\}^*$ such that $H(x) = y$,

Ideally, cryptographers expect a good hash function to somehow behave like a *random oracle*. A random oracle [6] is a black box that responds to every query with a randomly chosen answer and saves the query together with its responds. Given a query, the oracle checks whether the query was asked before. If so, the oracle gives the same answer as before, otherwise, the oracle selects an answer at random, and stores the query with the answer for future uses.

Current practical hash functions, such as SHA-1 or SHA-2 [119, 120], are *iterated* hash functions, using a *compression function* with a fixed-length input, say

$$h : \{0,1\}^{n+l} \to \{0,1\}^n.$$

SHA-1 or SHA-2 use the MERKLE-DAMGÅRD transformation [41, 113] to obtain a hash function H with arbitrary input sizes. The core idea is to split the message M into l-bit blocks $M_1, \dots, M_m \in \{0,1\}^l$ (with some padding, to ensure that all the blocks are of size l-bit), to define an *initial value* X_0, and to apply the recurrence $X_i = h(X_{i-1}, M_i)$. The final *chaining variable* X_m is used as the hash output or *hash value*. The main benefit of the MERKLE-DAMGÅRD transformation is that it preserves collision resistance: if the compression function is collision resistant, then so is the hash function [41, 113][2].

A hash function H with *length extension* property is hash function for which one can compute $h(x||y)$ for a chosen message y from $h(x)$. This is a weakness of MERKLE-DAMGÅRD hash functions, since the adversary has access to the entire internal state after the final message block is processed. The length extension attack can be applied as follows. For a given hash value one can append additional message blocks to the message and compute the new hash value.

A *Message Authentication Code (MAC)* computes a fingerprint of a given message using a secret key. The MAC protects both the message's integrity as well as its authenticity, by allowing verifiers, who also possess the secret key, to detect any changes to the message content. Consider for example a secret key K, a message M and define a MAC, $\text{MAC}(K,M) = H(K||M)$. For a MERKLE-DAMGÅRD-based hash function H, one can easily forge tags for messages related to M in the following way. Given $\text{MAC}(K,M) = H(K||M)$, where K is an unknown key and an arbitrary value Y, one can compute a valid MAC for the message $M||Y$, even without knowing K as $\text{MAC}(K,M||Y) = H(K||M||Y)$ using the length extension property of MERKLE-DAMGÅRD. A MAC can be build using a hash function, a block cipher, a stream cipher, or a dedicated algorithm to generate an output of fixed length.

[2]Note, that this is only true, if the padding rule offers a prefix-key encoding [36].

Chapter 3

Cryptanalysis of Block Ciphers

Since the AES is the world wide standard for symmetric key encryption it received much attention in cryptanalysis during the last years. Thus, analyzing the security of the AES is important. If there are any weaknesses an adversary can use, people would lose trust in the strength of the AES.

In 2003 a block cipher called ARIA was proposed to serve as a replacement for the AES. ARIA is very similar to the AES and shares some of its operations as well, but the authors claim that ARIA is more secure than the AES while its performance is only slightly lower than of the AES'.

In this chapter we present new cryptanalytic attacks on the block ciphers AES and ARIA.

3.1 Boomerang Attack on ARIA

The ARIA block cipher [99] was presented at ICISC'03. ARIA is a substitution-permutation network whose structure is based on the Advanced Encryption Standard (AES) [40]. ARIA [99] uses data blocks of 128 bits with a 128, 192 or 256-bit key. A different number of rounds are used depending on the length of the key, 10, 12, and 14 rounds when a 128, 192 or 256-bit key is used, respectively. Moreover, it is claimed to have better security against all existing attacks on block ciphers.

Previous cryptanalytic attacks at ARIA of Wu et al. [153] showed that ARIA is vulnerable to impossible differential attacks up to 6 rounds. Li et al. [134] also proposed some impossible differential attacks of up to 6 rounds of ARIA. Table 3.1 summarizes the existing results on ARIA. In the following we present a new attacks on 5-round ARIA.

Table 3.1: Comparison of attacks on ARIA

Attack	**# Rounds**	**Data**	**Memory**	**Time**	**Source**
Impossible Differential	5	$2^{71.3}$ CP	2^{72} mem*	$2^{71.6}$	[134]
Boomerang Attack	5	2^{109} ACPC	2^{57} mem	2^{110}	Sec. 3.1.2
Impossible Differential	6	2^{121} CP	2^{121} mem*	2^{112}	[153]
Impossible Differential	6	$2^{120.5}$ CP	2^{121} mem*	$2^{104.5}$	[134]
Impossible Differential	6	2^{113} CP	2^{113} mem*	$2^{121.6}$	[134]

CP: Chosen Plaintexts, ACPC: Adaptive Chosen Plaintexts and Ciphertexts, mem: memory usage in blocks.

* We estimated the memory usage of these attacks since it was not mentioned in the paper.

3.1.1 Description of ARIA

ARIA has an SPN structure, which contains two kinds of S-boxes and two types of substitution layers which are different between even and odd rounds. The diffusion layer of ARIA uses a 16×16 binary matrix with a branch number 8.[1] ARIA is claimed to be more efficient in 8-bit and 32-bit software implementations than the AES. The plaintexts are treated as a 4 x 4 byte matrix, which is called the state. A round applies three operations to the state:

- **Substitution layer (SL)** is a non-linear byte-wise substitution applied on every byte of the state matrix in parallel, where two different substitution layers exist, denoted by SL_1 and SL_2 (SL_1^{-1} and SL_2^{-1} for their inverse).
- **Diffusion layer (DL)** is a linear matrix multiplication of the state matrix with a 16×16 involution binary matrix.
- **Round key addition (ARK)** is the XORing of the state and a 128-bit subkey which is derived from the key.

Before the first round, an initial ARK operation is applied and the DL operation is omitted in the last round. The bytes coordinates of a 4 x 4 state matrix are labeled as in Figure 3.1.

Substitution Layer (SL). ARIA uses two S-boxes S_1 and S_2 and also their inverse S_1^{-1}, S_2^{-1}, where S_1 is the same S-box used for the AES. Each S-box is defined to be an

[1] A differential *branch number* of a linear transformation M is given by

$$\min_{a \neq 0}\{w_b(a) + w_b(M(a))\}$$

where $w_b(a)$ is the number of non-zero elements of the vector a. We always mean differential branch number whenever we write branch number.

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

Figure 3.1: Byte coordinates of a 4 x 4 state matrix of ARIA

affine transformation of the inversion function over GF(2^8).

$$S_1, S_2 : GF(2^8) \to GF(2^8),$$

$$S_1 : x \mapsto A \cdot x^{-1} \oplus a,$$

where

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \quad \text{and} \quad a = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}.$$

$$S_2 : x \mapsto B \cdot x^{247} \oplus b,$$

where

$$B = \begin{pmatrix} 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \end{pmatrix} \quad \text{and} \quad b = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \end{pmatrix}.$$

ARIA has two types of S-box layers for even and odd rounds as shown in Figures 3.2 and 3.3. Type 1 is used in the odd rounds and type 2 is used in the even rounds.

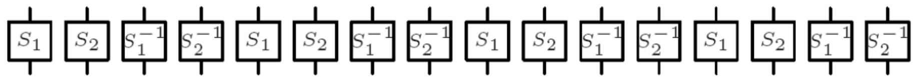


Figure 3.2: The S-box layer SL_1

Diffusion Layer (DL). A mapping GF$(2^8)^{16} \to$ GF$(2^8)^{16}$ is performed which is given by

$$(x_0, x_1, \ldots, x_{15}) \mapsto (y_0, y_1, \ldots, y_{15}),$$

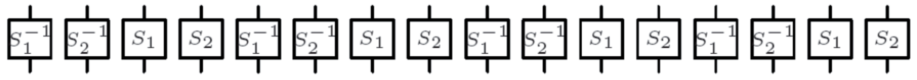


Figure 3.3: The S-box layer SL_2

where

$$\begin{aligned}
y_0 &= x_3 \oplus x_4 \oplus x_6 \oplus x_8 \oplus x_9 \oplus x_{13} \oplus x_{14},\\
y_1 &= x_2 \oplus x_5 \oplus x_7 \oplus x_8 \oplus x_9 \oplus x_{12} \oplus x_{15},\\
y_2 &= x_1 \oplus x_4 \oplus x_6 \oplus x_{10} \oplus x_{11} \oplus x_{12} \oplus x_{15},\\
y_3 &= x_0 \oplus x_5 \oplus x_7 \oplus x_{10} \oplus x_{11} \oplus x_{13} \oplus x_{14},\\
y_4 &= x_0 \oplus x_2 \oplus x_5 \oplus x_8 \oplus x_{11} \oplus x_{14} \oplus x_{15},\\
y_5 &= x_1 \oplus x_3 \oplus x_4 \oplus x_9 \oplus x_{10} \oplus x_{14} \oplus x_{15},\\
y_6 &= x_0 \oplus x_2 \oplus x_7 \oplus x_9 \oplus x_{10} \oplus x_{12} \oplus x_{13},\\
y_7 &= x_1 \oplus x_3 \oplus x_6 \oplus x_8 \oplus x_{11} \oplus x_{12} \oplus x_{13},\\
y_8 &= x_0 \oplus x_1 \oplus x_4 \oplus x_7 \oplus x_{10} \oplus x_{13} \oplus x_{15},\\
y_9 &= x_0 \oplus x_1 \oplus x_5 \oplus x_6 \oplus x_{11} \oplus x_{12} \oplus x_{14},\\
y_{10} &= x_2 \oplus x_3 \oplus x_5 \oplus x_6 \oplus x_8 \oplus x_{13} \oplus x_{15},\\
y_{11} &= x_2 \oplus x_3 \oplus x_4 \oplus x_7 \oplus x_9 \oplus x_{12} \oplus x_{14},\\
y_{12} &= x_1 \oplus x_2 \oplus x_6 \oplus x_7 \oplus x_9 \oplus x_{11} \oplus x_{12},\\
y_{13} &= x_0 \oplus x_3 \oplus x_6 \oplus x_7 \oplus x_8 \oplus x_{10} \oplus x_{13},\\
y_{14} &= x_0 \oplus x_3 \oplus x_4 \oplus x_5 \oplus x_9 \oplus x_{11} \oplus x_{14},\\
y_{15} &= x_1 \oplus x_2 \oplus x_4 \oplus x_5 \oplus x_8 \oplus x_{10} \oplus x_{15}.
\end{aligned}$$

Round Key Addition (ARK). The subkeys are derived from the key using the key schedule algorithm which uses a 3-round 256-bit Feistel cipher. We skip its description since we do not use it in our attack. We refer the reader to [99] for more details.

3.1.2 A Boomerang Attack on 5-Round ARIA

In this section we mount a boomerang attack on 5-round ARIA-128. The reduced cipher is treated as $E(P) = E1(E0(P))$, where a differential for $E0$ containing rounds 1 to 3 and a differential for $E1$ is covering rounds 4 to 5. We apply a key recovery attack to retrieve 56 key-bits of the first round. The notations used in our attack are defined as:

- P_i, O_i are plaintexts.

- C_i, D_i are ciphertexts.
- a is a known non-zero byte difference.
- $*$ is a non-zero byte differences.
- ? is an unknown byte differences.

The Differential for $E0$ ($\alpha \to \beta_{out}$)

Considering the S-box being used, for any non-zero difference there are 126 values which occur with probability 2^{-7}, one with probability 2^{-6} and 129 with probability 0. We pick a difference α with b differences in 7 byte positions, i.e., bytes 3, 4, 6, 8, 9, 13, and 14. We choose the b difference such that it transforms into an a difference with probability 2^{-6}. Thus, the non-zero differences in the difference α of the differential for $E0$ transforms into an a difference in bytes 3, 4, 6, 8, 9, 13 and 14 with probability 2^{-42}. DL_1 then leaves an a difference in byte 0, while the remaining bytes have a zero difference. Since ARK is linear it does not alter this difference. SL_2 produces a non-zero difference in byte 0 and DL_2 spreads this difference in bytes 3, 4, 6, 8, 9, 13 and 14. At the end of the differential we obtain a difference called β_{out} where all the 16 bytes of the state difference are unknown. We discuss this below in more details. The probability of the differential for $E0$, i.e., the transformation of an α difference into a β_{out} difference, is given by

$$Pr(\alpha \to \beta_{out}) \quad = \quad 2^{-42}.$$

The differential $E0$ is shown in Figure 3.4.

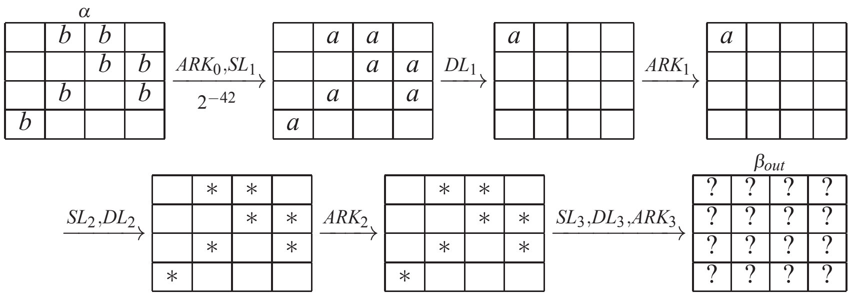


Figure 3.4: The differential for $E0$

The Differential for $E1^{-1}$ ($\delta \to \gamma$)

The ciphertext difference δ consists of a single a difference in byte 0 and a zero difference in the remaining bytes. The difference remains non-zero in only one byte (namly, byte 0) after the inverse of round 5. Recall, that there is no DL_5 operation in the last round of ARIA. The DL_4^{-1} operation spreads this non-zero difference to bytes 3, 4, 6, 8, 9, 13 and 14. We denote the the output difference of the differential by γ. The probability of $E1^{-1}$ is $\Pr(\delta \to \gamma) = 1$. The differential $E1^{-1}$ is shown in Figure 3.5.

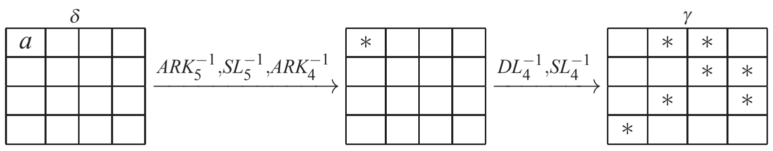


Figure 3.5: The differential for $E1^{-1}$

The Differential for $E0^{-1}$ ($\beta_{in} \to \alpha$)

For the following steps we need that the output difference β_{out} of the differential for $E0$ is equal to the difference β_{in} for the differential for $E0^{-1}$. Note that β_{in} and β_{out} are not only equal in the positions of non-zero differences but should also be equal in each byte. We compute the probability that this actually happens. From the boomerang condition inside the cipher for two differences γ_1 and γ_2 we know that

$$\beta_{out} \oplus \gamma_1 \oplus \gamma_2 = \beta_{in}$$

holds. When γ_1 and γ_2 are equal in all the bytes, we simply write γ. We compute the probability for that to occur below. Thus, the above condition reduces to:

$$\beta_{out} \oplus \gamma \oplus \gamma = \beta_{out} = \beta_{in} \tag{3.1}$$

Using the differentials above, the differences β_{in} and β_{out} are equal with probability 2^{-56}. This is the probability that the 7 non-zero bytes in γ_1 are equal to the 7 non-zero bytes in γ_2.

Let A, A', B, B' be the internal state after SL_3 in the forward direction when encrypting P, P', O, O', respectively. The notation from Figure 2.2 is used. Since DL is linear γ can be expressed as

$$\gamma = K_3 \oplus DL_3(A) \oplus K_3 \oplus DL_3(B) = DL_3(A \oplus B) \tag{3.2}$$

and as

$$\gamma = K_3 \oplus DL_3(A') \oplus K_3 \oplus DL_3(B') = DL_3(A' \oplus B'). \tag{3.3}$$

Equations (3.2) and (3.3) can be combined, which leaves $A \oplus A' = B \oplus B'$. In other words, DL_3 can be undone with probability 1 due to the boomerang condition (3.1).

This means that we know exactly that after DL_3 in the backward direction bytes 3, 4, 6, 8, 9, 13, and 14 are non-zero while the remaining bytes are with difference zero. There are several cases for which an a difference in bytes 3, 4, 6, 8, 9, 13, and 14 occurs after SL_3, which are one case with probability 2^{-42}, $7 \cdot 127$ with probability 2^{-43} and so forth until 126^7 cases with probability 2^{-49} each. Thus on average, after SL_3 an a difference in bytes 3, 4, 6, 8, 9, 13, and 14 occurs with probability $(2^{-6.93})^7 = 2^{-48.79}$. DL_2 outputs an a difference in byte 0 and a zero difference in the remaining bytes. SL_2 then transforms the a difference in byte 0 into a non-zero difference, which is spread into the bytes 3, 4, 6, 8, 9, 13, and 14 after DL_1. The output difference α of the differential for $E0^{-1}$ contains these non-zero and zero differences. The differential for $E0^{-1}$ has the probability $\Pr(\beta_{in} \to \alpha) \approx 2^{-49}$ to occur. It is shown in Figure 3.6.

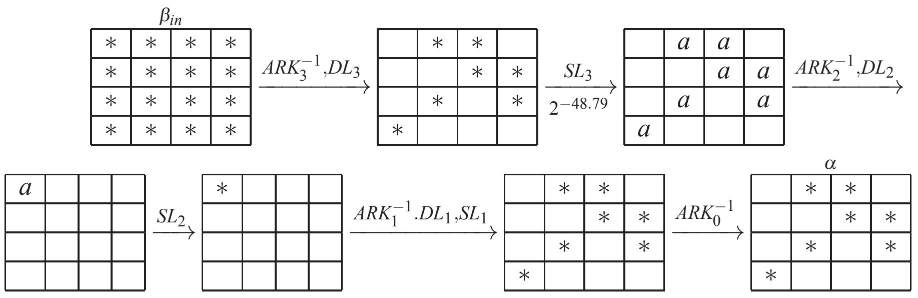


Figure 3.6: The differential for $E0^{-1}$

The Attack

The adversary first collects data and stores the filtered data in the set ϕ. A key-search is then applied to the remaining quartets in ϕ in order to find 56 bits of K_0. Let k_0 be a 56-bit subkey in the position of bytes 3, 4, 6, 8, 9, 13, and 14 of K_0. Let $e_{0,k}(X)$ be the partial encryption of X under the subkey k before DL_1 is applied. The attack is as follows:

1. Choose 2^{53} structures S_j, $j \in \{1, 2, \ldots, 2^{53}\}$ each consists of 2^{56} plaintexts $P_{i,j}$, $i \in \{1, 2, \ldots, 2^{56}\}$ which have all the possible values in seven bytes (3, 4, 6, 8, 9, 13, and 14). Ask for the encryption of all $P_{i,j}$ to obtain the ciphertexts $C_{i,j}$, i.e., $C_{i,j} = E(P_{i,j})$. Let $P'_{i,j} = P_{i,j} \oplus \alpha$ and let $C'_{i,j} = E(P'_{i,j})$.
2. For each ciphertext $C_{i,j}$ compute a new ciphertext $D_{i,j} = C_{i,j} \oplus \delta$, where δ is a fixed 128-bit value with a non-zero value a in byte 0 and zero in the remaining bytes. For each ciphertext $C'_{i,j}$ compute a new ciphertext $D'_{i,j} = C'_{i,j} \oplus \delta$, respectively.

3. Ask for the decryption of the $D_{i,j}$ and $D'_{i,j}$ to obtain the new ciphertexts $O_{i,j}$ and $O'_{i,j}$, i.e., $O_{i,j} = E^{-1}(D_{i,j})$ and $O'_{i,j} = E^{-1}(D'_{i,j})$, respectively.
4. Store only those quartets $(P_{i,j}, P'_{l,j}, O_{i,j}, O'_{l,j})$ in the set ϕ where $O_{i,j} \oplus O'_{l,j}$ have a non-zero difference in bytes 3, 4, 6, 8, 9, 13, and 14 and a zero difference in the remaining bytes.
5. For each 56-bit candidate key k
   - Set a counter to zero.

     For each quartet passing the test in Step 4:

     5.1. Partially encrypt the plaintext quartet $(P_{i,j}, P'_{l,j}, O_{i,j}, O'_{l,j})$, i.e., compute $\overline{P_{i,j}} = e_{0,k}(P_{i,j})$, $\overline{P'_{l,j}} = e_{0,k}(P'_{l,j})$, $\overline{O_{i,j}} = e_{0,k}(O_{i,j})$ and $\overline{O'_{l,j}} = e_{0,k}(O'_{l,j})$.

     5.2. Increase the counter for the used 56-bit subkey k by one if $\overline{P_{i,j}} \oplus \overline{P'_{l,j}}$ and $\overline{O_{i,j}} \oplus \overline{O'_{l,j}}$ have a difference of a in bytes 3, 4, 6, 8, 9, 13, and 14.
6. Output the 56-bit subkey k with the highest counter.

Analysis of the Attack

We have 2^{53} structures which contain 2^{55} plaintext pairs of the desired difference each. Thus, we expect about $\#PP = 2^{53} \cdot 2^{55} = 2^{108}$ quartets in total. Thus, the transition in this differential happens with probability one in this case. A right quartet occurs with probability

$$\begin{aligned} Pr_c &= \Pr(\alpha \to \beta_{out}) \cdot (\Pr(\gamma \leftarrow \delta))^2 \cdot \Pr(\gamma_1 = \gamma_2) \cdot \Pr(\alpha \leftarrow \beta_{in}) \\ &= 1 \cdot 1 \cdot 2^{-56} \cdot 2^{-49} = 2^{-105}, \end{aligned}$$

since all the differential conditions are fulfilled. A random difference of two plaintexts has 9 zero byte difference with probability $Pr_f = 2^{-72}$. Thus, after Step 4 we have about $\#C = 2^{108} \cdot 2^{-105} = 2^3$ right and approximately $\#F = 2^{108} \cdot 2^{-72} = 2^{36}$ wrong quartets. A quartet passes the test in Step 5.2 with probability $Pr_{filter} = 2^{-112}$, since we have a 56-bit filtering condition on both pairs of a quartet. Thus, we expect $\#CK_c = 2^3$ right boomerang quartets for the right key and $\#FK_c = \#F \cdot Pr_{filter} = 2^{36} \cdot 2^{-112} = 2^{-76}$ wrong quartets for each wrong subkey guess remain after this step.

Using the Poisson distribution[2] we can compute the success rate of our attack. The probability that the counter of a wrong key is at least 3 assuming $Y_i \sim Poisson(\mu =$

[2]Normally, we would use a binomial distribution but we use the Poisson distribution instead as a simplification. $X \sim Poisson(\mu)$ means that the random variable X follows the Poisson distribution with mean μ.

2^{-76}) is

$$\Pr(Y \geq 3) = e^{-2^{-76}} \cdot \frac{(2^{-76})^3}{3!} \approx 2^{-230}.$$

For all the $2^{56}-1$ wrong keys used in our analysis we expect about $2^{56} \cdot 2^{-230} = 2^{-174}$ wrong keys which have a count of at least 3 quartets. The probability that the right key has a count of at least 3 quartets using $Z \sim Poisson(\mu = 2^3)$ is

$$\Pr(Z \geq 3) \approx 0.98.$$

We can increase the success probability by increasing the number of quartets which also increases the data and time complexity of the attack.

Each structure of data can be analyzed sequentially. Thus, the total memory complexity is determined by Step 1 to 3, which is about $2 \cdot 2^{56} = 2^{57}$ blocks and additionally 2^{56} counters. The memory complexity of Steps 4, 5.1 and 5.2 is negligible compared to the memory complexity of the first two steps. The time complexity of Step 1 to 3 is $2 \cdot 2^{56} = 2^{57}$ encryptions for each structure. Since we have to run these steps for each structure of data the time complexity of the attack is about $2^{53} \cdot 2^{57} = 2^{110}$ 5-round encryptions. The data complexity is of size about $2^{53} \cdot 2^{56} = 2^{109}$ adaptive chosen plaintexts.

3.2 Related-Key Boomerang Attacks on 7 and 9-Round AES-192

In this section we present a related-key boomerang attack on 7 rounds of AES-192 using 4 related keys. Our related-key boomerang attack can also break 9 rounds using 256 related keys. Table 3.2 summarizes existing attacks on AES-192 and our new attacks on 7 and 9 rounds.

3.2.1 Description of AES-192

The AES [40] is a block cipher with 128-bit block size and 128, 192 or 256-bit key size. A different number of rounds is used depending on the length of the key, 10, 12 and 14 rounds when a 128, 192 or 256-bit key is used, respectively. The plaintexts are treated as a 4 x 4 byte matrix, which is called state. A round applies four operations to the state:

- SubBytes (SB) is a non-linear byte-wise substitution applied on every byte of the state matrix in parallel.

Table 3.2: Existing attacks on AES-192

Attack	**# Rounds**	**# Keys**	**Data**	**Time**	**Source**
Impossible Differential	7	1	2^{92}CP	2^{186}	[124]
Impossible Differential	7	1	2^{115}CP	2^{119}	[157]
Impossible Differential	7	1	2^{92}CP	2^{162}	[157]
Impossible Differential	7	1	$2^{113.8}$CP	$2^{118.8}MA$	[100]
Impossible Differential	7	1	$2^{91.2}$CP	$2^{139.2}$	[100]
Meet in the Middle	7	1	2^{34+n}CP	$2^{208-n}+2^{82+n}$	[43]
Square	7	1	2^{32}CP	2^{184}	[104]
Partial Sums	7	1	$19\cdot 2^{32}$CP	2^{155}	[49]
Partial Sums	7	1	$2^{128}-2^{119}$CP	2^{120}	[49]
Related-Key Differential	7	2	2^{37}RK-CP	2^{145}	[158]
Related-Key Differential-Linear	7	2	2^{22}RK-CP	2^{187}	[159]
Related-Key Differential-Linear	7	2	2^{70}RK-CP	2^{130}	[159]
Related-Key Impossible Differential	7	2	2^{111}RK-CP	2^{116}	[81]
Related-Key Impossible Differential	7	32	2^{56}RK-CP	2^{94}	[18]
Related-Key Impossible Differential	7	2	2^{52}RK-CP	2^{80}	[158]
Related-Key Boomerang Differential	7	4	2^{63}RK-ACPC	2^{63}	Sec. 3.2.2
Single-Key	7	1	2^{103+n}CP	2^{103+n}	[48]
Partial Sums	8	1	$2^{128}-2^{119}$CP	2^{188}	[49]
Related-Key Impossible Differential	8	2	2^{88}RK-CP	2^{183}	[81]
Related-Key Rectangle	8	4	$2^{86.5}$RK-CP	$2^{86.5}$	[78]
Related-Key Rectangle	8	2	2^{94}RK-CP	2^{120}	[90]
Related-Key Differential-Linear	8	2	2^{118}RK-CP	2^{165}	[159]
Related-Key Impossible Differential‡	8	32	2^{116}RK-CP	2^{134}	[18]
Related-Key Impossible Differential‡	8	32	2^{92}RK-CP	2^{159}	[18]
Related-Key Impossible Differential‡	8	32	$2^{68.5}$RK-CP	2^{184}	[18]
Related-Key Impossible Differential	8	2	$2^{64.5}$RK-CP	2^{177}	[158]
Related-Key Impossible Differential	8	2	2^{88}RK-CP	2^{153}	[158]
Related-Key Impossible Differential	8	2	2^{112}RK-CP	2^{136}	[158]
Meet-in-the-Middle	8	1	2^{113+n}CP	2^{172+n}	[48]
Related-Key Rectangle†	9	256	2^{86}RK-CP	2^{181}	[17]
Related-Key Rectangle	9	64	2^{85}RK-CP	2^{182}	[90]
Related-Key Boomerang	9	256	2^{112}RK-ACPC	$2^{134.8}$	Sec. 3.2.3
Related-Key Rectangle	10	256	$2^{121.2}$RK-CP	$2^{184.2}$	[90]
Related-Key Rectangle	10	64	$2^{119.2}$RK-CP	$2^{185.2}$	[90]
Related-Subkey Rectangle	12	4	2^{123}RS-ACPC	2^{176}	[25]

CP: Chosen Plaintexts, ACPC: Adaptive Chosen Plaintexts and Ciphertexts,
RK: Related-Key, RS: Related-Subkey#, MA: memory access,
Time: Encryption units.

† the attack with some flaws corrected by Kim et al. [90]. The table shows the corrected complexities.

‡ the attack with some flaws corrected by Zhang et al. [158]. The table shows the corrected complexities.

In a related-subkey attack the adversary chooses the difference between certain subkeys being used.

- ShiftRows (SR) is a cyclic left shift of the *i*-th row by *i* bytes, where $i \in \{0,1,2,3\}$.
- MixColumns (MC) is a multiplication of each column by a constant 4 x 4 matrix over $GF(2^8)$.
- AddRoundKey (ARK) is an XORing of the state and a 128-bit subkey which is derived from the key.

An AES round applies the SB, SR, MC and ARK operations in that order. Before the first round a whitening ARK operation is applied and the MC operation is omitted in the last round. We label the first round with 1. We concentrate on the 192-bit and 256-bit key versions of the AES in this thesis and refer to [40] for more details on the AES-128.

The state matrix is labeled as follows:

x_0	x_4	x_8	x_{12}
x_1	x_5	x_9	x_{13}
x_2	x_6	x_{10}	x_{14}
x_3	x_7	x_{11}	x_{15}

Let W_j be a 32-bit word and $W_j[i]$ be the *i*-th byte in W_j, then the 192-bit key is represented by $W_0||W_1||W_2||\dots||W_5$. The 192-bit key schedule algorithm works as follows:

- For $j = 6$ to 51
  - If $j \equiv 0 \mod 6$, then
    - $W_j[0] = W_{j-6}[0] \oplus SB(W_{j-1}[1]) \oplus Rcon(j/6)$,
    - For $i = 1$ to 3 do $W_j[i] = W_{j-6}[i] \oplus SB(W_{j-1}[i+1 \text{ mod } 4])$,
  - Else
    - For $i = 0$ to 3 do $W_j[i] = W_{j-6}[i] \oplus W_{j-1}[i]$,

where *Rcon(·)* denotes the constants being used. The whitening key is $W_0||W_1||W_2||W_3$, the subkey of round 1 is $W_4||W_5||W_6||W_7$, the subkey of round 2 is $W_8||W_9||W_{10}||W_{11}$, and so on.

3.2.2 A Related-Key Boomerang Attack on 7-Round AES-192

In this section we mount a key recovery attack on 7-round AES-192 using 4 related keys. The cipher is treated as $E(\cdot) = E1(E0(\cdot))$, where $E0$ represents rounds 1 to 4 and includes the whitening key addition as well as the key addition of round 4. $E1$ represents rounds 5 to 7. The notations used in our attack are defined as follows:

- K^a, K^b, K^c, K^d unknown keys (192 bits).
- $K_i^a, K_i^b, K_i^c, K_i^d$ unknown subkeys of round i, where $i \in \{0,1,2,\ldots,9\}$ (128 bits), ($K_0^a, K_0^b, K_0^c, K_0^d$ represent the whitening keys).
- $\Delta K^*, \Delta K'$ known key differences (192 bits).
- $\Delta K_i^*, \Delta K_i'$ known subkey differences of round i (128 bits).
- P_i^l, O_i^l plaintexts processed with the key K^l, $l \in \{a,b,c,d\}$.
- C_i^l, D_i^l ciphertexts processed with the key K^l, $l \in \{a,b,c,d\}$.
- $E0_{K^j}(\cdot)$ 4-round AES-192 encryption from round 1 to 4 under key K^j, where $j \in \{a,b,c,d\}$.
- $E0_{K^j}^{-1}(\cdot)$ 4-round AES-192 decryption from round 4 to 1 under key K^j, where $j \in \{a,b,c,d\}$.
- $E1_{K^j}^{-1}(\cdot)$ 3-round AES-192 decryption from round 7 to 5 under key K^j, where $j \in \{a,b,c,d\}$.
- a is a known non-zero byte difference.
- b is an output difference of S-box for the input difference a.
- $c, d, *$ are unknown non-zero byte differences.
- ? is an unknown byte differences.

The Structure of the Keys

In our attack we use four related but unknown keys K^a, K^b, K^c and K^d. Let K^a be the unknown key the adversary tries to recover. The relation that is required for the attack is:

$$\begin{aligned} K^b &= K^a \oplus \Delta K^* \\ K^c &= K^a \oplus \Delta K' \\ K^d &= K^a \oplus \Delta K^* \oplus \Delta K' \end{aligned}$$

ΔK^* is the key difference used in the first related-key differential for $E0$ and $\Delta K'$ is the key difference used in the second related-key differential for $E1$. The adversary only knows the differences ΔK^* and $\Delta K'$. Recall that the adversary does not know the keys themselves, but can choose the key differences as:

$$\Delta K^* = \begin{array}{|c|c|c|c|c|c|}\hline & & a & a & & \\ \hline & & & & & \\ \hline & & & & & \\ \hline & & & & & \\ \hline \end{array} \quad \text{and} \quad \Delta K' = \begin{array}{|c|c|c|c|c|c|}\hline a & & & & a & \\ \hline & & & & & \\ \hline & & & & & \\ \hline & & & & & \\ \hline \end{array}$$

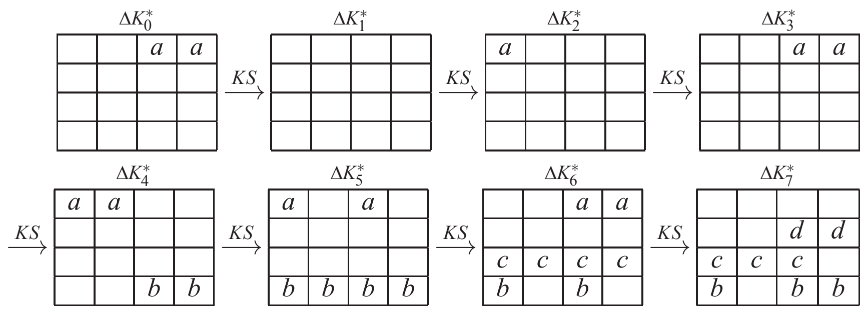


Figure 3.7: Subkey differences derived from ΔK^*

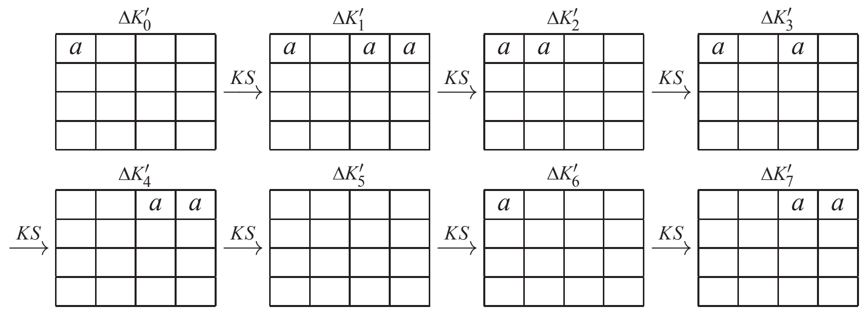


Figure 3.8: Subkey differences derived from $\Delta K'$

Using the key schedule algorithm of AES-192 we can use the key differences ΔK^* and $\Delta K'$ to derive the subkey differences $\Delta K_0^*, \ldots, \Delta K_7^*$ and $\Delta K'_0, \ldots, \Delta K'_7$, respectively.[3] These values are shown in Figure 3.7 and 3.8, respectively.

The Related-Key Differential for $E0$ ($\alpha \to \beta_{out}$)

The difference α of the differential for $E0$ has an a difference in bytes 8 and 12 and zero differences in the remaining bytes. The whitening key addition ARK_0 generates a zero difference in all the bytes of the state. These zero differences remain until ARK_2 is applied, since ΔK_1^* has only zero differences and does not alter the differences in the state matrix. ARK_2 introduces an a difference in byte 0, which is transformed into an unknown non-zero difference after SB_3. MC_3 generates a non-zero difference in bytes 0,1,2 and 3, while ARK_3 inserts an a difference in bytes 8 and 12. After applying SR_4 we have one non-zero byte in columns 0 and 1 and two non-zero bytes in columns 2 and 3. Four non-zero bytes remain after MC_4 in column 0 and 1 with probability one, while we do not know which bytes of column 2 and 3 are non-zero but at most two of

[3]These key differences are also used in [78].

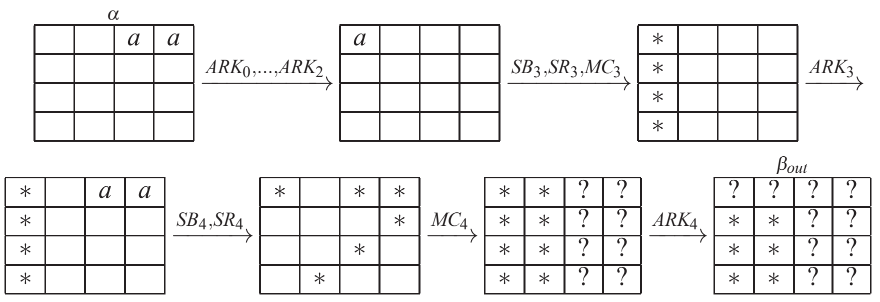


Figure 3.9: The related-key differential for $E0$

these can be zero (at most one in each column). Finally, we call the output difference β_{out}, which is the difference obtained after passing the related-key differential for $E0$. The probability of the differential for $E0$, i.e., the transformation of an α difference into a β_{out} difference is given by

$$Pr(\alpha \rightarrow \beta_{out}) \quad = \quad 1.$$

The related-key differential for $E0$ is shown in Figure 3.9.

The Related-Key Differential for $E1^{-1}$ ($\delta \rightarrow \gamma$)

The difference δ consists of a b difference in byte 0 and an a differences in bytes 8 and 12. The a differences vanish after ARK_7^{-1}, since $\Delta K_7'$ has two a differences in bytes 8 and 12 while the b difference remains. The adversary chooses b such that $\Pr(b \rightarrow a) = 2^{-6}$. If this occurs then the intermediate difference after SB_7^{-1} is equal to the subkey difference $\Delta K_6'$. Hence, all bytes have a zero difference after applying ARK_6^{-1}. All the bytes have a zero difference after ARK_5^{-1}, which are denoted by $\gamma = 0$. The probability of the differential for $E1^{-1}$ is $\Pr(\delta \rightarrow \gamma) = 2^{-6}$. The related-key differential for $E1^{-1}$ is shown in Figure 3.10.

The Related-Key Differential for $E0^{-1}$ ($\beta_{in} \rightarrow \alpha$)

For the following steps we need that the output difference β_{out} of the related-key differential for $E0$ is equal to the output difference β_{in} for the related-key differential for $E0^{-1}$. From the boomerang condition inside the cipher for the two differences γ_1 and γ_2 we know that

$$\beta_{out} \oplus \gamma_1 \oplus \gamma_2 = \beta_{in}$$

holds.

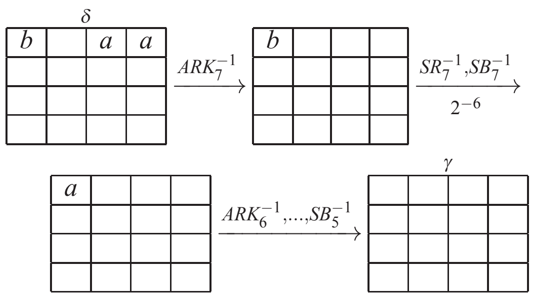


Figure 3.10: The related-key differential for $E1^{-1}$

Let A^a, A^b, B^c, B^d be the internal state after SR_4 when encrypting P^a, P^b, O^c, O^d under K^a, K^b, K^c, K^d, respectively (following the notations of Figure 2.2). Since MC is linear, γ can be expressed as

$$\gamma = K_4^a \oplus MC_4(A^a) \oplus K_4^c \oplus MC_4(B^c) = \overbrace{K_4^a \oplus K_4^c}^{\Delta K_4'} \oplus MC_4(A^a \oplus B^c) \tag{3.4}$$

and as

$$\gamma = K_4^b \oplus MC_4(A^b) \oplus K_4^d \oplus MC_4(B^d) = \overbrace{K_4^b \oplus K_4^d}^{\Delta K_4'} \oplus MC_4(A^b \oplus B^d). \tag{3.5}$$

Equations (3.4) and (3.5) can be combined, which leaves $A^a \oplus A^b = B^c \oplus B^d$. In other words, the MC_4 operation can be undone with probability 1 due to the boomerang condition (3.1). This means that we know that after MC_4^{-1} only bytes 0,7,8,10,12, and 13 have a non-zero difference, while all other bytes have a zero difference. SB_4^{-1} then transforms a non-zero difference into an a difference with probability 2^{-7}. Regarding bytes 8 and 12, we have probability 2^{-14} of doing so. These a differences vanish after ARK_3^{-1}. After MC_3^{-1} and SB_3^{-1} a column with four non-zero bytes is transformed into a byte with a difference in byte 0 with probability 2^{-32}.

This a difference is canceled out by ARK_2^{-1}. We call α the difference that is the output of the related-key differential for $E0^{-1}$, α, has an a difference in the bytes 8 and 12. The differential for $E0^{-1}$ has probability $\Pr(\beta_{in} \to \alpha) = 2^{-14} \cdot 2^{-32} = 2^{-46}$ and is shown in Figure 3.11.

The Attack

The attack recovers one byte of the last round's subkey. Let k_7^a be the 8-bit subkey in byte 0 of the subkey K_7^a. Let $d_{7k_7^i}(X)$, $i \in \{a,b,c,d\}$ be the seventh round decryption of X under the 8-bit subkey k^i in byte 0. The attack is as follows:

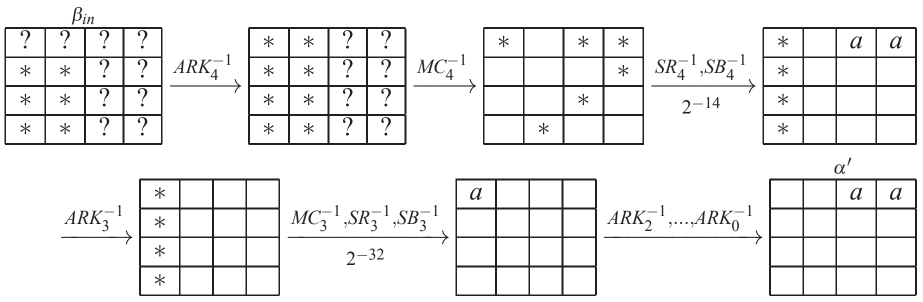


Figure 3.11: The related-key differential for $E0^{-1}$

1. Choose 2^{61} plaintexts P_i^a, $i \in \{1, 2, \ldots, 2^{61}\}$ at random. Ask for the encryption of P_i^a under K^a to obtain the ciphertexts C_i^a, i.e., $C_i^a = E_{K^a}(P_i^a)$.

2. Compute 2^{61} plaintexts $P_i^b = P_i^a \oplus \alpha$, where α is a 128-bit value with value a in bytes 8 and 12, while the remaining bytes are zero. Ask for the encryption of the P_i^b under K^b, where $K^b = K^a \oplus \Delta K^*$ to obtain the ciphertexts C_i^b, i.e., $C_i^b = E_{K^b}(P_i^b)$.

3. Compute 2^{61} ciphertexts D_i^c, such that $D_i^c = C_i^a \oplus \delta$, where δ is a fixed value with a b byte value in byte 0 and an a difference in bytes 8 and 12. Ask for the decryption of D_i^c under K^c to obtain the plaintexts O_i^c, i.e., $O_i^c = E_{K^c}^{-1}(D_i^c)$.

4. Compute 2^{61} ciphertexts D_i^d, such that $D_i^d = C_i^b \oplus \delta$, where δ is as in Step 3. Ask for the decryption of D_i^d under K^d to obtain the plaintexts O_i^d, i.e., $O_i^d = E_{K^d}^{-1}(D_i^d)$.

5. Store only those quartets $(P_i^a, P_j^b, O_i^c, O_j^d)$, $i, j \in \{1, 2, \ldots, 2^{61}\}$ in the set ϕ where $O_i^c \oplus O_j^d$ have an a difference in bytes 8 and 12, while the remaining byte differences are zero.

6. For each 8-bit candidate key k_7^a, compute $k_7^b = k_7^a, k_7^c = k_7^a$, and $k_7^d = k_7^a$.

   For each quartet remaining after the test in Step 5:

   6.1. Partially decrypt a ciphertext quartet $(C_i^a, C_j^b, D_i^c, D_j^d)$, i.e., $\overline{C_i^a} = d_{7k_7^a}(C_i^a)$, $\overline{C_j^b} = d_{7k_7^b}(C_j^b)$, $\overline{D_i^c} = d_{7k_7^c}(D_i^c)$, and $\overline{D_j^d} = d_{7k_7^d}(O_j^d)$.

   6.2. Increase the counter for the used 8-bit subkey k_7^a by one if $\overline{C_i^a} \oplus \overline{D_i^c}$ and $\overline{C_j^b} \oplus \overline{D_j^d}$ have difference a in byte 0.

7. Output the 8-bit subkeys with the highest counter, which are k_7^a and $k_7^a \oplus \delta$.

Analysis of the Attack

Using $\#PP = 2^{61}$ plaintext pairs we obtain the same amount of quartets. A right quartet occurs with probability

$$\begin{aligned} Pr_c &= \Pr(\alpha \to \beta_{out}) \cdot (\Pr(\gamma \leftarrow \delta))^2 \cdot \Pr(\alpha \leftarrow \beta_{in}) \\ &= 1 \cdot (2^{-6})^2 \cdot 2^{-46} = 2^{-58}, \end{aligned}$$

which is the probability that all related-key differentials are fulfilled. A random value of a plaintext difference is equal to the difference α with probability $Pr_f = 2^{-128}$. Thus, after Step 5 we have about

$$\#C = \#PP \cdot Pr_c = 2^{61} \cdot 2^{-58} = 2^3$$

right quartets and

$$\#F = \#PP \cdot Pr_f = 2^{61} \cdot 2^{-128} = 2^{-67}$$

wrong quartets stored in ϕ. The data and time complexities of Step 6 and 7 are negligible compared to the other steps before, since we expect to have only 2^3 quartets analyzed.

A wrong key is counted in Step 6.2 with probability 2^{-12}. This is the probability that an active byte with a b difference has an a difference after SB_7^{-1}. The output of Step 7 is k_7^a and $k_7^a \oplus b$, since both values are expected to have the highest counter. We expect that the right key has a counter of about $2^3 + 2^{-79} \approx 2^3$ quartets in total while for each wrong key we expect to count $2^3 \cdot 2^{-12} + 2^{-67} \cdot 2^{-12} \approx 2^{-9}$ quartets. The adversary can only distinguish k and $k \oplus b$ using exhaustive search of the remaining key bits and later check for correctness.

Using the Poisson distribution we can compute the success rate of our attack. The probability that the counter for each wrong key is at least 5 assuming $Y_i \sim Poisson(\mu = 2^{-9})$ is

$$\Pr(Y \geq 5) = e^{-2^{-9}} \cdot \frac{(2^{-9})^5}{5!} \approx 2^{-51}.$$

For all the $2^8 - 1$ wrong keys (but $k_7^a \oplus b$) used in our analysis we expect about $2^8 \cdot 2^{-51} = 2^{-43}$ wrong keys which have a count of at least 5 quartets. The probability that the right key has a count of at least 5 quartets using $Z \sim Poisson(\mu = 2^3)$ is

$$\Pr(Z \geq 5) \approx 0.90.$$

The data complexity of this attack is determined by Steps 1, 2, 3 and 4 which is about $2^{63} = 2^2 \cdot 2^{61}$ adaptive chosen plaintexts and ciphertexts. The time complexity is about $2^{63} = 2^2 \cdot 2^{61}$ seven round AES-192 encryptions.

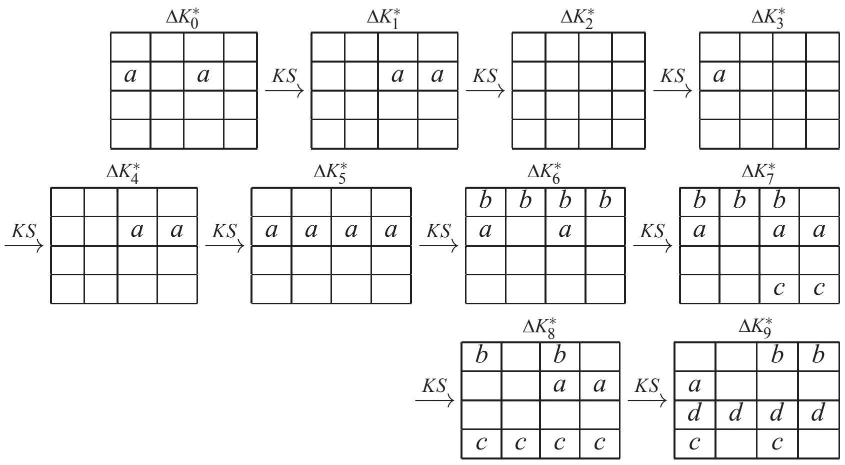


Figure 3.12: Subkey differences derived from ΔK^*

3.2.3 A Related-Key Boomerang Attack on 9-Round AES-192

Our related-key boomerang attack can be extended to attack 9 rounds of AES-192.

The Structure of the Keys

In the attack we use the key differences ΔK^* and $\Delta K'$ as shown below and derive the subkey differences $\Delta K_0^*, \dots, \Delta K_9^*$ and $\Delta K_0', \dots, \Delta K_9'$, respectively. These values are shown in Figures 3.12 and 3.13, respectively. This attack works for weak key classes for which the subkeys satisfy the differences shown in Figure 3.12 and 3.13, where an *a* difference transforms into a *b* difference. We assume that we are in this weak key class for our attack. One can transform the attack into a more general attack by increasing the number of keys being used.

The Related-Key Differential for $E0$ ($\alpha \rightarrow \beta_{out}$)

The related-key differential for $E0$ for rounds 1 to 5 is described as follows. The difference α of $E0$ has a non-zero difference in bytes 1,2,6,7,8, 10,11, 12 and byte 9 has an *a* difference. After SR_1 all non-zero bytes are found in columns 2 and 3. A column with four non-zero bytes is transformed into a column having an *a* difference in a fixed position after MixColumns with probability of 2^{-64}. The two *a* differences in bytes 9 and 13 are canceled out by the key addition ARK_1. Thus, each byte of the state matrix has a zero difference until ARK_3 introduces an *a* difference in byte 1, which is transformed to a non-zero difference by SB_4 and to four non-zero bytes by MC_4. After

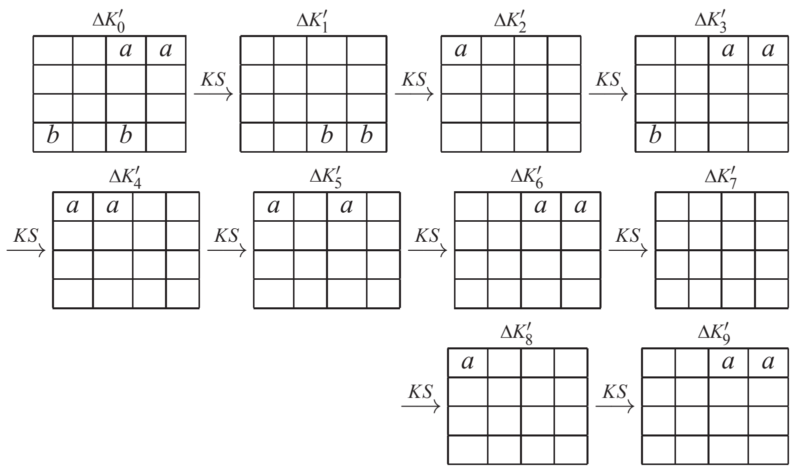


Figure 3.13: Subkey differences derived from $\Delta K'$

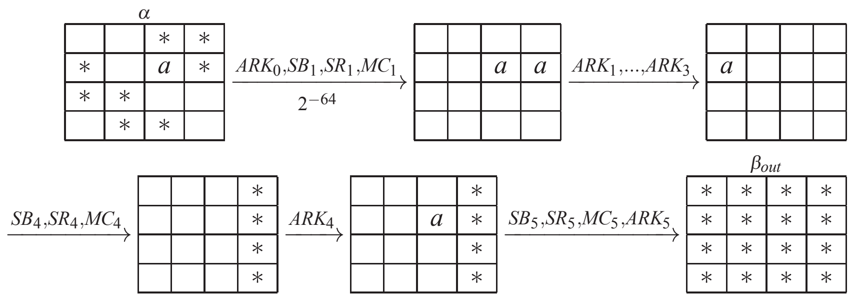


Figure 3.14: The related-key differential for $E0$

ARK_4 the state matrix has an a difference in byte 9 and non-zero differences in bytes 11,12,13,14. The difference which occurs after ARK_5 is the output difference which we call β_{out}, where all the bytes have non-zero difference except for bytes 4 and 7 which are unknown. The probability of the differential for $E0$, i.e., the transformation of an α difference into a β_{out} difference is given by

$$Pr(\alpha \rightarrow \beta_{out}) \quad = \quad 2^{-64}.$$

The related-key differential for $E0$ is shown in Figure 3.14.

The Related-Key Differential for $E1^{-1}$ ($\delta \rightarrow \gamma$)

From the decryption direction of the related-key boomerang distinguisher, the related-key differential for $E1^{-1}$ is used in rounds $9-6$ with the subkey differences of $\Delta K'$.

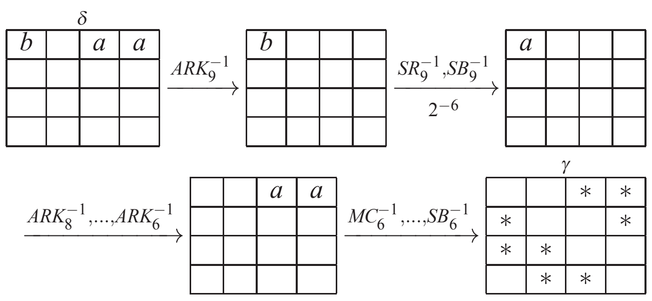


Figure 3.15: The related-key differential for $E1^{-1}$

The difference δ consists of a b difference in byte 0 which is chosen such that $\Pr(b \to a) = 2^{-6}$ and two a differences in bytes 8 and 12 after applying the S-box appear. The a differences vanish after ARK_9^{-1}, since $\Delta K_9'$ has two a differences in bytes 8 and 12 while the other bytes of $\Delta K_9'$ are zero. Only the b difference in byte 0 remains. SB_9^{-1} generates an a difference in byte 0 with probability 2^{-6}. If this occurs, the intermediate difference after SB_9^{-1} is equal to the subkey difference $\Delta K_8'$. Hence, all the bytes have a zero difference after applying ARK_8^{-1}. Passing ARK_6^{-1} the state matrix has two a differences in bytes 8, 12. We call γ the intermediate difference remaining after SB_6^{-1}. This intermediate difference has eight non-zero difference in bytes 1,2, 6,7,8,11,12 and 13. The probability of the differential $\delta \to \gamma$ in $E1^{-1}$ is $\Pr(\delta \to \gamma) = 2^{-6}$. The related-key differential for $E1^{-1}$ is shown in Figure 3.15.

The Related-Key Differential for $E0^{-1}$ ($\beta_{in} \to \alpha$)

As in the 7-round attack on AES-192 we need that the output difference β_{out} of the related-key differential for $E0$ is then equal to the difference β_{in} for the related-key differential for $E0^{-1}$. If this holds the MixColumns operation of round 5 can be undone with probability one. For a detailed description we refer to the analysis of our 7 round attack.

To achieve that β_{out} equals β_{in}, the differences γ_1 and γ_2 have to be equal. This happens with probability 2^{-56} since an a difference can be one of 2^7-1 values after an inverse S-box transformation and MixColumns is a linear operation. We know from the boomerang condition that $\beta_{out} \oplus \gamma_1 \oplus \gamma_2 = \beta_{out} = \beta_{in}$ then holds whenever $\gamma_1 = \gamma_2$ and MC_5 can be undone with probability one. This means that we know that a non-zero byte difference occurs after MC_5^{-1} only in bytes 3, 5, 6, 9 and 12, while the other bytes have a zero difference. SB_5^{-1} then transforms a non-zero difference in byte 9 into an a difference with probability 2^{-7} (as this difference was produced by a difference in the differential for $E0$). Four non-zero bytes remain after ARK_4^{-1} in the third column. With probability 2^{-24} we get a non-zero difference in byte 13. This turns into an a difference SB_4^{-1} which occurs with the probability 2^{-8}. The further steps operate such that the output difference of $E0^{-1}$ has non-zero differences in bytes 1,2,6,7,8,11,12,13 and an a difference in byte 9 they occur with probability one. The related-key differential for $E0^{-1}$ has the probability $\Pr(\beta_{in} \rightarrow \alpha) = 2^{-39}$ and is shown in Figure 3.16.

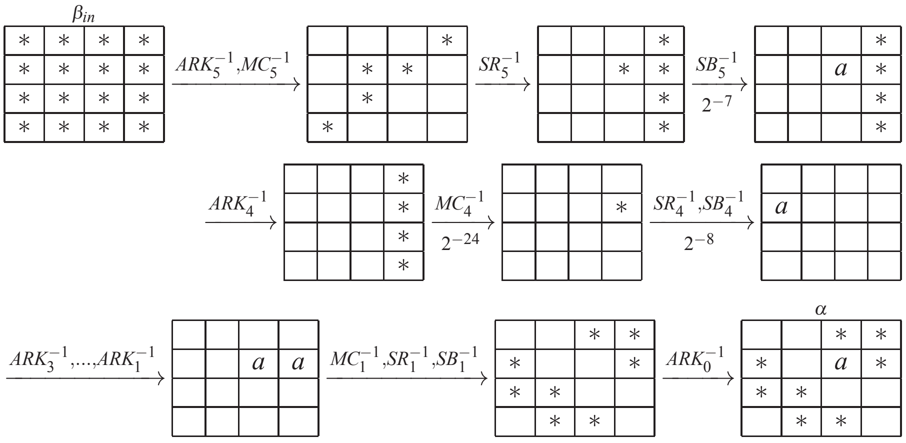


Figure 3.16: The related-key differential for $E0^{-1}$

The Attack

1. Choose 2^{46} structures of 2^{64} plaintexts $P^a_{i,j}$, $i \in \{1,2,\ldots,2^{64}\}$, $j \in \{1,2,\ldots, 2^{46}\}$ where bytes 0, 3, 4, 5, 9, 10, 14, 15 are fixed. Ask for the encryption of $P^a_{i,j}$ under K^a to obtain the ciphertexts $C^a_{i,j}$, i.e., $C^a_{i,j} = E_{K^a}(P^a_{i,j})$.
2. Compute 2^{46} structures of 2^{64} plaintexts $P^b_{i,j} = P^a_{i,j} \oplus \alpha$, where α is a 16-byte value of which byte 9 is a and all the other bytes are zero. Ask for the encryption of $P^b_{i,j}$ under K^b, where $K^b = K^a \oplus \Delta K^*$ to obtain the ciphertexts $C^b_{i,j}$, i.e., $C^b_{i,j} = E_{K^b}(P^b_{i,j})$.
3. Compute 2^{46} structures of 2^{64} ciphertexts $D^c_{i,j}$, $i \in \{1,2,\ldots,2^{64}\}$, $j \in \{1,2,\ldots, 2^{46}\}$ such that $D^c_{i,j} = C^a_{i,j} \oplus \delta$, where δ is the output difference depicted in Figure 3.15. Ask for the decryption of $D^c_{i,j}$ under $K^c = K^a \oplus \Delta\tilde{K}'$ to obtain the plaintexts $O^c_{i,j}$, i.e., $O^c_{i,j} = E^{-1}_{K^c}(D^c_{i,j})$.
4. Compute 2^{46} structures of 2^{64} ciphertexts $D^d_{i,j}$, $i \in \{1,2,\ldots,2^{64}\}$, $j \in \{1,2,\ldots, 2^{46}\}$ such that $D^d_{i,j} = C^b_{i,j} \oplus \delta$ where δ is as in Step 3.1. Ask for the decryption of $D^d_{i,j}$ under $K^d = K^a \oplus \Delta K^* \oplus \Delta\tilde{K}'$ to obtain the plaintext $O^d_{i,j}$, i.e., $O^d_{i,j} = E^{-1}_{K^d}(D^d_{i,j})$.

5. Store only those quartets $(P^a_{i,j}, P^b_{l,j}, O^c_{i,j}, O^d_{l,j})$ where $O^c_{i,j} \oplus O^d_{l,j}$ have a zero byte differences in bytes 0, 3, 4, 5, 10, 14, 15 and an a difference in byte 9.

6. Guess the 8-bit subkey $\overline{k^a_9}$ of K^a_9 in byte 0 and compute $\overline{k^b_9}, \overline{k^c_9}, \overline{k^d_9}$, respectively.

   6.1. Partially decrypt each remaining quartet $(C^a_{i,j}, C^b_{l,j}, D^c_{i,j}, D^d_{l,j})$ under $\overline{k^a_9}, \overline{k^b_9}, \overline{k^c_9}, \overline{k^d_9}$, respectively.

   6.2. Check if $d_{\overline{k^a_9}}(C^a_{i,j}) \oplus d_{\overline{k^c_9}}(D^c_{i,j})$ and $d_{\overline{k^b_9}}(C^b_{l,j}) \oplus d_{\overline{k^d_9}}(D^d_{l,j})$ have an a-difference after SB_9^{-1} in byte 0. Record $(\overline{k^a_9})$ and discard all quartets which do not satisfy the condition.

7. Guess the 32-bit subkey of bytes 2,7,8,13 of k'^a_0 of K^a_0 and compute $k'^b_0 = k'^c_0 = k'^d_0 = k'^a_0$ *(ΔK^*_0 and $\Delta K'_0$ are zero in these four bytes)*

   7.1. Partially encrypt each remaining quartet $(P^a_{i,j}, P^b_{l,j}, O^c_{i,j}, O^d_{l,j})$ under $k'^a_0, k'^b_0, k'^c_0, k'^d_0$, respectively.

   7.2. Check if $e_{k'^a_0}(P^a_{i,j}) \oplus e_{k'^b_0}(P^b_{l,j})$ and $e_{k'^c_0}(O^c_{i,j}) \oplus e_{k'^d_0}(O^d_{l,j})$ have an a difference in byte 9 while the remaining bytes are zero. Record $(\overline{k^a_9}, k'^a_0)$ and discard all the quartets which do not satisfy the condition.

8. Guess the 32-bit subkey k^{*a}_0 of K^a_0 in bytes 1,6,11,12 and compute $k^{*b}_0 = k^{*a}_0 \oplus M_1$, with the 32-bit value $M_1 = (a,0,0,0)$, compute $k^{*c}_0 = k^{*a}_0 \oplus M_2$, with the 32-bit value $M_2 = (0,0,b,a)$ and compute $k^{*d}_0 = k^{*a}_0 \oplus M_1 \oplus M_2$.

   8.1. Partially encrypt each remaining quartet $(P^a_{i,j}, P^b_{l,j}, O^c_{i,j}, O^d_{l,j})$ under $k^{*a}_0, k^{*b}_0, k^{*c}_0, k^{*d}_0$, respectively.

   8.2. Check if $e_{k^{*a}_0}(P^a_{i,j}) \oplus e_{k^{*b}_0}(P^b_{l,j})$ and $e_{k^{*c}_0}(O^c_{i,j}) \oplus e_{k^{*d}_0}(O^d_{l,j})$ have an a difference in byte 13. If there exist at least 5 boomerang quartets passing this test, record $(\overline{k^a_9}, k'^a_0, k^{*a}_0)$ and all the remaining quartets and then go to Step 9. Otherwise, repeat Step 8 with another guessed key. If all the possible keys are tested, then repeat Step 7 with another guessed key. If all the possible keys are tested, then repeat Step 6 with another guessed key.

9. Output the keys $(\overline{k^a_9}, k'^a_0, k^{*a}_0)$.

Analysis of the Attack

Using 2^{64} plaintexts we can combine them in about $\left(2^{64}\right)^2 = 2^{128}$ quartets. To increase the amount of data, we use 2^{46} structures to obtain a total of $2^{46} \cdot 2^{128} = 2^{174}$ quartets.

The following analysis of the data and time complexity regards only one structure of data. The data complexity of Step 1, 2, 3 and 4 is $2^2 \cdot 2^{64} = 2^{66}$ adaptive chosen plaintexts, while the time complexity is about $2^2 \cdot 2^{64} = 2^{66}$ encryptions for Steps 1 to 4.

Due to the 64-bit filtering condition of Step 5 about $2^{128} \cdot 2^{-64} = 2^{64}$ of the 2^{128} quartets in each structure remain after this step.

Step 6.1 takes about $(1/9) \cdot (1/16) \cdot 2^8 \cdot 2^2 \cdot 2^{64} = 2^{66.83}$ nine round encryptions, since for each of the 2^8 keys we have to encrypt the 2^{64} quartets in one S-box of one round. The number of remaining quartets after Step 6.2 is $2^{64} \cdot 2^{-12} = 2^{52}$, since we have a 6-bit filtering on both pairs of a quartet. The time complexity of Step 7.1 is about $(1/9) \cdot (4/16) \cdot 2^{32} \cdot 2^8 \cdot 2^2 \cdot 2^{52} = 2^{88.83}$ nine round encryptions. Due to the 32-bit filtering on both pairs we obtain about $2^{52} \cdot 2^{-64} = 2^{-12}$ quartets after Step 7.2. The time complexity of Step 8.1 is negligible, while about $2^{-12} \cdot 2^{-64} = 2^{-76}$ quartets remain after this step for each key.

Now let us look at all structures together. About $2^{46} \cdot 2^{-76} = 2^{-30}$ quartets are counted in Step 8.2 with a wrong key. A right quartet occurs with probability

$$\begin{aligned} Pr_c &= \Pr(\alpha \to \beta_{out}) \cdot (\Pr(\gamma \leftarrow \delta))^2 \cdot \Pr(\beta_{out} = \beta_{in}) \cdot \Pr(\alpha \leftarrow \beta_{in}) \\ &= 2^{-64} \cdot (2^{-6})^2 \cdot 2^{-56} \cdot 2^{-39} = 2^{-171}, \end{aligned}$$

since all related-key differential conditions are fulfilled, thus we expect about $2^3 = 2^{174} \cdot 2^{-171}$ right quartets among all quartets. For the right key we expect a counter of $2^3 + 2^{-30} \approx 2^3$ quartets, while for each wrong key we expect 2^{-30} quartets to be counted.

Using the Poisson distribution we can compute the success rate of our attack. The probability that the counter for each wrong key is at least 4 using $Y_i \sim Poisson(\mu = 2^{-30})$ is

$$\Pr(Y \geq 4) = e^{-2^{-30}} \cdot \frac{(2^{-30})^4}{4!} \approx 2^{-124}.$$

For all the $2^{72} - 1$ wrong keys used in our analysis we expect about $2^{72} \cdot 2^{-124} = 2^{-52}$ wrong keys which have a count of at least 4 quartets. The probability that the right key has a count of at least 4 quartets using $Z \sim Poisson(\mu = 2^3)$ is

$$\Pr(Z \geq 4) \approx 0.95.$$

The data complexity is about $2^2 \cdot 2^{46} \cdot 2^{64} = 2^{112}$ adaptive chosen plaintexts and ciphertexts and the time complexity is about $2^{46} \cdot 2^{88.83} \approx 2^{134.8}$ nine round AES-192 encryptions.

3.3 Related-Key Boomerang Attack on 9-Round AES-256

In this section we propose a 9-round related-key boomerang attack on AES-256. In addition to the previous attack, we introduce a key differential that has a probability below one. Up to now, this is the first analysis, which used this technique on AES-256. Table 3.3 summarizes existing attacks on AES-256 and our new attack on 9 rounds.

3.3.1 Description of AES-256

AES-256 has 14 rounds and a 256-bit key. As AES' encryption is the same as AES-192 (beside the number of rounds), we only describe the different key schedule. As before, let W_i be a 32-bit word and $W_{i,j}$ the *i*-th byte in W_j, then the 256-bit key is represented by $W_0||W_1||W_2||\dots||W_7$. The 256-bit key schedule algorithm works as follows:

- For $j = 8$ to 59
  - If $j \equiv 0 \mod 8$, then
    - $W_{0,j} = W_{0,j-8} \oplus SB(W_{1,j-1}) \oplus Rcon(j/8)$,
    - For $i = 1$ to 3
      - $W_{i,j} = W_{i,j-8} \oplus SB(W_{i+1 \bmod 4,j-1})$,
  - Else if $j \equiv 4 \mod 8$, then
    - For $i = 0$ to 3 do $W_{i,j} = W_{i,j-8} \oplus SB(W_{i,j-1})$,
  - Else
    - For $i = 0$ to 3 do $W_{i,j} = W_{i,j-8} \oplus W_{i,j-1}$,

where *Rcon(·)* denotes the constants being used. The whitening key is $W_0||W_1||W_2||W_3$, the subkey of round 1 is $W_4||W_5||W_6||W_7$, the subkey of round 2 is $W_8||W_9||W_{10}||W_{11}$ and so on.

3.3.2 A Related-Key Boomerang Attack on 9-Round AES-256

In this section we mount a key recovery attack on 9-round AES-256 using four related keys. We assume that we are in a weak key class where all the subkey differences occur. It might be possible to transform our attack into a general attack by using about 2^{12} quartets of keys. This amount of keys is necessary such that we can expect about one key quartet, which fulfills the required subkey difference.

Table 3.3: Existing attacks on round reduced AES-256

Attack	# **Rounds**	# **Keys**	**Data / Time**	**Source**
Square	7	1	2^{32}CP / 2^{200}	[104]
Collision	7	1	2^{32}CP / 2^{140}	[69]
Partial Sums	7	1	$2^{128} - 2^{119}$CP / 2^{120}	[49]
Partial Sums	7	1	$21 \cdot 2^{32}$CP / 2^{172}	[49]
Impossible Differential	7	1	$2^{92.5}$CP / $2^{250.5}$CP	[127]
Impossible Differential	7	1	2^{92}CP / $2^{163}MA$	[100]
Impossible Differential	7	1	$2^{115.5}$CP / 2^{119}	[157]
Impossible Differential	7	1	$2^{113.8}$CP / $2^{118.8}MA$	[100]
Meet-in-the-Middle	7	1	2^{32}ACPC / 2^{208}	[43]
Partial Sums	8	1	$2^{128} - 2^{119}$CP / 2^{204}	[49]
Meet-in-the-Middle	8	1	2^{32}ACPC / 2^{209}	[43]
Impossible Differential	8	1	$2^{116.5}$CP / $2^{247.5}$	[157]
Impossible Differential	8	1	$2^{89.1}$CP / $2^{229.7}MA$	[100]
Impossible Differential	8	1	$2^{111.1}$CP / $2^{227.8}MA$	[100]
Related-Key	8	2	2^{31}RK-CP / 2^{31}	[24]
Relate Subkey	8	2	$2^{26.5}$RK-CC / $2^{26.5}$	[24]
Meet-in-the-Middle	8	1	2^{113+n} CP / 2^{196+n}	[48]
Partial Sums	9	256	2^{85}CP / $5 \cdot 2^{224}$	[49]
Related-Key Rectangle	9	4	2^{99}RK-CP / 2^{120}	[90]
Related-Key Boomerang†	9	4	2^{104} RK-ACPC / 2^{127}	Sec. 3.3.2
Related-Key	9	2	2^{38}RK-CP / 2^{39}	[24]
Related Subkey	9	2	2^{32}RS-CC / 2^{32}	[24]
Related-Key Rectangle	10	256	$2^{114.9}$RK-CP / $2^{171.8}$	[17]
Related-Key Rectangle‡	10	64	$2^{113.9}$ RK-CP / $2^{172.8}$	[90]
Related Subkey	10	2	2^{49}RS-CC / 2^{48}	[24]
Related Subkey	10	2	2^{45}RS-CC / 2^{44}	[24]
Related-Key Boomerang	13	4	2^{76}RK-ACPC / 2^{76}	[26]
Related-Key Rectangle	14	4	2^{99}RS-ACPC / 2^{99}	[25]

CP: Chosen Plaintexts, CP: Chosen Ciphertexts, ACPC: Adaptive Chosen Plaintexts and Ciphertexts, mem: Memory Usage.

RK: Related-Key, RS: Related-Subkey, Time: Encryption units.

† : the attack is in a weak key class of size 2^{244} keys.

‡ : the attack is based on the 10-round attack of Biham et al. [17], which has some flaws. The table shows the corrected complexity from [90].

The cipher is represented as $E(P) = E1(E0(P))$, where $E0$ covers rounds 1 to 5 and includes the whitening key addition as well as the key addition of round 5. $E1$ is covering rounds 6 to 9. The notations used in our attack are defined as follows:

- K^a, K^b, K^c, K^d unknown keys each (256 bits).
- $K_i^a, K_i^b, K_i^c, K_i^d$ unknown subkeys of round i, where $i \in \{0, 1, 2, \ldots, 10\}$ each (128 bits), (where $K_0^a, K_0^b, K_0^c, K_0^d$ represent the whitening keys).

The Structure of the Keys

In our attack the relation that is required for the attack is:

$$\begin{aligned} K^b &= K^a \oplus \Delta K^* \\ K^c &= K^a \oplus \Delta K' \\ K^d &= K^a \oplus \Delta K^* \oplus \Delta K' \end{aligned}$$

ΔK^* is the key difference used in the first related-key differential for $E0$ and $\Delta K'$ is the key difference used in the second related-key differential for $E1$. The adversary only chooses the differences ΔK^* and $\Delta K'$ but does not know the keys. The chosen key differences are:

$$\Delta K^* = \begin{array}{|c|c|c|c|c|c|c|c|} \hline & & & & a & a & & \\ \hline & & & & & & & \\ \hline & & & & & & & \\ \hline & & & & & & & \\ \hline \end{array} \quad \text{and} \quad \Delta K' = \begin{array}{|c|c|c|c|c|c|c|c|} \hline & b & & & & a & & \\ \hline & & & & & & & \\ \hline & & & & & & & \\ \hline & & & & & & & \\ \hline \end{array}$$

Using the key schedule algorithm of AES-256, we can use the key differences ΔK^* and $\Delta K'$ to derive the subkey differences $\Delta K_0^*, \ldots, \Delta K_{10}^*$ and $\Delta K_0', \ldots, \Delta K_{10}'$, respectively.[4] These values are shown in Figures 3.17 and 3.18, respectively. In the weak key class the transition $a \to b$ occurs with probability one, thus, the subkey differences $\Delta K_0^*, \ldots, \Delta K_{10}^*$ occur with probability one and the subkey differences $\Delta K_0', \ldots, \Delta K_{10}'$ occur also with probability one for (K^a, K^c) and (K^b, K^d) simultaneously.

Note that the key differential $\Delta K'$ has a probability of 2^{-6} to occur, which increases the necessary number of keys for the attack. We have to pay 2^{-12} in probability since we have two key pairs where this occur.

The Related-Key Differential for $E0$ ($\alpha \to \beta_{out}$)

The difference α of the differential for $E0$ has a non-zero difference in bytes 0,3,4,5,9, 10,14, and 15. After SR_1 all non-zero byte differences are in columns 0 and 1. With

[4]The key difference ΔK^* is also used in [90].

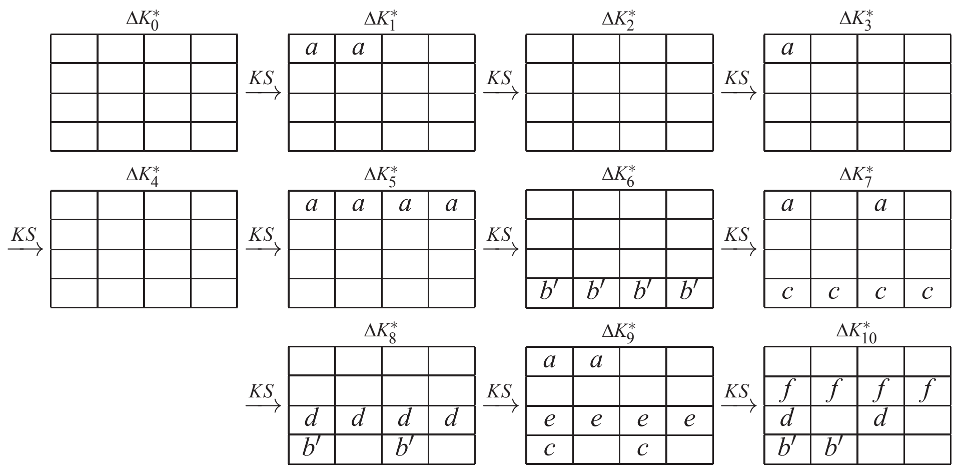


Figure 3.17: Subkey differences derived from ΔK^*

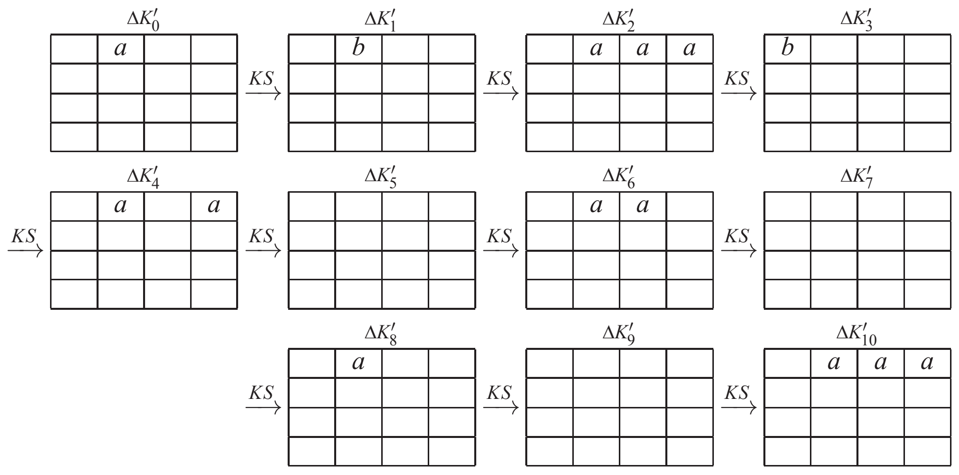
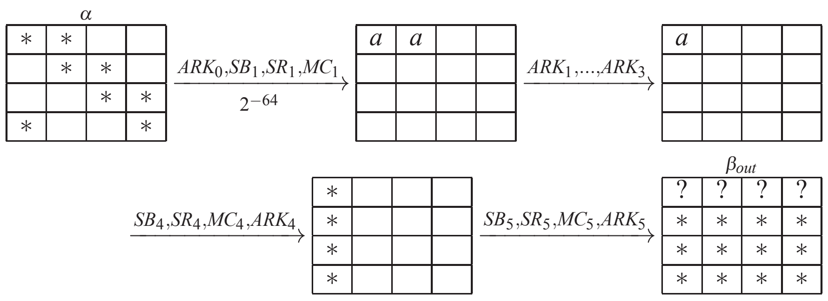


Figure 3.18: Subkey differences derived from $\Delta K'$



α
ARK_0, SB_1, SR_1, MC_1
2^{-64}
$ARK_1, \ldots, ARK_3$
SB_4, SR_4, MC_4, ARK_4
SB_5, SR_5, MC_5, ARK_5
β_{out}

Figure 3.19: The related-key differential for $E0$

probability 2^{-64} we obtain an a difference in bytes 0 and 4 after MC_1. The two a differences in bytes 0 and 4 are canceled out by the key addition ARK_1. Thus, each byte of the state matrix has a zero difference until ARK_3 introduces an a difference in byte 0, which is transformed to a non-zero difference by SB_4 and to four non-zero byte differences after MC_4. The state matrix has a non-zero differences in bytes 0, 1, 2, and 3. We denote the difference which occurs after ARK_5 by β_{out}, where all the bytes have non-zero difference. The differential for $E0$ has a probability of 2^{-64} to occur, and is shown in Figure 3.19.

The Related-Key Differential for $E1^{-1}$ ($\delta \to \gamma$)

From the bottom up direction, the differential for $E1^{-1}$ is used with the subkey differences of $\Delta K'$. The difference δ consists of a b difference in byte 4. The difference b is chosen such that $\Pr(b \to a) = 2^{-6}$ holds (note that this holds for the inverse S-box), using the inverse S-box. In this case SB_9^{-1} generates an a difference with probability 2^{-6} This happens in byte 4. If this occurs the difference of the intermediate encryption values after SB_9^{-1} is equal to the subkey difference $\Delta K'_8$. Hence, all bytes have a zero difference after applying ARK_8^{-1}. Passing ARK_6^{-1} the state matrix has two a differences in bytes 4 and 8. We denote the difference of the intermediate encryption values remaining after SB_6^{-1} by γ. This text difference has eight non-zero difference in bytes 2,3,4,7,8,9,13 and 14. The probability of the differential for $E1^{-1}$ is $\Pr(\delta \to \gamma) = 2^{-6}$. The related-key differential for $E1^{-1}$ is shown in Figure 3.20.

The Related-Key Differential for $E0^{-1}$ ($\beta_{in} \to \alpha$)

For the following steps we need that the output difference β_{out} of the related-key differential for $E0$ is equal to the difference β_{in} for the related-key differential for $E0^{-1}$.

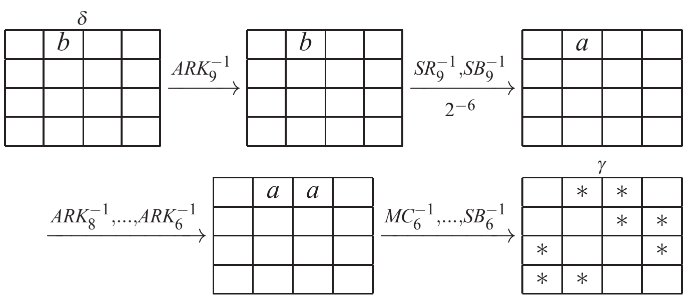


Figure 3.20: The related-key differential for $E1^{-1}$

As in our attack on AES-192 we know that

$$\beta_{out} \oplus \gamma_1 \oplus \gamma_2 = \beta_{in}$$

holds. If γ_1 and γ_2 are equal in all bytes, we simply write γ and obtain :

$$\beta_{out} \oplus \gamma \oplus \gamma = \beta_{out} = \beta_{in}. \tag{3.6}$$

Using the differentials above, the differences γ_1 and γ_2 are equal with probability 2^{-56}. This leaves β_{out} equal to β_{in}. If this occurs, we know that MC_5 can be undone with probability one. This means that we know that a non-zero byte difference occurs after MC_5^{-1} in bytes 0,7,10 and 13, while the other bytes are zero. Four non-zero bytes remain after ARK_4^{-1} in column 0. With probability 2^{-24} MC_4^{-1} generates a non-zero difference in byte 0 while the remaining byte differences are zero. After the next S-box operation we have an a difference with probability 2^{-8}. The following steps operate such that the output difference of $E0^{-1}$ has non-zero differences in bytes 1,2,6,7,8,11,12 and 13. The differential for $E0^{-1}$ has the probability $\Pr(\beta_{in} \to \alpha) = 2^{-32}$ and is shown in Figure 3.21.

The Attack

1. Choose 2^{38} structures S_j, $j \in \{1,2,\ldots,2^{38}\}$ of 2^{64} plaintexts $P^a_{i,j}$, $i \in \{1,2,\ldots,2^{64}\}$ each where the bytes 0, 3, 4, 5, 9, 10, 14, 15 are fixed and the other bytes have all possible values. Ask for the encryption of $P^a_{i,j}$ under K^a to obtain the ciphertexts $C^a_{i,j}$, i.e., $C^a_{i,j} = E_{K^a}(P^a_{i,j})$.
2. Compute 2^{38} structures S'_j, $j \in \{1,2,\ldots,2^{38}\}$ of 2^{64} plaintexts $P^b_{i,j} = P^a_{i,j}$ each. Ask for the encryption of $P^b_{i,j}$ under K^b, where $K^b = K^a \oplus \Delta K^*$ to obtain the ciphertexts $C^b_{i,j}$, i.e., $C^b_{i,j} = E_{K^b}(P^b_{i,j})$.
3. Compute 2^{38} structures S^*_j, $j \in \{1,2,\ldots,2^{38}\}$ of 2^{64} ciphertexts $D^c_{i,j}$, such that $D^c_{i,j} = C^a_{i,j} \oplus \delta$ each, where δ is a fixed 128-bit value of which byte

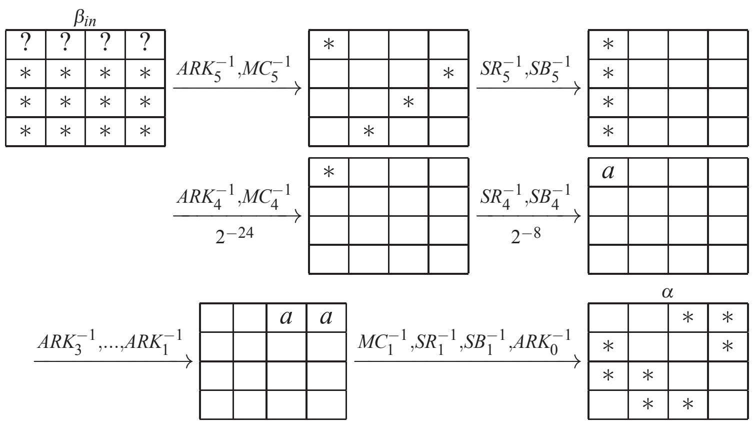


Figure 3.21: The related-key differential for $E0^{-1}$

4 has a known value b while the remaining bytes are zero. Ask for the decryption of $D^c_{i,j}$ under $K^c = K^a \oplus \Delta\tilde{K}'$ to obtain the plaintexts $O^c_{i,j}$, i.e., $O^c_{i,j} = E^{-1}_{K^c}(D^c_{i,j})$.

4. Compute 2^{38} structures S'^{*}_j, $j \in \{1,2,\ldots,2^{38}\}$ of 2^{64} ciphertexts $D^d_{i,j}$, such that $D^d_{i,j} = C^b_{i,j} \oplus \delta$ each where δ is as in Step 3. Ask for the decryption of $D^d_{i,j}$ under $K^d = K^a \oplus \Delta K^* \oplus \Delta\tilde{K}'$ to obtain the plaintexts $O^d_{i,j}$, i.e., $O^d_{i,j} = E^{-1}_{K^d}(D^d_{i,j})$.

5. Store only those quartets $(P^a_{i,j}, P^b_{l,j}, O^c_{i,j}, O^d_{l,j})$ where $O^c_{i,j} \oplus O^d_{l,j}$ have zero differences in bytes 0, 3, 4, 5, 9, 10, 14, and 15.

6. Guess the 8-bit subkey $\overline{k^a}_9$ of K^a_9, i.e., byte 4, and compute $\overline{k^b}_9, \overline{k^c}_9, \overline{k^d}_9$, respectively.

   6.1. Partially decrypt each quartet $(C^a_{i,j}, C^b_{l,j}, D^c_{i,j}, D^d_{l,j})$ remaining after Step 5 under $\overline{k^a}_9, \overline{k^b}_9, \overline{k^c}_9, \overline{k^d}_9$, respectively.

   6.2. Check if $d_{\overline{k^a}_9}(C^a_{i,j}) \oplus d_{\overline{k^c}_9}(D^c_{i,j})$ and $d_{\overline{k^b}_9}(C^b_{l,j}) \oplus d_{\overline{k^d}_9}(D^d_{l,j})$ have an a difference after SB_9^{-1} in byte 4. Record $(\overline{k^a}_9)$ and discard all the quartets which do not satisfy this condition.

7. Guess the 32-bit subkey $k^{a'}_0$ of K^a_0 in the positions of bytes 0,5,10,15 and compute $k^{b'}_0 = k^{c'}_0 = k^{d'}_0 = k^{a'}_0$ *(ΔK^*_0 and $\Delta K'_0$ are zero in these four bytes)*.

   7.1. Partially encrypt each quartet $(P^a_{i,j}, P^b_{l,j}, O^c_{i,j}, O^d_{l,j})$ remaining after Step 6.2 under $k^{a'}_0, k^{b'}_0, k^{c'}_0, k^{d'}_0$, respectively.

7.2. Check if $e_{k_0^{a'}}(P^a_{i,j}) \oplus e_{k_0^{b'}}(P^b_{l,j})$ and $e_{\underline{k_0^{c'}}}(O^c_{i,j}) \oplus e_{k_0^{d'}}(O^d_{l,j})$ have an a difference in byte 0 after MC_1. Record ($\overline{k^a}_9, k_0^{a'}$) and discard all the quartets which do not satisfy this condition.

8. Guess the 32-bit subkey k_0^{a*} of K_0^a in the positions of bytes 3,4,9,14 and compute $k_0^{b*} = k_0^{a*}$, compute $k_0^{c*} = k_0^{a*} \oplus M_1$, with the 32-bit value $M_1 = (a,0,0,0)$ and compute $k_0^{d*} = k_0^{b*} \oplus M_1$.

   8.1. Partially encrypt each quartet $(P^a_{i,j}, P^b_{l,j}, O^c_{i,j}, O^d_{l,j})$ remaining after Step 7.2 under $k_0^{a*}, k_0^{b*}, k_0^{c*}, k_0^{d*}$, respectively.

   8.2. Check if $e_{k_0^{a*}}(P^a_{i,j}) \oplus e_{k_0^{b*}}(P^b_{l,j})$ and $e_{k_0^{c*}}(O^c_{i,j}) \oplus e_{k_0^{d*}}(O^d_{l,j})$ have an a difference in byte 4 after MC_1. If there are at least 2 quartets passing this test, record ($\overline{k^a}_9, k_0^{a'}, k_0^{a*}$) and all the remaining quartets and then go to Step 9. Otherwise, repeat Step 8 with another guessed key. If all the possible keys are tested, then repeat Step 7 with another guessed key. If all the possible keys are tested, then repeat Step 6 with another guessed key.

9. Output the subkeys ($\overline{k^a}_9, k_0^{a'}, k_0^{a*}$).

Analysis of the Attack

The amount of 2^{64} plaintexts can be combined in about $\left(2^{64}\right)^2 = 2^{128}$ quartets. We use 2^{38} structures of 2^{64} plaintexts to increase the amount of quartets to $2^{128+38} = 2^{166}$. We note that each structure can be analyzed separately. In the following we discuss the complexities for one structure.

The data complexity of Steps 1, 2, 3 and 4 is $2^2 \cdot 2^{64} = 2^{66}$ chosen plaintexts, while the time complexity is about $2^2 \cdot 2^{64} = 2^{66}$ encryptions for Steps 1 to 4. The memory complexity of Step 5 is $2^3 \cdot 2^{64} = 2^{67}$ plaintexts. Since we have a 64-bit filtering condition in Step 5 about $2^{128} \cdot 2^{-64} = 2^{64}$ wrong quartets remain after this step. Step 6.1 takes about

$$(1/9) \cdot (1/16) \cdot 2^8 \cdot 2^2 \cdot 2^{64} = 2^{66.83}$$

nine round encryptions. The number of remaining wrong quartets after Step 6.2 are $2^{64} \cdot 2^{-12} = 2^{52}$, since we have a 6-bit filtering on both pairs of a quartet. The time complexity of Step 7.1 is about

$$(1/9) \cdot (4/16) \cdot 2^{32} \cdot 2^8 \cdot 2^2 \cdot 2^{52} = 2^{88.83}$$

nine round encryptions. Due to the 32-bit filtering on both pairs we obtain about $2^{52} \cdot 2^{-64} = 2^{-12}$ wrong quartets after Step 7.2. The time complexity of Step 8.1 is negligible, while about $2^{-12} \cdot 2^{-64} = 2^{-76}$ quartets remain after this step.

Now we regard all the structures together. We obtain about $2^{38} \cdot 2^{-76} = 2^{-38}$ wrong quartets for all structures. A right quartet occurs with probability

$$\begin{aligned} Pr_c &= \Pr(\alpha \to \beta_{out}) \cdot (\Pr(\gamma \leftarrow \delta))^2 \cdot \Pr(\beta_{out} = \beta_{in}) \cdot \Pr(\alpha \leftarrow \beta_{in}) \\ &= 2^{-64} \cdot (2^{-6})^2 \cdot 2^{-56} \cdot 2^{-32} = 2^{-164}, \end{aligned}$$

since all related-key differential conditions are fulfilled. Thus, we expect to obtain about $2^{166} \cdot 2^{-164} = 2^2$ right quartets after Step 5 in total.

For the right key we count about $2^2 + 2^{-38}$ quartets and for some wrong key we expect a count of about 2^{-38} quartets.

Using the Poisson distribution we can compute the success rate of our attack. The probability that the counter for each wrong key is at least 2 assuming $Y_i \sim Poisson(\mu = 2^{-38})$ is

$$\Pr(Y \geq 2) = e^{-2^{-38}} \cdot \frac{(2^{-38})^2}{2!} \approx 2^{-77}.$$

For all the $2^{72} - 1$ wrong keys used in our analysis we expect about $2^{72} \cdot 2^{-77} = 2^{-5}$ wrong keys which have a count of at least 2 quartets. The probability that the right key has a count of at least 2 quartets using $Z \sim Poisson(\mu = 2^2)$ is

$$\Pr(Z \geq 2) \approx 0.91.$$

The memory complexity is about $2^{67} = 2^3 \cdot 2^{64}$ plaintexts and ciphertexts and the time complexity is about $2^{126.83} = 2^{38} \cdot 2^{88.83}$ 9-round AES-256 encryptions. The data complexity is $2^{38} \cdot 2^{64} \cdot 2^2 = 2^{104}$ adaptive chosen plaintexts and ciphertexts.

Chapter 4

Cryptanalysis of Block Ciphers inside Hash Functions

Block ciphers are well studied primitives which are often used as building blocks for hash functions as the base of their compression function. They are often used in modes of operations like *Davies-Meyer* [152], *Matyas-Meyer-Oseas*, or *Miyaguchi-Preneel* [111].

In this chapter we present attacks on block ciphers which are used as such building blocks. We propose an attack on a round reduced version of SHACAL-2 and an attack which breaks the entire internal block ciphers of Tiger as well as another one which breaks the HAS-160 in encryption mode. It is quite unknown if a weak internal block cipher leads also to a weak hash function, but nevertheless it shows the existance of flaws in some special attack scenarios.

4.1 A Related-Key Boomerang Attack on SHACAL-2

SHACAL-2 [75] is a 256-bit block cipher which is based on the compression function of the hash function standard SHA-256 [120].

SHACAL-2 has 64 rounds and supports key lengths from 128 up to 512 bits. The first cryptanalytic result on SHACAL-2 was an impossible differential attack [77] on a 30-round reduced version of SHACAL-2. A differential-nonlinear attack [140] and a square-nonlinear attack [140] were introduced which can attack up to 32 and 28 rounds, respectively. Introducing related keys leads to an improved attack of up to 37 rounds which was called the related-key differential-nonlinear attack [92]. The best cryptanalytic results on SHACAL-2 are the related-key rectangle attacks on 42 rounds

Table 4.1: Comparison of attacks on SHACAL-2

Attack	# Rounds	# Keys	Data	Time	Memory	Source
Square-Nonlinear	28	1	$463 \cdot 2^{32}$ CP	$2^{494.1}$	$2^{45.9}$	[140]
Impossible Differential	30	1	744 CP	$2^{495.1}$	$2^{14.5}$	[77]
Differential-Nonlinear	32	1	$2^{43.4}$ CP	$2^{504.2}$	$2^{48.4}$	[140]
RK Boomerang	34	2	2^{225} RK-ACPC	2^{225}	2^{10}	Sec. 4.1.2
RK Differential-Nonlinear	35	2	$2^{42.32}$ RK-CP	$2^{452.10}$	$2^{47.32}$	[92]
RK Rectangle	37	2	$2^{235.16}$ RK-CP	$2^{486.95}$	$2^{240.16}$	[92]
RK Rectangle	40	2	$2^{243.38}$ RK-CP	$2^{448.43}$	$2^{247.38}$	[102]
RK Rectangle	42	2	$2^{243.38}$ RK-CP	$2^{488.37}$	$2^{247.38}$	[102]
RK Rectangle†	43	2	$2^{240.38}$ RK-CP	$2^{480.4}$	$2^{245.38}$	[146]
RK Rectangle	44	2	2^{233} RK-CP	$2^{497.2}$	2^{238}	[101]

RK: Related-Key, CP: Chosen Plaintexts, ACPC: Adaptive Chosen Plaintexts and Ciphertexts, Memory in bytes

† : The attack has a flaw pointed out in [101]. The table shows the corrected values.

by Lu et al. [102], a 43-round attack by Wang [146] and a 44-round attack by Lu et al. [101]. The disadvantage of these attacks is the huge memory requirements, which we address in this thesis. We present the first attack on SHACAL-2 which can break up to 34 rounds using two related keys with very little memory. Table 4.1 summarizes the results known from the literature and our results on SHACAL-2.

4.1.1 Description of SHACAL-2

For keys smaller than 512 bits, zeros are padded until the key length reaches 512 bits. A 256-bit plaintext is divided into eight 32-bit words:

$$P_0 = A_0||B_0||C_0||D_0||E_0||F_0||G_0||H_0$$

The corresponding ciphertext P_{64} is denoted by

$$P_{64} = A_{64}||B_{64}||C_{64}||D_{64}||E_{64}||F_{64}||G_{64}||H_{64}.$$

Round i can be described as follows:

$$\begin{aligned}
T^1_{i+1} &= K_i \boxplus \Sigma_1(E_i) \boxplus Ch(E_i, F_i, G_i) \boxplus H_i \boxplus W_i, \\
T^2_{i+1} &= \Sigma_0(A_i) \boxplus Maj(A_i, B_i, C_i), \\
H_{i+1} &= G_i, \\
G_{i+1} &= F_i, \\
F_{i+1} &= E_i,
\end{aligned}$$

$$
\begin{aligned}
E_{i+1} &= D_i \boxplus T^1_{i+1},\\
D_{i+1} &= C_i,\\
C_{i+1} &= B_i,\\
B_{i+1} &= A_i,\\
A_{i+1} &= T^1_{i+1} \boxplus T^2_{i+1},
\end{aligned}
$$

where K_i is the i-th subkey and W_i is the i-th round constant. The four functions in the encryption algorithm are defined as follows:

$$
\begin{aligned}
Ch(X,Y,Z) &= (X \wedge Y) \oplus (\neg X \wedge Z),\\
Maj(X,Y,Z) &= (X \wedge Y) \oplus (X \wedge Z) \oplus (Y \wedge Z),\\
\Sigma_0(X) &= (X^{\ggg 2}) \oplus (X^{\ggg 13}) \oplus (X^{\ggg 22}),\\
\Sigma_1(X) &= (X^{\ggg 6}) \oplus (X^{\ggg 11}) \oplus (X^{\ggg 25}).
\end{aligned}
$$

We do not describe the constants here since they are not relevant for our analysis. Figure 4.1 shows the round function of SHACAL-2.

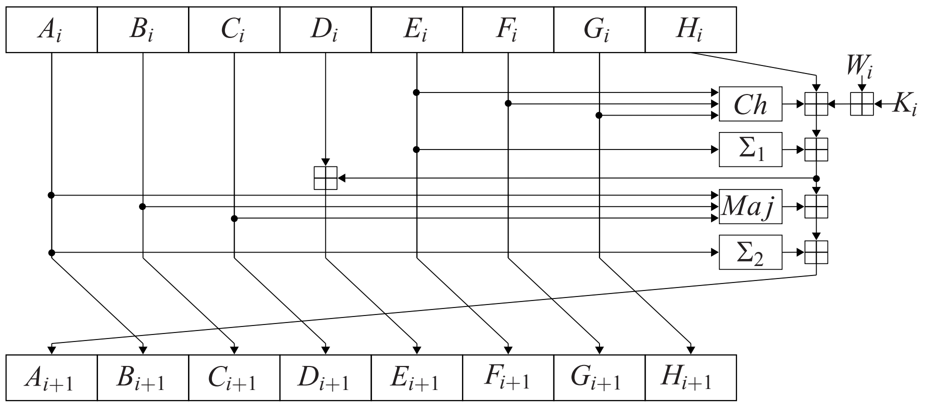


Figure 4.1: The round function of SHACAL-2

The key schedule algorithm of SHACAL-2 takes as an input a 512-bit key. As stated above, as many zeros as necessary are padded to obtain a full 512-bit key. The 512-bit key K is then divided into sixteen 32-bit words $K_0, K_1, \ldots, K_{15}$. These are the subkeys for the first 16 rounds. The i-th subkey ($16 \le i \le 63$) is computed as

$$K_i = \sigma_1(K_{i-2}) + K_{i-7} + \sigma_0(K_{i-15}) + K_{i-16},$$

where

$$
\begin{aligned}
\sigma_0(X) &= (X^{\ggg 7}) \oplus (X^{\ggg 18}) \oplus (X^{\gg 3}),\\
\sigma_1(X) &= (X^{\ggg 17}) \oplus (X^{\ggg 19}) \oplus (X^{\gg 10}).
\end{aligned}
$$

In the following we present some basic properties of the $Ch(\cdot)$ and $Maj(\cdot)$ functions which are needed in our attack.

Proposition 1 *(from [146]) The nonlinear function $Ch(X,Y,Z) = (X \wedge Y) \oplus (\neg X \wedge Z)$, possesses the following properties:*

1. *$Ch(x,y,z) = Ch(\neg x,y,z)$ if and only if $y = z$.*

   $Ch(0,y,z) = 0$ and $Ch(1,y,z) = 1$ if and only if $y = 1$ and $z = 0$.

   $Ch(0,y,z) = 1$ and $CH(1,y,z) = 0$ if and only if $y = 0$ and $z = 1$.

2. *$Ch(x,y,z) = Ch(x,\neg y,z)$ if and only if $x = 0$.*

   $Ch(x,0,z) = 0$ and $Ch(x,1,z) = 1$ if and only if $x = 1$.

3. *$Ch(x,y,z) = Ch(x,y,\neg z)$ if and only if $x = 1$.*

   $Ch(x,y,0) = 0$ and $Ch(x,y,1) = 1$ if and only if $x = 0$.

Proposition 2 *(from [146]) The nonlinear function*
$Maj(X,Y,Z) = (X \wedge Y) \oplus (X \wedge Z) \oplus (Y \wedge Z)$, possesses the following properties:

1. *$Maj(x,y,z) = Maj(\neg x,y,z)$ if and only if $y = z$.*

   $Maj(0,y,z) = 0$ and $Maj(1,y,z) = 1$ if and only if $y = \neg z$.

2. *$Maj(x,y,z) = Maj(x,\neg y,z)$ if and only if $x = z$.*

   $Maj(x,0,z) = 0$ and $Maj(x,1,z) = 1$ if and only if $x = \neg z$.

3. *$Maj(x,y,z) = Maj(x,y,\neg z)$ if and only if $x = y$.*

   $Maj(x,y,0) = 0$ and $Maj(x,y,1) = 1$ if and only if $x = \neg y$.

4.1.2 Memoryless Related-Key Boomerang Attack on 34-Round SHACAL-2

In this section we propose a 34-round almost memoryless related-key boomerang attack on SHACAL-2 using two related keys. The cipher E is treated as a cascade of two subciphers $E(P) = E1(E0(P))$, $E0$ is a subcipher containing rounds 1 to 25 while the subcipher $E1$ covers rounds 25 to 34. Our related-key differentials are based on the differentials used in [146]. We use these differentials to build a 34-round related-key boomerang distinguisher, which can be used in a memoryless attack to break 34 rounds of SHACAL-2. The main advantage of our attack is that we do not need to store all

Table 4.2: The fixed plaintext bits for our attack on SHACAL-2

$A^1_{0,i}$	$B^1_{0,i}$	$E^1_{0,i}$	$F^1_{0,i}$
$a^1_{31} = b^1_{31}$, $a^1_k = c^1_k$ $(k = 6,9,18,20,25,29)$	$b^1_k = \neg f^1_k$ $(k = 19,30)$ $b^1_9 = e^1_9$	$e^1_{31} = 0$, $e^1_k = 0$ $(k = 18,29)$	$f^1_k = g^1_k$ $(k = 9,13,19)$

a^1_k, b^1_k, c^1_k, e^1_k, f^1_k, g^1_k are the k-th bits of $A^1_{0,i}, B^1_{0,i}, C^1_{0,i}, E^1_{0,i}, F^1_{0,i}, G^1_{0,i}$

the quartets as in the related-key rectangle attack of [92, 101, 102, 146]. Thus, we only require a very small amount of memory to successfully mount our attack.

The notations used in the attack are as follows:

- K, K^* keys (512-bit).
- K_i, K^*_i subkeys of round i, where $i \in \{0,1,2,\dots,38\}$ (32-bit).
- ΔK is the key difference, $\Delta K = (e_{31},0,0,0,0,0,0,0,0,e_{31},0,0,0,0,0,0)$.
- ΔK_i is the i-th subkey difference derived from ΔK.

In the following we describe the related-key differentials used in our attack.

The Related-Key Differential for $E0$ and for $E0^{-1}$

The related-key differential for $E0$ covers rounds 1 to 25, and is

$$(e_{6,9,18,20,25,29}, e_{31}, 0, 0, e_{6,20,25}, e_{31}, 0, 0) \to (0, 0, e_M, e_{31}, 0, e_{9,13,19}, e_{18,29}, e_{31}).$$

The key schedule of SHACAL-2 has a low difference propagation in the first several rounds. Thus, if two keys K and K^* are different in only two words in the first 16 rounds, namely $\Delta K_0 = e_{31}$ and $\Delta K_9 = e_{31}$, we can introduce a zero subkey difference from round 10 to 23 as shown in Table 4.2. Wang [146] improves the 25-round differential characteristic introduced by Lu et al. [102] such that one does not have to guess the first subkeys K_0 and K^*_0. This can be done by using Proposition 1 and Proposition 2 and fixing 16-bit conditions in the plaintexts to obtain a probability of 1 for the first round of the related-key differential. The bit conditions are presented in Table 4.2. The related-key differential for $E0$ is shown in Table 4.3. Since we do not deal with truncated differentials, the probability for a differential remains the same if it runs in backward direction. Thus, $\Pr(\alpha \to \beta) = 2^{-47}$ holds for the related-key differential for $E0$. As explained in [102], many possible differences ΔK_{24} are caused due to the addition modulo 2^{32} operations in the key schedule. The addition of the round keys in round 24 are taken with the output of round 23. Due to the zero difference in the output

Table 4.3: The Related-Key Differential for $E0$

i	ΔA_i	ΔB_i	ΔC_i	ΔD_i	ΔE_i	ΔF_i	ΔG_i	ΔH_i	ΔK_i	Prob.
0	0	e_M	e_{31}	0	$e_{9,13,19}$	$e_{18,29}$	e_{31}	Δ_H	e_{31}	1
1	0	0	e_M	e_{31}	0	$e_{9,13,19}$	$e_{18,29}$	0	0	2^{-11}
2	e_{31}	0	0	e_M	0	0	$e_{9,13,19}$	$e_{18,29}$	0	2^{-10}
3	0	e_{31}	0	0	$e_{6,20,29}$	0	0	$e_{9,13,19}$	0	2^{-7}
4	0	0	e_{31}	0	0	$e_{6,20,25}$	0	0	0	2^{-4}
5	0	0	0	e_{31}	0	0	$e_{6,20,25}$	0	0	2^{-3}
6	0	0	0	0	e_{31}	0	0	$e_{6,20,25}$	0	2^{-4}
7	0	0	0	0	0	e_{31}	0	0	0	2^{-1}
8	0	0	0	0	0	0	e_{31}	0	0	2^{-1}
9	0	0	0	0	0	0	0	e_{31}	e_{31}	1
10	0	0	0	0	0	0	0	0	0	1
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
23	0	0	0	0	0	0	0	0	0	1
24	0	0	0	0	0	0	0	0	$\cdot$	2^{-6}
25	$e_{13,24,28}$	0	0	0	$e_{13,24,28}$	0	0	0	$\cdot$	$\cdot$

$$M = \{6,9,18,20,25,29\}, \Sigma_1(E_0 \oplus e_{9,13,19}) - \Sigma_1(E_0) + \Delta_H = 0$$

of round 23, we can count over the possibilities for all the additions together when we compute the probability for $E0$. The subkey differences ΔK_{24} and ΔK_{25} are marked with $\cdot$, since we do not know the exact values.

The related-key differential for $E0^{-1}$ is very similar to the related-key differential for $E0$ presented in Table 4.3. It starts from the bottom of the table (round 25) and goes up to the top (round 1). Since we cannot fix some bits before the first round, the probability going from round two to one in backward direction is less than one. Thus, the probability in the backward direction decreases by 2^{-18}. This gives a total probability of $\Pr(\alpha \leftarrow \beta) = 2^{-47} \cdot 2^{-18} = 2^{65}$.

The Differential for $E1^{-1}$

Our differential for $E1^{-1}$ covers rounds 34 to 25, which can be written as

$$(e_{31}, 0, 0, 0, e_{31}, 0, 0, 0) \rightarrow (e_{31}, e_{31}, e_{M'}, 0, 0, e_{9,13,19}, e_{18,29,31}, 0).$$

Note that this differential does not uses related keys. The differential for $E1^{-1}$ occurs with probability $\Pr(\gamma \leftarrow \delta) = 2^{-54}$ and can be found in Table 4.4. We emphasize that it does not use any key differences.

Table 4.4: The Differential for $E1^{-1}$

i	ΔA_i	ΔB_i	ΔC_i	ΔD_i	ΔE_i	ΔF_i	ΔG_i	ΔH_i	Prob.
34	e_{31}	0	0	0	e_{31}	0	0	0	1
33	0	0	0	0	0	0	0	e_{31}	2^{-1}
32	0	0	0	0	0	0	e_{31}	e_{31}	1
31	0	0	0	0	0	e_{31}	e_{31}	e_{31}	2^{-4}
30	0	0	0	0	e_{31}	e_{31}	e_{31}	$e_{6,20,25}$	2^{-7}
29	0	0	0	e_{31}	e_{31}	e_{31}	$e_{6,20,25}$	0	2^{-8}
28	0	0	e_{31}	e_{31}	e_{31}	$e_{6,20,25}$	0	0	2^{-7}
27	0	e_{31}	e_{31}	e_{31}	$e_{6,20,25}$	0	0	$e_{9,13,19}$	2^{-12}
26	e_{31}	e_{31}	e_{31}	$e_{M'}$	0	0	$e_{9,13,19}$	$e_{18,29,31}$	2^{-15}
25	e_{31}	e_{31}	$e_{M'}$	0	0	$e_{9,13,19}$	$e_{18,29,31}$	0	–

$M' = \{6,9,18,20,25,29,31\}$

The Attack

The attack works as follows:

- For $i = 1,2,\dots,2^{223}$ do

  1. Choose a plaintext $P^a_{0,i}$ and a difference ΔH_i satisfying the conditions presented in Table 4.2, where $\Sigma_1(E_0 \oplus e_{9,13,19}) - \Sigma_1(E_0) + \Delta H_0 = 0$ is satisfied. Ask for the encryption of $P^a_{0,i}$ under K to obtain the plaintext $P^a_{35,i}$.
  2. Use the previously chosen $P^a_{0,i}$ and the difference $\alpha_i = (0, e_M, e_{31}, 0, e_{9,13,19}, e_{18,29}, e_{31}, \Delta H_i)$, $M = \{6,9,18,20,25,29\}$, to compute $P^b_{0,i} = P^a_{0,i} \oplus \alpha_i$. Ask for the encryption of $P^b_{0,i}$ under K^* to obtain $P^b_{35,i}$.
  3. Compute the ciphertexts $P^c_{35,i} = P^a_{35,i} \oplus \delta$, where $\delta = (e_{6,9,18,20,25,29}, e_{31}, 0, 0, e_{6,20,25}, e_{31}, 0, 0)$. Ask for the decryption of $P^c_{35,i}$ under K to obtain $P^c_{0,i}$.
  4. Compute the ciphertexts $P^d_{35,i} = P^b_{35,i} \oplus \delta$. Ask for the decryption of $P^d_{35,i}$ under K^* to obtain $P^d_{0,i}$.
  5. Check if $P^c_{0,i} \oplus P^d_{0,i} = \alpha_i$. Discard all quartets $(P^a_{0,i}, P^b_{0,i}, P^c_{0,i}, P^d_{0,i})$ which do not satisfy this condition and store the remaining in ϕ.

- For all 32-bit candidate subkeys K_0 do

  6. Set a counter to zero.

  - For all quartets $(P^a_{0,i}, P^b_{0,i}, P^c_{0,i}, P^d_{0,i})$ in ϕ

    7. Encrypt $P^c_{0,i}, P^d_{0,i}$ one round under $K^*_0 = K_0 \oplus e_{31}$, respectively.

8. Check if $P^c_{1,i} \oplus P^d_{1,i} = \tau$, where $\tau = (0, 0, e_M, e_{31}, 0, e_{9,13,19}, e_{18,29}, 0)$. If so, increase the counter for the used subkey by one.
9. Take the guess of K_0 with the highest counter as the correct subkey pair.

10. Output the subkeys K_0 and $K_0^* = K_0 \oplus e_{31}$.

Analysis of the Attack

A right quartet occurs with probability

$$\begin{aligned} Pr_c &= \Pr(\alpha \to \beta) \cdot (\Pr(\gamma \leftarrow \delta))^2 \cdot \Pr(\alpha \leftarrow \beta) \\ &= 2^{-47} \cdot (2^{-54})^2 \cdot 2^{-65} = 2^{-220}, \end{aligned}$$

when all related-key differential conditions are fulfilled. We expect to obtain about $2^{223} \cdot 2^{-220} = 2^3$ right quartets and about $2^{223} \cdot 2^{-256} = 2^{-33}$ wrong quartets stored in ϕ. Note that a random difference is equal to α_i with probability 2^{-256}. We need to store only 2^3 quartets, since each quartet can be computed independently of the others, and discarded if needed. This is equivalent to $2^2 \cdot 2^3 = 2^5$ plaintexts which are about 2^{10} bytes of memory.

The data and time complexity of Steps 1 to 4 is about $2^2 \cdot 2^{223} = 2^{225}$ adaptive chosen plaintexts and ciphertexts. Since we expect to have about $2^3 + 2^{-33} \approx 2^3$ quartets stored in ϕ. The time and data complexity of Steps 6 to 10 is negligible compared to the previous steps.

We expect to have about $2^3 + 2^{-33} \approx 2^3$ quartets counted for the right key in Step 8. For each wrong key we expect to have about 2^{-33} quartets.

Using the Poisson distribution we can compute the success rate of our attack. The probability that the counter of a wrong key is at least 2 using $Y_i \sim Poisson(\mu = 2^{-33})$ is

$$\Pr(Y \geq 3) = e^{-2^{-33}} \cdot \frac{(2^{-33})^3}{3!} \approx 2^{-101}.$$

For all the $2^{32} - 1$ wrong keys used in our analysis we expect about $2^{32} \cdot 2^{-101} = 2^{-69}$ wrong keys which have a count of at least 2 quartets. The probability that the right key has a count of at least 2 quartets using $Z \sim Poisson(\mu = 2^3)$ is

$$\Pr(Z \geq 3) \approx 0.95.$$

We can increase the success probability of our attack by increasing the initial number of quartets, which will also increase the data, time and memory complexity of our attack. The data and time complexity of our attack is determined by Steps 1 to 4 which is about 2^{244} adaptive chosen plaintexts and ciphertexts. This attack needs only 2^{10} bytes of memory.

4.2 A Related-Key Boomerang Attack on Tiger Block Cipher

Tiger [1] is a 192-bit MERKLE-DAMGÅRD [41, 113] hash function, which operates on 512-bit message blocks.

Table 4.5 summarizes the results known from the literature on the Tiger hash function. The best cryptanalytic result on the Tiger block cipher is a 22-round related-key boomerang and rectangle attack [45]. See Table 4.6 for a comparison of both attacks and our new attack on full round Tiger block cipher. We present the first attack that can break the full 24-round Tiger block cipher, using a related-key boomerang attack which has a data complexity of 2^{161} adaptive chosen plaintexts and ciphertexts. The time complexity is about 2^{161} 24-round Tiger encryptions.

Table 4.5: Comparison of attacks on Tiger

Attack	# **Rounds**	**Time**	**Source**
Collision	17	2^{49}	[87]
Collision	19	2^{62}	[106]
Near-Collision	20	2^{49}	[87]
Pseudo-Near-Collision	22	2^{44}	[106]
Pseudo-Near-Collision	24	2^{47}	[110]
Pseudo-Collision	24	2^{47}	[110]
2nd-Pre-Image	12	$2^{63.5}$	[79]
2nd-Pre-Image	16	2^{161}	[80]
2nd-Pre-Image	23	$2^{187.5}$	[148]
Pre-Image	12	$2^{64.5}$	[79]
Pre-Image	13	$2^{128.5}$	[79]
Pre-Image	16	2^{161}	[80]
Pre-Image	23	1.4×2^{189}	[148]
Pre-Image	24	$2^{184.3}$	[73]

Table 4.6: Comparison of attacks on Tiger Block Cipher

Attack	# **Rounds**	**Data**	**Time**	**Source**
Related-Key Rectangle	22	2^{97}	$2^{154.3}$	[45]
Related-Key Boomerang	22	2^{16}	$2^{14.25}$	[45]
Related-Key Boomerang	24	2^{161}	2^{161}	Sec. 4.2.2

4.2.1 Description of Tiger's Block Cipher

Tiger's compression function is based on applying an internal "block cipher like" function, which takes a 192-bit "plaintext" and a 512-bit key to compute a 192-bit "ciphertext". The "block cipher like" function is applied according to the Davies-Meyer construction: a 512-bit message block is basically used as a key to encrypt the 192-bit chaining value, and then the input chaining value is fed forward to make the whole function non-invertible. In the remainder of this section, we describe Tiger in sufficient detail to follow the course of our attack; for a more detailed description of the hash function and its design rationale, the reader is referred to [1].

Tiger has been designed with 64-bit architectures in mind. Accordingly, in this section we denote a 64-bit unsigned integer as a "word". We represent a word as a hexadecimal number.

Furthermore, it uses the bit-wise NOT function, e.g., for $X = \texttt{0xEEEE AAAA 6666 0000}$, the negation $\overline{X}$ of X is $\overline{X} = \texttt{0x1111 5555 9999 FFFF}$.

The Tiger Round Function

In the terminology of [138], Tiger's block cipher is a target-heavy unbalanced Feistel cipher. The block is broken into three words, labeled A, B, and C. A plaintext is denoted by the triple (A_{-1}, B_{-1}, C_{-1}) and a ciphertext is denoted by (A_{23}, B_{23}, C_{23}), respectively. Each round, a subkey X, derived from the key schedule below, is XORed into C:

$$C \quad := \quad C \oplus X.$$

Then A and B are modified:

$$\begin{aligned} A &:= A \boxminus \mathbf{even}(C), \\ B &:= B \boxplus \mathbf{odd}(C), \\ B &:= B \boxtimes (\text{const}), \end{aligned}$$

with a round-dependent constant (const) $\in \{5, 7, 9\}$. The results are then shifted around, so that A, B, C becomes B, C, A. See Figure 4.2 for a description of the round function of Tiger. For the definition of **even** and **odd**, consider the word C being split into eight bytes $C_7, \ldots, C_0$, with C_7 as the most significant byte. The functions **even** and **odd** employ four S-boxes $T_1, \ldots, T_4 : \{0,1\}^8 \to \{0,1\}^{64}$ as follows:

$$\mathbf{even}(C) := T_1[C_0] \oplus T_2[C_2] \oplus T_3[C_4] \oplus T_4[C_6],$$

$$\mathbf{odd}(C) := T_1[C_7] \oplus T_2[C_5] \oplus T_3[C_3] \oplus T_4[C_1].$$

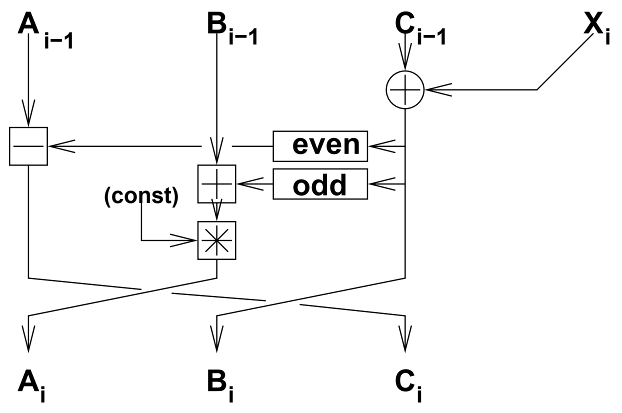


Figure 4.2: The round function of Tiger

We refer to C_0, C_2, C_4, and C_6 as the "even bytes of C". In our attack we also use the **odd** function on differences. For example, for a difference $\Delta C = C \oplus C'$ we obtain

$$\mathbf{odd}(\Delta C) = (T_1[C_7] \oplus T_2[C_5] \oplus T_3[C_3] \oplus T_4[C_1]) \oplus (T_1[C_7'] \oplus T_2[C_5'] \oplus T_3[C_3'] \oplus T_4[C_1']).$$

The round function spreads changes around very quickly. A one-bit difference introduced into C in the first round changes about half of the bits of the block by the end of the third round.

The Key Schedule

Tiger consists of 24 rounds. The rounds are labeled as $0,1,2,\ldots,22,23$. Each round uses one sukbey word X_i as its subkey. The first eight subkeys $X_0,\ldots,X_7$ are identical to the 512-bit key (or rather, to the 512-bit message block). The remaining 16 subkeys are generated by applying the key schedule function:

$$\begin{aligned}(X_8,\ldots,X_{15}) &:= \text{KeySchedule}(X_0,\ldots,X_7)\\ (X_{16},\ldots,X_{23}) &:= \text{KeySchedule}(X_8,\ldots,X_{15})\end{aligned}$$

The key schedule modifies its input $(x_0,\ldots,x_7)$ in two passes:

	first pass		second pass
1.	$x_0 := x_0 \boxminus (x_7 \oplus \text{Const}_1)$	9.	$x_0 := x_0 \boxplus x_7$
2.	$x_1 := x_1 \oplus x_0$	10.	$x_1 := x_1 \boxminus (x_0 \oplus (\overline{x_7}^{\ll 19}))$
3.	$x_2 := x_2 \boxplus x_1$	11.	$x_2 := x_2 \oplus x_1$
4.	$x_3 := x_3 \boxminus (x_2 \oplus (\overline{x_1}^{\ll 19}))$	12.	$x_3 := x_3 \boxplus x_2$
5.	$x_4 := x_4 \oplus x_3$	13.	$x_4 := x_4 \boxminus (x_3 \oplus \overline{x_2}^{\gg 23}))$
6.	$x_5 := x_5 \boxplus x_4$	14.	$x_5 := x_5 \oplus x_4$
7.	$x_6 := x_6 \boxminus (x_5 \oplus (\overline{x_4}^{\gg 23}))$	15.	$x_6 := x_6 \boxplus x_5$
8.	$x_7 := x_7 \oplus x_6$	16.	$x_7 := x_7 \boxminus (x_6 \oplus \text{Const}_2)$

The final values $(x_0,\ldots,x_7)$ are used as the key schedule output. The constants written in Heaxadecimal are $\mathrm{Const}_1 = \texttt{0xA5A5}\ldots\texttt{A5A5}$ and $\mathrm{Const}_2 = \texttt{0x0123}\ldots\texttt{CDEF}$.

4.2.2 A Memoryless Related-Key Boomerang Attack on the full Tiger's Block Cipher

In this section, we propose a 22-round related-key boomerang distinguisher, which is used for our memoryless related-key boomerang attack on the full 24-round Tiger block cipher. We make an extensive use of the following property in our attack.

Property 1 *Switching between an additive and an XOR difference holds with some probablitiy. e.g., if* $X - Y = 2^i \bmod 2^{64}$*, then* $\Pr[X \oplus Y = 2^i] = 2^{-1}$*. We have* $\Pr[X \oplus Y = 2^{63}] = 1$*, i.e., switching between the additive difference* $\mathbf{I} = 2^{63}$ *and the XOR-difference* $\mathbf{I}$ *is for free.*

A 22-Round Related-Key Boomerang Distinguisher

Let K be a key which can be written as $K = x_0, x_1, \ldots, x_7$, where x_i is a 64-bit word. We use four different but related keys K^a, K^b, K^c and K^d to mount our related-key rectangle attack on the full Tiger encryption mode. The key differences are as follows:

$$\begin{aligned}\Delta K^* &= K^a \oplus K^b = K^c \oplus K^d = (I,I,0,0,0,I,0,0),\\ \Delta K' &= K^a \oplus K^c = K^b \oplus K^d = (0,0,0,I,0,0,0,I)\end{aligned}$$

Since the key schedule of Tiger offers a high degree of linearity we can determine most of the subkey differences derived from the key differences ΔK^* and $\Delta K'$, respectively. Using the above key schedule we can derive the subkey differences from the key differences ΔK^* and $\Delta K'$. The subkey differences for $E0$ propagate as:[1]

$$(I,I,0,0,0,I,0,0) \longrightarrow (0,0,0,0,0,I,0,I) \longrightarrow (I,0,I,0,?,?,?,?)$$

The ? indicates an unknown value of a key difference. We obtain the subkey differences for $E1$ as:[2]

$$(0,0,0,I,0,0,0,I) \longrightarrow (0,I,0,0,0,0,0,I) \longrightarrow (0,0,0,0,0,0,0,I)$$

For our attack we use an 11-round related-key differential from round 2 to 12 for $E0$ $(\alpha \to \beta)$ using the key difference ΔK^*. The related-key differential is $\alpha \to \beta$, where

[1]This related keys were also used in the attack on the Tiger encryption mode from [45].

[2]The same key differential was used by pseudo-collision attack of Mendel and Rijmen [110].

$\alpha = (0, 0 \boxminus \Delta S, I)$ and $\beta = (0,0,0)$, where $\Delta S = odd(I)$ is chosen to be

$$\begin{aligned} \Delta S &= \texttt{T1(0x00)} \oplus \texttt{T1(0x80)} \\ &= \texttt{0x02AA B17C F7E9 0C5E} \oplus \texttt{0x0E57 15A2 D149 AA23} \\ &= \texttt{0x0CFD A4DE 26A0 A67D}. \end{aligned}$$

Note that the attack also works with different values ΔS than the one we give here. The differential for $E0^{-1}$ $(\beta \to \alpha)$ is the differential for $E0$ in the reverse direction. The related-key differential for $E0$ and for $E0^{-1}$ are shown in Table 4.7. Note that the table shows in each row the differences after round i. For example, the difference $(\Delta A_1, \Delta B_1, \Delta C_1) = (0, 0 \boxminus \Delta S, I)$ enters round 2 with the subkey difference $\Delta k_2 = 0$. The output differences of round 2 are $(\Delta A_2, \Delta B_2, \Delta C_2) = (0, I, 0)$, which happens with probability 2^{-7}.

Table 4.7: The Related-Key Differentials for $E0$ and for $E0^{-1}$ in the reverse order

Round(i)	ΔA_i	ΔB_i	ΔC_i	Δk_i	Prob.
1	0	$0 \boxminus \Delta S$	I	–	2^{-7}
2	0	I	0	0	1
3	I	0	0	0	1
4	0	0	I	0	1
5	0	0	0	I	1
6	0	0	0	0	1
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
12	0	0	0	0	–

We exploit another 11 rounds related-key differential for $E1^{-1}$ $(\delta \to \gamma)$ that covers rounds 23 to 12 using the key difference $\Delta K'$. The related-key differential is $\delta \to \gamma$, where $\delta = (0, I, 0)$ and $\gamma = (\Delta S, I, 0)$. The related-key differential for $E1^{-1}$ is shown in Table 4.8. Note that the table shows in each row the differences before round i.

The difference $(\Delta A_{23}, \Delta B_{23}, \Delta C_{23}) = (\Delta S, 1, 0)$ enters round 23 with the subkey difference $\Delta k_{23} = I$. The output in backward direction of this round is $(\Delta A_{22}, \Delta B_{22}, \Delta C_{22}) = (0,0,0)$, which happens with probability 2^{-7}. The resulting differences $(\Delta A_{22}, \Delta B_{22}, \Delta C_{22})$ are now the input differences of round 22. Our 22-round related-key boomerang distinguisher holds with probability 2^{-28}, since $p = 2^{-7}$ and $q = 2^{-7}$ which leads to $(p \cdot q)^2 = 2^{-28}$.

The Attack

The attack works as follows:

1. Guess the two 64-bit subkeys k_0^a, k_1^a and compute k_0^i, k_1^i, $i \in \{b, c, d\}$ using the known subkey differences ΔK^* and $\Delta K'$.

Table 4.8: The Related-Key Differential for $E1^{-1}$

Round(i)	ΔA_i	ΔB_i	ΔC_i	Δk_i	Prob.
23	ΔS	I	0	I	2^{-7}
22	0	0	0	0	1
21	0	0	0	0	1
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
16	0	0	0	0	1
15	0	0	0	I	1
14	0	0	I	0	1
13	I	0	0	0	1
12	0	I	0	–	–

- For $i = 1,2,\ldots,2^{31}$ do
  2. Choose a plaintext $P^a_{1,i}$ uniformly at random and compute $P^b_{1,i} = P^a_{1,i} \oplus \alpha$, where $\alpha = (0, 0 \boxminus \Delta S, I)$. Decrypt $P^a_{1,i}$ and $P^b_{1,i}$ under k^a_1, k^a_0 and k^b_1, k^b_0, respectively, and obtain the plaintexts $P^a_{-1,i}$ and $P^b_{-1,i}$. Ask for the encryption of the plaintexts $P^a_{-1,i}$ and $P^b_{-1,i}$ under K^a and K^b, respectively, to obtain the ciphertexts $P^a_{23,i}$ and $P^b_{23,i}$.
  3. Compute the ciphertexts $P^c_{23,i} = P^a_{23,i} \oplus \delta$ and $P^d_{23,i} = P^b_{23,i} \oplus \delta$, for $\delta = (\Delta S, I, 0)$. Ask for the decryption of the ciphertexts $P^c_{23,i}, P^d_{23,i}$ under K^c and K^d, respectively, and obtain the plaintexts $P^c_{-1,i}$ and $P^d_{-1,i}$.
  4. Partially encrypt $P^c_{-1,i}, P^d_{-1,i}$ under k^c_0, k^c_1 and k^d_0, k^d_1, respectively, and obtain $P^c_{1,i}$ and $P^d_{1,i}$. Check if $P^c_{1,i} \oplus P^d_{1,i} = \alpha$. If true, store the quartet $(P^a_{1,i}, P^b_{1,i}, P^c_{1,i}, P^d_{1,i})$ in ϕ.

5. Output the candidate keys k^i_0, k^i_1 $(i \in \{a,b,c,d\})$ with the highest counter.

Analysis of the Attack

From $\#Q = 2^{31}$ quartets we expect about $\#C = 2^{31} \cdot 2^{-28} = 2^3$ right boomerang quartets in ϕ. A difference $P^c_{1,i} \oplus P^d_{1,i}$ is equal to α with probability $Pr_f = 2^{-192}$. Thus, we expect $\#F = \#Q \cdot Pr_f = 2^{31} \cdot 2^{-192} = 2^{-161}$ wrong boomerang quartets which pass the test in Step 4.

The data complexity for each candidate key pair k^a_0, k^a_1 of Step 1 to 4 is about $2^2 \cdot 2^{31} = 2^{33}$ adaptive chosen plaintext and ciphertexts. We expect to have only 2^3 quartets stored in ϕ that satisfy the condition in Step 4, hence, we do not have to store 2^{33} as one might expect.

Step 1 is applied 2^{128} times for each of the 2^{128} keys. The time complexity of Step 2 is determined by two 24-round Tiger encryptions. Steps 3 and 4 have negligible time complexity. Thus, the overall time complexity is bounded by $2^{128} \cdot 2^{31} \cdot 2^2 = 2^{161}$.

A wrong quartet is filtered in Step 4 with probability 2^{-192}. Thus, about $2^{161} \cdot 2^{-192} = 2^{-31}$ wrong quartets are expected for each key candidate. Furthermore, we expect about $2^3 + 2^{161} \cdot 2^{-192} \approx 2^3$ quartets are encountered after this step for the right key.

Using the Poisson distribution we can compute the success rate of our attack. The probability that the counter of a wrong key is at least 5 assuming $Y_i \sim Poisson(\mu = 2^{-31})$ is

$$\Pr(Y \geq 5) = e^{-2^{-31}} \cdot \frac{(2^{-31})^5}{5!} \approx 2^{-161}.$$

For all the $2^{128} - 1$ wrong keys used in our analysis we expect about $2^{128} \cdot 2^{-161} = 2^{-33}$ wrong keys to have at least 5 quartets. The probability that the right key has a count of at least 5 quartets using $Z \sim Poisson(\mu = 2^3)$ is

$$\Pr(Z \geq 5) \approx 0.9.$$

The data complexity of our attack is determined by Steps 1 to 4 which is $2^{33+128} = 2^{161}$ adaptive chosen plaintexts and ciphertexts, while the time complexity is about 2^{161} 24-round Tiger encryptions. Each quartet can be analyzed separately and thus we only need to store 2^5 plaintexts and ciphertext pairs.

4.3 A Related-Key Rectangle Attack on the HAS-160 Encryption Mode

HAS-160 is a hash function widely used by the Korean industry, following its standardization by the Korean government (TTAS.KO-12.0011/R1) [143]. Based on the MERKLE-DAMGÅRD structure [41, 113], it uses a compression function with message block of 512 bits and a chaining value of 160 bits. HAS-160 consists of a round function which is applied 80 times for each input message block. The overall design of the compression function is similar to the design of SHA-1 [119] and the MD family [132, 131], except some modifications in the rotation constants and in the key schedule.

Up until now there were only a few cryptographic results on HAS-160. Yun et al. [155] found a collision on 45-round HAS-160 with complexity 2^{12} by using the techniques introduced by Wang et al. [149]. Cho et al. extended the previous result to break 53-round HAS-160 in time 2^{55} [33]. Mendel and Rijmen shown the attack complexity in [33] to 2^{35} hash computations and they were able to present a colliding message pair for the 53-round version of HAS-160 [109]. They also shown how the attack can be extended to 59-round HAS-160 with a complexity of 2^{55}.

HAS-160 in encryption mode is resistant to most of the attacks that can be applied to SHACAL-1, e.g., since it offers different rotation constants in each round and its key schedule does not offer any sliding properties. Nevertheless, the linearity of the key schedule, makes it vulnerable to related-key attacks.

In this section we analyze the internal block cipher of HAS-160 and present the first cryptographic result on 77-round HAS-160 used as a block cipher, which we call encryption mode. Using a related-key rectangle attack with four related keys we can break 77-round HAS-160, i.e., recovering some key bits faster than exhaustive search.

4.3.1 Description of the HAS-160 Encryption Mode

Now, we briefly describe the structure of HAS-160 and how it can be used as a block cipher. The inner block cipher operates on a 160-bit plaintext as the message block and a 512-bit key. A 160-bit plaintext, P_0, is divided into five 32-bit words, i.e., $P_0 = A_0||B_0||C_0||D_0||E_0$. HAS-160 consists of 4 passes of 20 rounds each, and the round function is applied 80 times in total. The corresponding ciphertext, P_{80}, is denoted by $A_{80}||B_{80}||C_{80}||D_{80}||E_{80}$. The round function at round i $(i = 1, \ldots, 80)$ can be described as follows:

$$\begin{aligned} A_i &\leftarrow A_{i-1}^{\lll s_{1,i}} \boxplus f_i(B_{i-1}, C_{i-1}, D_{i-1}) \boxplus E_{i-1} \boxplus k_i \boxplus c_i, \\ B_i &\leftarrow A_{i-1}, \\ C_i &\leftarrow B_{i-1}^{\lll s_{2,i}}, \\ D_i &\leftarrow C_{i-1}, \\ E_i &\leftarrow D_{i-1}, \end{aligned}$$

where c_i and k_i represents the i-th round constant and the i-th round subkey, respectively, while $f_i(\cdot)$ represents a boolean function. Figure 4.3 shows the round function of HAS-160. The function $f_i(\cdot)$ and the constants c_i of round i can be found in Table 4.9. The rotation constant $s_{1,i}$ used in round i are given in Table 4.10.

Table 4.9: Boolean functions and constants

Pass	Round (i)	Boolean function (f_i)	Constant (c_i)
1	1 – 20	$(x \wedge y) \vee (\neg x \wedge z)$	`0x00000000`
2	21 – 40	$x \oplus y \oplus z$	`0x5a827999`
3	41 – 60	$(x \vee \neg z) \oplus y$	`0x6ed9eba1`
4	61 – 80	$x \oplus y \oplus z$	`0x8f1bbcdc`

The rotation constant $s_{2,i}$ depends on the pass, i.e., it changes when the pass is changed but it is constant in each pass. The pass-dependent values of $s_{2,i}$ are:

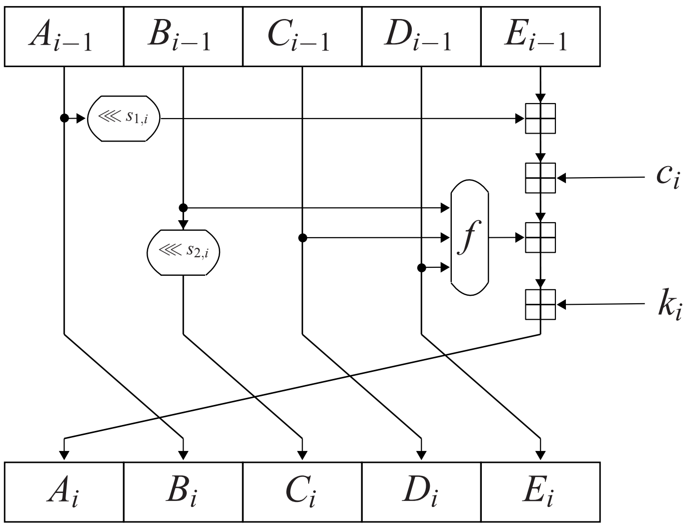


Figure 4.3: The round function of HAS-160

Table 4.10: The bit rotation s_1

Round $(i \bmod 20)+1$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
$s_{1,i}$	13	5	11	7	15	6	13	8	14	7	12	9	11	8	15	6	12	9	14	5

- Pass 1: $s_{2,i} = 10$
- Pass 2: $s_{2,i} = 17$
- Pass 3: $s_{2,i} = 25$
- Pass 4: $s_{2,i} = 30$

The 80 subkeys k_i, $i \in \{1,2,\ldots,80\}$ are derived from the key K, which consists of sixteen 32-bit words $K = x_0, x_1, \ldots, x_{15}$. The subkeys k_i are obtained from the key schedule described in Table 4.11.

Table 4.11: The key schedule

Round $(i \bmod 20)+1$	Pass 1	Pass 2	Pass 3	Pass 4
1	$x_8 \oplus x_9 \oplus x_{10} \oplus x_{11}$	$x_{11} \oplus x_{14} \oplus x_1 \oplus x_4$	$x_4 \oplus x_{13} \oplus x_6 \oplus x_{15}$	$x_{15} \oplus x_{10} \oplus x_5 \oplus x_0$
2	x_0	x_3	x_{12}	x_4
3	x_1	x_6	x_5	x_2
4	x_2	x_9	x_{14}	x_{13}
5	x_3	x_{12}	x_7	x_8
6	$x_{12} \oplus x_{13} \oplus x_{14} \oplus x_{15}$	$x_7 \oplus x_{10} \oplus x_{13} \oplus x_0$	$x_8 \oplus x_1 \oplus x_{10} \oplus x_3$	$x_{11} \oplus x_6 \oplus x_1 \oplus x_{12}$
7	x_4	x_{15}	x_0	x_3
8	x_5	x_2	x_9	x_{14}
9	x_6	x_5	x_2	x_9
10	x_7	x_8	x_{11}	x_4
11	$x_0 \oplus x_1 \oplus x_2 \oplus x_3$	$x_3 \oplus x_6 \oplus x_9 \oplus x_{12}$	$x_{12} \oplus x_5 \oplus x_{14} \oplus x_7$	$x_7 \oplus x_2 \oplus x_{13} \oplus x_8$
12	x_8	x_{11}	x_4	x_{15}
13	x_9	x_{14}	x_{13}	x_{10}
14	x_{10}	x_{14}	x_6	x_5
15	x_{11}	x_4	x_{15}	x_0
16	$x_4 \oplus x_5 \oplus x_6 \oplus x_7$	$x_{15} \oplus x_2 \oplus x_5 \oplus x_8$	$x_0 \oplus x_9 \oplus x_2 \oplus x_{11}$	$x_3 \oplus x_{14} \oplus x_9 \oplus x_4$
17	x_{12}	x_7	x_8	x_{11}
18	x_{13}	x_{10}	x_1	x_6
19	x_{14}	x_{13}	x_{10}	x_1
20	x_{15}	x_0	x_3	x_{12}

4.3.2 Some Properties

Property 2 *(from [77]) Let $Z = X \boxplus Y$ and $Z^* = X^* \boxplus Y^*$ with X, Y, X^*, Y^* being 32-bit words. Then, the following properties hold:*

1. *If $X \oplus X^* = e_j$ and $Y = Y^*$, then $Z \oplus Z^* = e_{j,j+1,\cdots,j+k-1}$ holds with probability 2^{-k} for ($j < 31, k \geq 1$ and $j+k-1 \leq 30$). In addition, if $j = 31$, $Z \oplus Z^* = e_{31}$ holds with probability 1.*

2. *If $X \oplus X^* = e_j$ and $Y \oplus Y^* = e_j$, then $Z \oplus Z^* = e_{j+1,\cdots,j+k-1}$ holds with probability 2^{-k} for ($j < 31, k \geq 1$ and $j+k-1 \leq 30$). In addition, in the case of $j = 31$, $Z = Z^*$ holds with probability 1.*

A more general description of these properties can be derived from the following theorem.

Theorem 1 *(from [102]) Given three 32-bit XOR differences $\Delta X, \Delta Y$ and ΔZ. If the probability $\Pr[(\Delta X, \Delta Y) \xrightarrow{\boxplus} \Delta Z] > 0$, then*

$$\Pr[(\Delta X, \Delta Y) \xrightarrow{\boxplus} \Delta Z] = 2^{-k},$$

where the integer k is given by $k = \#\{i | 0 \le i \le 30,$ *not* $((\Delta X)_i = (\Delta Y)_i = (\Delta Z)_i)\}$.

Property 3 *Consider the difference* $\Delta P_i = (\Delta A_i, \Delta B_i, \Delta C_i, \Delta D_i, \Delta E_i)$ *of a message pair in round i. Then we know some* 32*-bit differences in rounds* $i+1$, $i+2$, $i+3$ *and* $i+4$. *The known word differences are as follows:*

$$\begin{aligned}
(\Delta B_{i+1}, \Delta C_{i+1}, \Delta D_{i+1}, \Delta E_{i+1}) &= (\Delta A_i, \Delta B_i^{\lll s_{2,i+1}}, \Delta C_i, \Delta D_i),\\
(\Delta C_{i+2}, \Delta D_{i+2}, \Delta E_{i+2}) &= (\Delta A_i^{\lll s_{2,i+2}}, \Delta B_i^{\lll s_{2,i+1}}, \Delta C_i),\\
(\Delta D_{i+3}, \Delta E_{i+3}) &= (\Delta A_i^{\lll s_{2,i+2}}, \Delta B_i^{\lll s_{2,i+1}}),\\
(\Delta E_{i+4}) &= (\Delta A_i^{\lll s_{2,i+2}})
\end{aligned}$$

Property 4 *Consider the differences*

$$\Delta P_i = (\Delta A_i, \Delta B_i, \Delta C_i, \Delta D_i, \Delta E_i)$$

and

$$\Delta P_{i+1} = (\Delta A_{i+1}, \Delta B_{i+1}, \Delta C_{i+1}, \Delta D_{i+1}, \Delta E_{i+1})$$

of a message pair in rounds i and $i+1$, *respectively. Assume that the intermediate encryption values* P_{i+1} *and* $P'_{i+1} = \Delta P_{i+1} \oplus P_{i+1}$ *are also known. Then*

$$\begin{aligned}
A_{i+1} &= A_i^{\lll s_{1,i}} \boxplus f_i(B_i, C_i, D_i) \boxplus E_i \boxplus k_{i+1} \boxplus c_{i+1},\\
&= B_{i+1}^{\lll s_{1,i}} \boxplus f_i(B_i, C_i, D_i) \boxplus E_i \boxplus k_{i+1} \boxplus c_{i+1},\\
&= B_{i+1}^{\lll s_{1,i}} \boxplus f_i(C_{i+1}^{\ggg s_{2,i}}, C_i, D_i) \boxplus E_i \boxplus k_{i+1} \boxplus c_{i+1},\\
&= B_{i+1}^{\lll s_{1,i}} \boxplus f_i(C_{i+1}^{\ggg s_{2,i}}, D_{i+1}, E_{i+1}) \boxplus E_i \boxplus k_{i+1} \boxplus c_{i+1}.
\end{aligned}$$

We can rearrange the last formula as

$$E_i = B_{i+1}^{\lll s_{1,i}} \boxplus f_i(C_{i+1}^{\ggg s_{2,i}}, D_{i+1}, E_{i+1}) \boxplus A_{i+1} \boxplus k_{i+1} \boxplus c_{i+1}. \tag{4.1}$$

Looking at the righthand side of Equation (4.1) we can see, that the only unknown value is k_{i+1}. *For the known values* P_{i+1} *and* P'_{i+1} *we obtain two equations*

$$\begin{aligned}
E_i &= B_{i+1}^{\lll s_{1,i}} \boxplus f_i(C_{i+1}^{\ggg s_{2,i}}, D_{i+1}, E_{i+1}) \boxplus A_{i+1} \boxplus k_{i+1} \boxplus c_{i+1}, \text{ and}\\
E'_i &= B_{i+1}^{'\lll s_{1,i}} \boxplus f_i(C_{i+1}^{'\ggg s_{2,i}}, D'_{i+1}, E'_{i+1}) \boxplus A'_{i+1} \boxplus k_{i+1} \boxplus c_{i+1}.
\end{aligned}$$

From the differential for $E1$ *we know the value for* $\Delta E_i = E_i \oplus E'_i$ *and thus, the relevant bits in* k_{i+1} *and* k'_{i+1} *are determined. Let j mark the position of the most significant active bit below bit* 31 *in* ΔE_i *which is set to one. In a scenario where we want to recover the subkeys of round i, via decryption, we can only recover bits* $j, j-1, \ldots, 1, 0$ *of the subkeys of round i. This is due to the fact that the most significant bits* $31, \ldots, j+2, j+1$ *of the subkeys in round i do not influence the difference* ΔE_i.

4.3.3 Related-Key Rectangle Attack on a 77-round Compression Function of HAS-160 in Encryption Mode

In this section, we describe a 68-round related-key rectangle distinguisher, which can be used to mount a related-key rectangle attack on 77-round HAS-160 in encryption mode. We can use Property 3 to partially determine whether a candidate quartet is a right one or not. A wrong quartet can be discarded during the stepwise computation, which reduces the complexity of the subsequent steps and also the overall complexity of the attack.

4.3.4 A 68-Round Related-Key Rectangle Distinguisher

Let $K = x_0, x_1, \ldots, x_{15}$, be a key where x_i is a 32-bit word. Our attack uses four different, but related keys, K^a, K^b, K^c and K^d. The key differences are as follows:

$$\begin{aligned}\Delta K^* &= K^a \oplus K^b = K^c \oplus K^d = (e_{31},0,0,0,0,0,0,0,0,0,e_{31},0,0,0,0,0),\\ \Delta K' &= K^a \oplus K^c = K^b \oplus K^d = (0,0,0,0,0,0,0,0,0,0,0,0,e_{31},0,e_{31},0).\end{aligned}$$

Since the key schedule of HAS-160 is linear, we can easily determine all the 80 subkey differences derived from the key differences ΔK^* and $\Delta K'$, respectively. We observe that if we choose $\Delta x_0 = \Delta x_{10}$ and the remaining word differences as zero, i.e., $\Delta x_i = 0$, $i = 1,2,\ldots,8,9,11,12,\ldots,15$, then all the subkeys of rounds 14 to 37 have a zero difference. We use this observation for the related-key differential for $E0$. Moreover, we can observe that if $\Delta x_{12} = \Delta x_{14}$ and the remaining word differences in $\Delta K'$ are all zero, then all the subkeys of rounds 44 to 65 have a zero difference. This observation is used in our related-key differential for $E1$.

Considering Property 2 and Theorem 1 we found a 36-round related-key differential from round 3 to 39 for $E0$ $(\alpha \to \beta)$ using the key difference ΔK^*. The related-key differential is:

$$(e_{23}, e_{19,23}, e_{1,4,21,29}, e_{1,4,21,23,29}, e_{1,4,6,11,21,23,29}) \to (e_{4,31}, e_{31}, 0, 0, 0).$$

The related-key differential for $E0$ is shown in Table 4.12. Note that the following holds: Let Δc_i be the difference of the constants used in round i. We know that $\Delta c_i = 0$ always holds. Thus, $\Pr[(\Delta c_i, \Delta k_i) \overset{\boxplus}{\to} \Delta k_i] = 1$ always holds due to Property 2 and since Δk_i is either zero or e_{31}.

The probability for the differential for $E0$ is 2^{-29}.

We exploit a 32-round related-key differential $\gamma \to \delta$ for $E1$ that covers rounds 39 to 71 using the key difference $\Delta K'$. The related-key differential is:

$$(e_6, 0, 0, 0, e_{19}) \to (e_{5,6,7,14,17,18,19,28,29,30}, e_{5,8,9,19,21,29}, e_{5,26,27}, e_{19}, e_5)$$

Table 4.12: The Related-Key Differential for $E0$

i	ΔA_i	ΔB_i	ΔC_i	ΔD_i	ΔE_i	Δk_i	Prob.
3	e_{23}	$e_{19,23}$	$e_{1,4,21,29}$	$e_{1,4,21,23,29}$	$e_{1,4,6,11,21,23,29}$	0	2^{-5}
4	$e_{11,23}$	e_{23}	$e_{1,29}$	$e_{1,4,21,29}$	$e_{1,4,21,23}$	0	2^{-3}
5	e_{23}	$e_{11,23}$	e_1	$e_{1,29}$	$e_{1,4,21,29}$	0	2^{-6}
6	e_{21}	e_{23}	$e_{1,21}$	e_1	$e_{1,29}$	0	2^{-5}
7	0	e_{21}	e_1	$e_{1,21}$	e_1	0	2^{-1}
8	0	0	e_{31}	e_1	$e_{1,21}$	0	2^{-1}
9	e_{21}	0	0	e_{31}	e_1	0	2^{-3}
10	0	e_{21}	0	0	e_{31}	0	2^{-2}
11	0	0	e_{31}	0	0	e_{31}	2^{-1}
12	0	0	0	e_{31}	0	0	2^{-1}
13	0	0	0	0	e_{31}	0	1
14	0	0	0	0	0	e_{31}	1
15	0	0	0	0	0	0	1
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
37	0	0	0	0	0	0	1
38	e_{31}	0	0	0	0	e_{31}	2^{-1}
39	$e_{4,31}$	e_{31}	0	0	0	0	

The 160-bit difference δ can be written as a concatenation of five 32-bit word differences

$$\delta = (\delta_A, \delta_B, \delta_C, \delta_D, \delta_E) = (\Delta A_{71}, \Delta B_{71}, \Delta C_{71}, \Delta D_{71}, \Delta E_{71}). \tag{4.2}$$

The related-key differential for $E1$ is shown in Table 4.13.

The probability for $E1$ is 2^{-24}. Thus, the probability of our related-key rectangle distinguisher for rounds 1–71 is:

$$\left(2^{-29} \cdot 2^{-24}\right)^2 \cdot 2^{-160} = 2^{-266}$$

At the same time, the correct difference δ occurs in two ciphertext pairs of a quartet for a random cipher with probability $(2^{-160})^2 = 2^{-320}$.

4.3.5 The Attack

The attack works as follows:

1. Choose 2^{136} plaintexts $P^a_{0,i} = (A_{0,i}, B_{0,i}, C_{0,i}, D_{0,i}, E_{0,i})$, $i = 1, 2, \ldots, 2^{136}$. Compute 2^{136} plaintexts $P^b_{0,i}$, by setting $P^b_{0,i} = P^a_{0,i} \oplus \alpha$, $\alpha = (e_{23}, e_{19,23}, e_{1,4,21,29}, e_{1,4,21,23,29}, e_{1,4,6,11,21,23,29})$. Set $P^c_{0,i} = P^a_{0,i}$ and $P^d_{0,i} = P^b_{0,i}$. Ask for the encryption of the plaintexts $P^a_{0,i}, P^b_{0,i}, P^c_{0,i}, P^d_{0,i}$ under K^a, K^b, K^c, and K^d, respectively, and obtain the ciphertexts $P^a_{80,i}, P^b_{80,i}, P^c_{80,i}$, and $P^d_{80,i}$.

Table 4.13: The Related-Key Differential for $E1$

i	ΔA_i	ΔB_i	ΔC_i	ΔD_i	ΔE_i	Δk_i	Prob.
39	e_6	0	0	0	e_{19}	–	2^{-1}
40	0	e_6	0	0	0	0	2^{-1}
41	0	0	e_{31}	0	0	0	1
42	0	0	0	e_{31}	0	e_{31}	2^{-1}
43	0	0	0	0	e_{31}	0	1
44	0	0	0	0	0	e_{31}	1
45	0	0	0	0	0	0	1
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
65	0	0	0	0	0	0	1
66	e_{31}	0	0	0	0	e_{31}	2^{-1}
67	e_7	e_{31}	0	0	0	0	2^{-1}
68	e_{21}	e_7	e_{29}	0	0	e_{31}	2^{-3}
69	$e_{7,28,29}$	e_{21}	e_5	e_{29}	0	0	2^{-6}
70	$e_{5,8,9,19,21,29}$	$e_{7,28,29}$	e_{19}	e_5	e_{29}	0	2^{-10}
71	$e_{5,6,7,14,17,18,19,28,29,30}$	$e_{5,8,9,19,21,29}$	$e_{5,26,27}$	e_{19}	e_5	0	

2. Guess the 32-bit subkeys $k^a_{80}, k^a_{79}, k^a_{78}, k^a_{77}, k^a_{76}$, and k^a_{75}. Guess the 28 least significant bits of k^a_{74} and set its most significant bits to one. Compute $k^l_{80}, k^l_{79}, k^l_{78}, k^l_{77}, k^l_{76}, k^l_{75}, k^l_{74}$, $l \in \{b,c,d\}$ using the known subkey differences.

   2.1. Decrypt each of the ciphertexts $P^a_{80,i}, P^b_{80,i}, P^c_{80,i}, P^d_{80,i}$ under $k^l_{80}, k^l_{79}, k^l_{78}, k^l_{77}, k^l_{76}, k^l_{75}, k^l_{74}$, $l \in \{a,b,c,d\}$, respectively, and obtain the intermediate encryption values $P^a_{73,i}, P^b_{73,i}, P^c_{73,i}$ and $P^d_{73,i}$.

   2.2. Regarding the differential for $E1$ we know that the value of the 96-bit differences $\delta_{A\lll 30}$, $\delta_{B\lll 30}$ and δ_C. Using Properties 3 and 4 discard all quartets $(P^a_{73,i}, P^b_{73,i}, P^c_{73,i}, P^d_{73,i})$ which do not satisfy the following conditions:

$$\begin{aligned} C^a_{73,i} \oplus C^c_{73,i} &= \delta_{A\lll 30} = C^b_{73,i} \oplus C^d_{73,i}, \\ D^a_{73,i} \oplus D^c_{73,i} &= \delta_{B\lll 30} = D^b_{73,i} \oplus D^d_{73,i}, \\ E^a_{73,i} \oplus E^c_{73,i} &= \delta_C = E^b_{73,i} \oplus E^d_{73,i} \end{aligned}$$

3. Guess the 4 most significant bits of k^a_{74} and the 20 least significant bits of k^a_{73} and set its most significant bits to one. Compute k^l_{74}, k^l_{73}, $l \in \{b,c,d\}$ using the known subkey differences.

   3.1. Redecrypt each quartet $(P^a_{74,i}, P^b_{74,i}, P^c_{74,i}, P^d_{74,i})$ under the full subkeys k^l_{74}, $l \in \{a,b,c,d\}$. Decrypt each remaining quartet $(P^a_{73,i}, P^b_{73,i}, P^c_{73,i}, P^d_{73,i})$ under k^l_{73}, $l \in \{a,b,c,d\}$, respectively, and obtain the intermediate encryption values $P^a_{72,i}, P^b_{72,i}, P^c_{72,i}$ and $P^d_{72,i}$.

3.2. Regarding the differential for $E1$ we know that the value of the 128-bit differences δ_A, $\delta_{B\lll 30}, \delta_C$ and δ_D. Using Properties 3 and 4 discard all quartets $(P^a_{72,i}, P^b_{72,i}, P^c_{72,i}, P^d_{72,i})$ which do not satisfy the following conditions:

$$\begin{aligned} B^a_{72,i} \oplus B^c_{72,i} &= \delta_A = B^b_{72,i} \oplus B^d_{72,i}, \\ C^a_{72,i} \oplus C^c_{72,i} &= \delta_{B\lll 30} = C^b_{72,i} \oplus C^d_{72,i}, \\ D^a_{72,i} \oplus D^c_{72,i} &= \delta_C = D^b_{72,i} \oplus D^d_{72,i}, \\ E^a_{72,i} \oplus E^c_{72,i} &= \delta_D = E^b_{72,i} \oplus E^d_{72,i} \end{aligned}$$

4. Guess the 12 most significant bits of k^a_{73}. Guess the 6 least significant bits of k^a_{72} and set its most significant bits to one. Compute k^l_{73}, k^l_{72}, $l \in \{b,c,d\}$ using the known subkey differences.

   4.1. Redecrypt each remaining quartet $(P^a_{73,i}, P^b_{73,i}, P^c_{73,i}, P^d_{73,i})$ under the full subkeys k^l_{73}, $l \in \{a,b,c,d\}$. Decrypt each quartet $(P^a_{73,i}, P^b_{73,i}, P^c_{73,i}, P^d_{73,i})$ under k^l_{72}, $l \in \{a,b,c,d\}$, respectively, and obtain the intermediate encryption values $P^a_{71,i}, P^b_{71,i}, P^c_{71,i}$ and $P^d_{71,i}$.

   4.2. Regarding the differential for $E1$ we know that the value of the 160-bit differences δ. Discard all quartets $(P^a_{71,i}, P^b_{71,i}, P^c_{71,i}, P^d_{71,i})$ which do not satisfy the following conditions:

$$\begin{aligned} A^a_{71,i} \oplus A^c_{71,i} &= \delta_A = A^b_{71,i} \oplus A^d_{71,i}, \\ B^a_{71,i} \oplus B^c_{71,i} &= \delta_B = B^b_{71,i} \oplus B^d_{71,i}, \\ C^a_{71,i} \oplus C^c_{71,i} &= \delta_C = C^b_{71,i} \oplus C^d_{71,i}, \\ D^a_{71,i} \oplus D^c_{71,i} &= \delta_D = D^b_{71,i} \oplus D^d_{71,i}, \\ E^a_{71,i} \oplus E^c_{71,i} &= \delta_E = E^b_{71,i} \oplus E^d_{71,i} \end{aligned}$$

5. Output the keys $(k^l_{80}, k^l_{79}, k^l_{78}, k^l_{77}, k^l_{76}, k^l_{75}, k^l_{74}, k^l_{73}, k^l_{72})$, $l \in \{a,b,c,d\}$.

4.3.6 Analysis of the Attack

There are 2^{136} pairs (P^a_i, P^b_i) and 2^{136} pairs (P^c_i, P^d_i) of plaintexts, thus we have $(2^{136})^2 = 2^{272}$ quartets. The expected number of right quartets that remain for the right subkeys is about $2^{272} \cdot 2^{-266} = 2^6$.

The data complexity of Step 1 is $2^2 \cdot 2^{136} = 2^{138}$ chosen plaintexts. The time complexity of Step 1 is about $2^2 \cdot 2^{136} = 2^{138}$ 80-round encryptions.

In Step 2 the adversary guesses the subkeys of round 80 to 75 as well as 28-bits of the subkey of round 74. The most significant bit in ΔE_{73} is e_{27}. Due to Property 4 only bits $27, 26, \ldots, 0$ of Δk_{74} are relevant for the key recovery of k_{74} in this stage. Step 2.1 requires time about $2^{32\cdot6+28} \cdot 2^2 \cdot 2^{136} \cdot (7/80) \approx 2^{354.5}$ 80-round encryptions. The number of remaining wrong quartets after Step 2.2 is $2^{272} \cdot (2^{-96})^2 = 2^{80}$, since we have a 96-bit filtering condition on both pairs of a quartet.

In Step 3 the adversary has to guess the 4 most significant bits of k_{74}, as their values affect the difference in the previous rounds. He also guesses the 20 least significant bits of k_{73}, since the most significant active bit in ΔE_{72} is e_{19}. The time complexity of Step 3.1 is about $2^{20+4} \cdot 2^{220} \cdot 2^2 \cdot 2^{80} \cdot (1/80) \approx 2^{319.6}$ encryptions. After Step 3.2 about $2^{80} \cdot (2^{-32})^2 = 2^{16}$ wrong quartets remain, since we have a 32-bit filtering condition on both pairs of a quartet.

In Step 4 the adversary has to guess the 12 most significant bits of k_{73}, as their values affect the difference in the previous rounds. He also guesses the 6 least significant bits of k_{72}, since the most significant active bit in ΔE_{71} is e_5 following Property 4. The time complexity of Step 4.1 is $2^{6+12} \cdot 2^{24} \cdot 2^{220} \cdot 2^2 \cdot 2^{16} \cdot (1/80) \approx 2^{273.6}$ encryptions. After Step 4.2 the number of remaining wrong quartets is about $2^{16} \cdot (2^{-32})^2 = 2^{-48}$ for wrong each subkey guess, since we have a 32-bit filtering condition on both pairs of a quartet.

Using the Poisson distribution we can compute the success rate of our attack. The probability that the counter of a wrong key is at least 6 assuming $Y_i \sim Poisson(\mu = 2^{-48})$ is

$$\Pr(Y \geq 6) = e^{-2^{-48}} \cdot \frac{(2^{-48})^6}{6!} \approx 2^{-297}.$$

For all the $2^{262} - 1$ wrong keys used in our analysis we expect about $2^{262} \cdot 2^{-297} = 2^{-35}$ wrong keys which have a count of at least 6 quartets. The probability that the right key has a count of at least 6 quartets assuming $Z \sim Poisson(\mu = 2^6)$ is

$$\Pr(Z \geq 6) \approx 0.99.$$

The data complexity of our attack is $2^{136} \cdot 2^2 = 2^{138}$ chosen plaintexts. The time complexity is determined by Step 2 which is about 2^{320} 80-round HAS-160 encryptions.

Chapter 5

Cryptanalysis and Design of Hash Functions

Hash functions gained a lot of attention during the last years. Most popular hash functions such as MD5 [132], SHA-0 [118] or SHA-1 [119] possess weaknesses in their design, leading to a huge amount of attacks that were recently found [14, 15, 32, 44, 130, 149, 150, 151]. SHA-1, MD4 and MD5 where broken [150]. Thus, the interest in hash functions increases, due to the need for new one.

This chapter addresses two main parts. The first is the introduction of a new method for attacking cryptographic hash functions. The second is a new approach for the design of hash functions and a specific SHA-3 candidate that follows this methodology. This candidate was submitted and accepted for the first round of the SHA-3 competition.

5.1 Slide Attacks on a Class of Hash Functions

A natural idea for thwarting the MERKLE-DAMGÅRD limitations is to increase the size of the internal chaining variables [103] in the iterated process and add a counter into the compression function, see for example [97]. Using a similar patch, sponge functions [10] followed the idea to employ a huge internal state (to hold a huge chaining variable) and to claim a *capacity* c, typically $c \gg n$, where a capacity defines the size of the internal state and n is the size of the hash output. This defends against adversaries even if they can perform more than $2^{n/2}$ operations (but are still restricted to less than $2^{c/2}$ units of work).

In this section, we study the applicability of slide attacks to sponge functions. On one hand, our results indicate that slide attacks can be a serious threat for hash functions fitting into the sponge framework. On the other hand, if the hash function designer is aware of slide attacks, we believe that it is easy to defend against such attacks. We give concrete examples by providing attacks against GRINDAHL [97] and two slightly tweaked versions of RADIOGATÚN [8]. Our attack applies for both flavours of Grindahl, the 256-bit version and the 512-bit version.

5.1.1 Slide Attacks

Block ciphers are often designed as a keyed permutation which is applied multiple times. It is a common belief that increasing the number of rounds makes the cipher stronger, but this is true only for statistical attacks such as differential or linear cryptanalysis. Some attacks can be applied even for block cipher variants with an arbitrary number of rounds. This is true for certain related-key attacks, and for slide attacks. Slide attacks [27] are a special form of realted-key attacks that utilizes the self-similarity of the cipher, typically caused by a periodic key schedule.

Slide Attacks on Block Ciphers

Slide attacks on block ciphers have been applied to several ciphers with a weak key schedule (for example [19, 27, 28, 37, 67, 84, 125, 126]). The original slide attack [27] works as follows. An n-bit block cipher E with r rounds is split into b identical rounds of the same keyed permutation F^i for $i = \{1, \ldots, b\}$. In the simplest case we have $b = r$ where the key schedule produces the same key in each round.[1] Thus, we write the cipher as $E = F^1 \circ F^2 \circ \cdots \circ F^b = F \circ F \circ \cdots \circ F$. A plaintext P_j is then encrypted as

$$P_j \xrightarrow{F} X^{(1)} \xrightarrow{F} X^{(2)} \xrightarrow{F} \cdots \xrightarrow{F} X^{(b-1)} \xrightarrow{F} C_j$$

where $X^{(i)}$ represents the intermediate encryption value after the application of F^i and $X^{(b)} = C_j$ is the corresponding ciphertext. To mount a slide attack one has to find a slid pair (P_j, P_k), such that

$$P_k = F(P_j) \quad \text{and} \quad C_k = F(C_j) \tag{5.1}$$

hold, see also Figure 5.1 for more details.

The original slide attacks can be applied only to a small class of ciphers with weak permutations periodic key schedules. In this context, a permutation is weak if, given the two equations in (5.1), it is easy to extract a non negligible part of the secret key. With $2^{n/2}$ known plaintext/ciphertext pairs (P_i, C_i) we expect about one pair satisfying $P_k = F(P_j)$ among these texts by the birthday paradox. This gives us a slid pair. Thus,

[1]Note that F^i might include more than one round of the cipher. If the key schedule produces identical keys with period p then F^i includes p rounds of the original cipher.

$$\begin{tikzcd}
P_j \arrow[r, "F"] & X^{(1)} \arrow[r, "F"] & X^{(2)} \arrow[r, "F"] & X^{(3)} \arrow[r, "F"] & \cdots \arrow[r, "F"] & C_j & & \\
& P_k \arrow[r, "F"] & X^{(1)} \arrow[r, "F"] & X^{(2)} \arrow[r, "F"] & \cdots \arrow[r, "F"] & X^{(b-1)} \arrow[r, "F"] & C_k &
\end{tikzcd}$$

Figure 5.1: A slid pair for a block cipher

the classical slide attack allows to recover the unknown key of an n-bit block cipher using $O(2^{n/2})$ known plaintexts and $O(2^n)$ time. In the case of Feistel ciphers the data complexity is $O(2^{n/4})$ chosen plaintexts, and slid pairs are easier to identify because of its structure.

Advanced sliding techniques like the *complementation slide* and *sliding with a twist* were introduced by Biryukov and Wagner [28]. These techniques allow to attack ciphers with a more complex key schedule. The basic concept of the complementation slide is to slide two encryptions against each other where the inputs to the rounds may have a difference, which is canceled out by a difference in the keys, while an encryption is slid against a decryption using a sliding with a twist technique. Biham et al. [19] improved the slide attack to detect a large amount of slid pairs very efficiently by using the relation between the cycle structure of the entire cipher and that of the keyed permutation. Other improvements of the slide attacks are discussed in [30, 37].

Slide Attacks on Hash Functions

Slide attacks in hash function settings have attracted very little attention in the literature. To our knowledge, there is one paper that considers an attack on the internal block cipher of SHA-1 [135] and another paper which uses a slide attack against the ESSENCE hash function [114]. However, this cannot be transformed into a practical attack on the hash function so far.

The application of slide attacks to hash function is different in some sense than when applying it on block ciphers. Block ciphers depend on a secret key, and slide attacks are typically employed to distinguish a block cipher from a random permutation or used for key recovery attacks.

In the hash function case, there is no secret key to recover, just a message to be hashed, and the adversary is allowed to know this message – or even to choose it. Typical attacks on hash functions are about finding collisions or preimages – and it is hard to see how slide attacks could be employed in this context. But even for hash functions, a slide property that can be detected with some significant probability allows us to differentiate the scheme from a random oracle. Indeed, with such a property, one can show a non random behavior of the hash function. Going further, when secret data is used as a part of the input of the hash function, one can try to recover some information from the hash function. The natural primitive where hash functions handle secret data are the Message Authentication Codes (MAC), that permit to authenticate a message M with a symmetric secret key K. For example, constructions such as HMAC [5] are implemented in a lot of different applications and make only two calls to the hash function. HMAC has the advantage of being secure when the internal function is *weakly collision resistant* [4] and also provides secure MACs with MERKLE-DAMGÅRD-based hash functions [5]. Note that a HMAC-based patch is one of the new domain extension algorithm proposed by Coron et al. [36] to thwart the simple MD-based MAC attacks.

Generally speaking, a good hash function H should provide a good MAC with the following computations: $\text{MAC}(K,M) = H(K||M)$ or $\text{MAC}(K,M) = H(K||M||K)$. Just like for block ciphers, if the hash function considered is not secure, one may be able to recover some non negligible part of the secret key K with a slide property that can be detected with a good probability. We also note some relevant work from Sasaki et al. [136] that attacks prefix, suffix and hybrid approaches for MAC constructions by using inner collisions for MD4, and a work from Preneel and Van Oorschot [128] that studies the envelope approach instantiated with MD5.

Slide Attacks on "Extended" Sponge Constructions

We use the "extended" sponge functions, a more general framework for our attack.

The "Extended" Sponge Framework Assume that H is an iterative hash function with an internal state of c words of p bits each and a final output size of n bits. Let $M = M^1||M^2||\cdots||M^l$ be the $m \times p$-bit blocks of the message to hash with $M^l \neq 0^{m\times p}$ (the message is padded before being split into blocks). Let M^i be the message block hashed at each round i and X^i be the internal state after processing M^i, with $X^0 = IV$. We then have $X^i = F(S(X^{i-1}, M^i))$, where F is the round function and S defines how the message is incorporated into the internal state. Once all the l message blocks have been processed, r blank rounds (rounds with no message input) are applied, i.e., for $i = l, \ldots, l+r-1$, compute $X^i = F(X^{i-1})$. Let $A := X^{l+r}$ be the final internal state. Finally, we derive n output bits by using the final output function $T(X^{l+r})$. Such a hash function can be written as

$$H(M) = X^0 \xrightarrow{F(S(X^0,M^1))} \cdots \xrightarrow{F(S(X^{l-1},M^l))} X^l \overbrace{\xrightarrow{F(X^l)} \cdots \xrightarrow{F(X^{l+r-1})}}^{\text{r times}} \overbrace{X_i^{l+r}}^{A} \xrightarrow{T(A)} T_A,$$

where T_A represents the hash output. We note that due to efficiency reasons, given the large internal state, F is usually a quite light and introduces little nonlinearity.

This framework is really general and especially more general than the original sponge framework one. More precisely, in the original model, S introduces the message blocks by XORing them to particular positions of the internal state. However, in our description, we can also consider a function S that replaces some bits of the internal state by the message bits. We call the former *XOR sponge* and the latter *overwrite sponge*.

Moreover, in the original model, the final output function T, may continue to apply some blank rounds and extract some bits from the internal state at the end of each application, until n bits have been received. In our framework we can also consider the case where the output bits come from a direct truncation of the final internal state A, and we call it a *truncated sponge*.

There are two security issues related to the general design of sponge functions. One issue is *invertibility*: one can run the function F in both directions. The second issue is *self-similarity*: all the blank rounds behave identically, and even a normal round can behave as a blank round if we have $X^{i-1} = F(S(X^{i-1}, M^i))$ (the effect of adding the message block is void). In the case of an XOR sponge we require $M^i = 0$ and in the case of an overwrite sponge we require that M_i is equal to the overwritten part of the internal state.

We exploit the self-similarity for a slide attack. The idea is that if one message $M_1 = M^1||\dots||M^l$ is the prefix of a message $M_2 = M^1||\dots||M^l||M^{l+1}$, than the extended state after processing the first l blocks is the same. If X^{l+1} is equal to $F(X^l, M^{l+1})$, processing the next message block M^{l+1} for the longer message is the same as the first blank round when hashing the shorter message. The extended states remain identical. We call these two messages a *slid* pair: the two final internal states are just one permutation away $B := X_j^{l+r+1} = F(X_i^{l+r})$. The slide attack is shown in Figure 5.2. Finding a

Figure 5.2: A slid pair

$$\begin{array}{ccccc}
 & X_i^0 & & X_j^0 & \\
F(S(X_i^0, M^0)) & \downarrow & \leftarrow M^0 \rightarrow & \downarrow & F(S((X_j^0, M^0)) \\
 & \vdots & & \vdots & \\
F(S(X_i^{l-1}, M^l)) & \downarrow & \leftarrow M^l \rightarrow & \downarrow & F(S((X_j^{l-1}, M^l)) \\
 & X_i^l & & X_j^l & \\
F(X_i^l) & \downarrow & M^{l+1} \rightarrow & \downarrow & F(S((X_j^l, M^{l+1})) \\
 & \vdots & & X_j^{l+1} & \\
F(X_i^{l+r-1}) & \downarrow & & \downarrow & F(X_j^{l+1}) \\
 & X_i^{l+r} = A & & \vdots & \\
T(A) & \downarrow & & \downarrow & F(X_j^{l+r}) \\
 & T_A & & X_j^{l+r+1} = B & \\
 & & & \downarrow & T(B) \\
 & & & T_B &
\end{array}$$

slid pair depends on the output function T. When T is defined as in the original sponge framework, it is very easy to detect a slid pair: most of the output bits are equal, just shifted by one round. If T is a truncation, we need to do a case by case analysis depending on the strength of the round function F and the number of bits thrown away. Yet finding a slid pair already allows us to differentiate the hash function from a random oracle, if it appears more (or less) often than a random oracle.

We can try to go further, by attacking a MAC with prefix key, i.e., $\text{MAC}(K,M) = h(K||M)$. Note that such a construction makes sense as using HMAC based on a sponge hash function turns out to be very inefficient. This is due to the fact that hashing very short messages (required in HMAC by the second hash function call) is quite slow because of the blank rounds. Therefore, Bertoni et al. [11] propose to use prefix-MAC instead of HMAC. We note that due to the finalization this MAC does not suffer from length extension when sponges are used.

Consider a secret key K. For simplicity and without loss of generality, we assume that some K is uniformly distributed $(k \times m \times p)$-bit random value (i.e., m-message words long), for some public integer constant k. We write $K = (K^1, \ldots, K^m) \in (\{0,1\}^{m\times p})^k$ or $\{0,1\}^{c\times p}$ if $c < k \times m$. The adversary is allowed to choose message challenges C_i, while the oracle replies with $\text{MAC}(K,C_i) = H(K||C_i)$. Ideally, finding K in such a scenario would require the adversary to exhaustively search over the set of all possible $K \in \{0,1\}^{k\times m\times p}$, thus taking $2^{k\times m\times p-1}$ units of time on average. Forging a valid MAC depends on the size of the hash output and the size of the key, with a generic attack it requires $\min\{2^{k\times m\times p-1}, 2^n\}$ units of time, where n is the size of the hash output. A pair of challenges (C_i, C_j), with $C_i = C_i^1||C_i^2||\cdots||C_i^l$ and $C_j = C_i||C_j^{l+s}$, $s \in \{1,2,\ldots,r\}$ is called a slid pair for K if their final internal state are slid by one application of the blank round function as:

$$X_j^{k+l+s+1} = F(X_i^{k+l+s})$$

Provided that one can generate slid pairs and detect them, one can also try to retrieve the internal state X_i^{k+l+s} thanks to this information. Again, a case by case analysis is required here. When X_i^{k+l+s} is known, one can invert all the blank rounds and get X_i^{k+l}. In the following, our slid pair consists of two message where the longer message contains one additional block. Note that with this information, an adversary can directly forge valid MACs for any message that contains M as prefix (exactly like the extension attacks against MERKLE-DAMGÅRD-based hash functions). In case the round function with the message is invertible, we can continue to invert all the challenge rounds and get X_i^k. This allows us to recover some non trivial information on the secret key K and to forge MACs for all messages.

A general outline of the attack is as follows:

1. Find and detect slid pairs of messages
2. Recover the internal state
3. Uncover some part of the secret key or forge valid MACs

The padding is very important here: for the XOR sponge functions, an appropriate padding can avoid slide attacks. Indeed, in this case, we require $M^l = 0^{m\times p}$ to get a slid pair. This gives an explanation why the condition $M^l \neq 0^{m\times p}$ is needed for the indifferentiability proofs of XOR sponge functions. However, for the truncated sponge function, a padding is ineffective in avoiding slide attacks.

5.1.2 Applications of Slide Attacks

The GRINDAHL Hash Function

GRINDAHL is a new hash function introduced by Knudsen et al. in [97], that fits the extended sponge framework. More precisely, it is an overwrite sponge function. There are two concrete instantiations of the GRINDAHL hash function family: a 256-bit and a 512-bit hash function are proposed in the original GRINDAHL paper [97]. We give a short description of GRINDAHL in the following. For a detailed description the reader is refered to [97]. The parameters of these instantiations in the framework are defined as follows. The internal state of GRINDAHL can also be viewed as a matrix. Therefore, we define N_r and N_c to be the number of rows and columns of p-bit word, respectively: we have $N_r \times N_c = c$.

Grindahl-256 [97]. Grindahl-256 is a 256-bit hash function with $N_r = 4$ and $N_c = 12$. The rotation amounts are $(\rho_0, \dots, \rho_3) = (1, 2, 4, 10)$.

Grindahl-512 [97]. Grindahl-512 is a 512-bit hash function with $N_r = 8$ and $N_c = 12$. The rotation amounts are $(\rho_0, \dots, \rho_7) = (1, 2, \dots, 8)$.

For each instance of GRINDAHL we have $p = 8$. The message chunk entering at each round can then be viewed as one column, thus $m = N_r$.

For GRINDAHL the padding consists of 10-padding and length-padding:

1. *10-padding* appends a “1”-bit to the message, followed by as many “0”-bits as needed to complete the last message block.

2. *Length-padding* then appends the number of message blocks (not bits!) for the entire padded message as a 64-bit value.

One effect of the 10-padding is that the last message block before the Length-padding can be any value, except for the all-zero block. Or equivalently, any nonzero block B can be split up into an incomplete block R plus 10-padding: $B = R + P^{“10''}$. Note that R is 0-bit long if $B = 1000\dots0$.

A message $M = M^1||\dots||M^l$ of 32-bit blocks M^i in the case of GRINDAHL-256, and an incomplete block M^l, is padded to $Pad(M) = M^1||\dots||M^l + P_1^{“10''}||M^{l+1}||M^{l+2}$, where $P_1^{“10''}$ is the 10-padding. This padded message has the following properties:

1. The last-but-two message blocks are not zero: $M^l + P_1^{“10''} \neq 0^{32}$.
2. The final two message blocks contain the 64-bit integer l: $(M^{l+1}||M^{l+2}) = l$. (From the GRINDAHL sample implementation, we note that the 32 least significant bits of the 64-bit value are stored in M^{l+2}, while the high-significant bits go into M^{l+1}.)

Similarly for GRINDAHL-512, a message $M = M^1||\dots||M^l$ of 64-bit blocks M^i, where M^l is also incomplete, is padded to $Pad(M) = M^1||\dots||M^l + P_1^{“10''}||M^{l+1}$ has the following properties after padding:

1. The last-but-one message blocks are not zero: $M^l + P_1^{“10''} \neq 0^{64}$.
2. The last message block contains the 64-bit integer l: $M^{l+1} = l$.

After the message insertion, the output is the first $n/(p \times N_r)$ columns of of the final internal state, i.e., GRINDAHL is a truncated sponge. The compression function takes one m-word message block and an $(N_r \times N_c)$-word internal state as its input and generates a new internal state (again of size $(N_r \times N_c)$ words), as its output.

GRINDAHL follows a general three-step design strategy. An m-word message block, which is written as M^i is injected into the internal state which is of size $(N_r \times N_c)$ words, which is written as a N_c tuple of N_r-words as $(X^1,\dots,X^{N_c}) \in (\{0,1\}^{p\times N_r})^{N_c}$. The injection step which concatenates a message block to the internal state is straightforward:

$$\begin{aligned} S: \{0,1\}^{p\times N_r} \times \{0,1\}^{p\times N_r\times N_c} &\rightarrow \{0,1\}^{p\times N_r + p\times N_r\times N_c}, \\ S(M^i,(X^1,\dots,X^{N_c})) &= (M^i, X^1,\dots,X^{N_c}). \end{aligned}$$

The $(p \times N_r + p \times N_r \times N_c)$-bit output of the injecting S, is the *extended state* labeled by $(X^0,\dots,X^{N_c})$. The second step is a non linear permutation over the extended state:

$$\begin{aligned} F: \{0,1\}^{p\times N_r + p\times N_r\times N_c} &\rightarrow \{0,1\}^{p\times N_r + p\times N_r\times N_c}, \\ F(X^0,\dots,X^{N_c}) &= (Y^0,\dots,Y^{N_c}). \end{aligned}$$

F is a permutation based on RIJNDAEL [40]:

$$\begin{aligned}&\mathrm{F}(X^0,\ldots,X^{N_c})\\&\quad = \mathrm{MixColumns} \circ \mathrm{ShiftRows} \circ \mathrm{SubBytes} \circ \mathrm{AddConstant}(X^0,\ldots,X^{N_c}).\end{aligned}$$

MixColumns. Is a linear matrix multiplication of each state column with a constant matrix over GF(2^8). This transformation is defined as in the RIJNDAEL specifications for the 256-bit version of GRINDAHL. An 8×8 MDS matrix is proposed for the 512-bit version. We do not treat this matrix because we do not need it in our analysis.

ShiftRows. This transformation cyclically shifts each row by a few bytes. The i-th row is rotated by ρ_i positions to the right. For GRINDAHL-256 the rotation constants are $(1,2,4,10)$, and $(1,2,3,4,5,6,7,8)$ for GRINDAHL-512.

SubBytes. The only non-linear part of the permutation, defined as function applying RIJNDAEL's S-box to each and every byte.

AddConstant. This function is a simple XORing of the state matrix with a constant matrix M of the same size, where all bytes are zero except for the last one which is set to `01`.

We refer the reader to [97] for a detailed description of GRINDAHL. The third operation, R, is as straightforward as the first one – the first $p \times N_r$-bits of the $(p \times N_r + p \times N_r \times N_c)$-bit extended state are truncated away, to get a new $p \times N_r \times N_c$-bit internal state $(Y^1,\ldots,Y^{N_c})$:

$$\mathrm{R}\colon \{0,1\}^{p\times N_r + p\times N_r\times N_c} \to \{0,1\}^{p\times N_r\times N_c},\ \mathrm{R}(Y^0,\ldots,Y^{N_c}) = (Y^1,\ldots,Y^{N_c}).$$

See Figure 5.3 for a visual illustration of this design strategy. Note that the final truncation in one iteration and the initial concatenation of the b-bit message block in the next iteration together are simply overwriting the corresponding column of the extended internal state.

Let α be the internal state matrix with N_c columns and N_r rows, while $\hat{\alpha}$ represents the extended internal state with $N_c + 1$ columns and N_r rows. For a padded message $M = M^1||\ldots||M^d$ the GRINDAHL hash function does for $0 < i < d$:

$$\alpha \quad \leftarrow \quad R(P(S(M^i,\alpha)))$$

For the last message input M^d GRINDAHL performs $\hat{\alpha} \leftarrow P(S(M^d,\alpha))$. The truncation R is omitted after the last message input and finally 8 blank rounds with no message input are performed. These rounds only consists of the P operation on $\hat{\alpha}$. The n-bit output remains after applying the output truncation T defined as:

$$\mathrm{T}\colon \{0,1\}^{p\times N_r + p\times N_r\times N_c} \to \{0,1\}^{n},\ \mathrm{T}(Y^0,\ldots,Y^{N_c}) = (Y^1,\ldots,Y^{n/(p\times N_r)}).$$

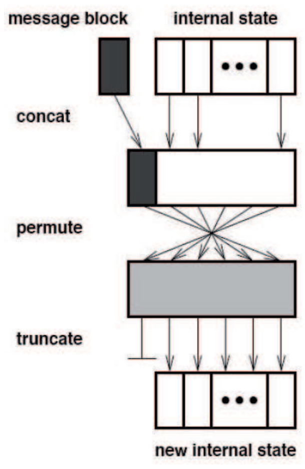


Figure 5.3: The general design of the GRINDAHL compression function

Slide Attacks on GRINDAHL**-512**

Finding Slid Pairs Building the challenge that generates a slid pair works as follows. We choose a message $M_1 = M_1^0||M_1^1||\dots||M_1^{l-1}||M_1^l$, where M_1^l is a non complete block which is padded. The MAC therefore processes

$$Pad(K||M_1) = K||M_1^0||M_1^1||\dots||M_1^{l-1}||M_1^l + P_1^{\text{“10”}}||P_1^L$$

where $P_1^{\text{“10”}}$ is the 10-padding to M_1^l and P_1^L is the one-block of the message length. The value of P_1^L can be chosen by the adversary while modifying the message length. For each M_1 we build the message $M_2 = M_1^0||M_1^1||\dots||M_1^{l-1}||M_1^l + P_1^{\text{“10”}}||R$, where R is a random incomplete block. The MAC then computes

$$Pad(K||M_2) \quad = \quad K||M_1^0||M_1^1||\dots||M_1^{l-1}||M_1^l + P_1^{\text{“10”}}||R + P_2^{\text{“10”}}||P_2^L$$

and in some cases we have

$$Pad(K||M_2) \quad = \quad K||M_1^0||M_1^1||\dots||M_1^{l-1}||M_1^l + P_1^{\text{“10”}}||P_1^L||P_2^L.$$

The messages M_1 and M_2 only differ in one additional block at the end. Such a pair (M_1, M_2) is a slid pair with probability 2^{-64}, which is the probability that a random input message block is of the correct form.

Detecting a slid pair is quite simple. Let $T_A = A^0, \dots, A^7$ and $T_B = B^0, \dots, B^7$ be the queries' output (the truncated final internal states A and B). Then the condition $B = P(A)$ holds only for a slid pair. We can not directly apply another blank round to A since we only know T_A and not A. However, T_A and T_B leave enough information for detecting a slid pair. We can invert T_B one blank round and compare the resulting bytes with the known bytes from T_A. Thus, we can compare 34 bytes of T_A with the known bytes obtained from inverting T_B. Figure 5.4 shows the backward computation of one blank round.

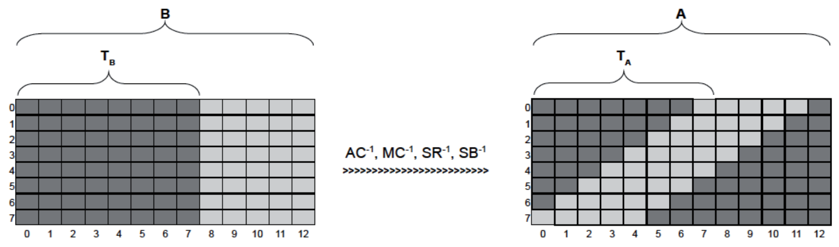


Figure 5.4: Detecting a slid pair of messages for GRINDAHL-512. Cells in dark gray mark known bytes while cells in light gray mark unknown bytes. The inverse MixColumns (MC^{-1}) and the inverse ShiftRows (SR^{-1}) are the only two operations which are important for our analysis: AddConstant and SubBytes functions leave a known (respectively, unknown) bytes known (respectively, unknown).

Recovering the Internal State A challenge (M_1, M_2) which produces a slid pair (T_A, T_B) can be used to recover the final internal state A (corresponding to the computation of M_1) just before the final truncation. Since columns A^8 to A^{12} are unknown, we have to recover 40 bytes of the unknown state. As shown in Figure 5.4, we can directly recover 30 bytes from A by applying to T_B one blank round backwards, exactly as when we tried to detect slid pairs: we can fully invert the MixColumns transformation for the eight first columns (where all the bytes are known), then it is also very easy to invert ShiftRows, SubBytes and AddConstant transformations. So, when looking at Figure 5.4, it is clear that the adversary can directly get 30 out of the unknown bytes of A. The remaining 10 unknown bytes can be recovered in a different way. For each possibility among those bytes ($2^{8\cdot 10} = 2^{80}$ possibilities), we invert all the blank rounds and check if the last added word (the first encountered when computing backwards) is P_1^L. Indeed, when inverting the real internal state A, we surely come to the insertion of P_1^L and this can be easily detected since we know this message block and since the message insertion overwrite the first column of the internal state. Now we are dealing with $2^{80-64} = 2^{16}$ possibilities only and we have to be careful, since some bytes are undetermined, if we continue the backward computation. The undetermined bytes are those which are replaced by the inserted message words during the message input step (due to the overwriting). We can compute one more round backward to check if we finally obtain the message word $M_1^l + P_1^{\text{“10”}}$ inserted. This gives us the complete internal state A. We need 2^{64} pairs of M_1 and M_2 for the attack to work.

Uncovering some Parts of the Secret Key or forge valid MACs By knowing the whole internal state A it is straightforward to invert the blank rounds. With this information, we can directly generate new valid MACs for messages which contain M_1 as prefix; by just continuing the computation of the hash function ourselves.

We can also try to invert the rounds where known message words are inserted. Some parts of the internal state are undetermined because of the truncation when adding message words as mentioned in the previous section. We can guess those undetermined columns by only keeping those, which lead to the correct inserted message words in the first column. This is equal to what we did above to recover the final internal state. By trying all the possible values of the truncated column, we can continue going backwards and check which one leads to the known correct values of the message blocks inserted a few rounds before. Some trials lead to wrong message blocks inserted and can be discarded. The ones leading to the good values have a good chance to be the real erased bytes. Thus, we can go backward for all the known message words and recover the erased columns until we have to stop this procedure when we reach the unknown secret key word. The complexity for this step is in the worst case equivalent to 2 compression function calls for each guess, which are in total 2^{65} compression function calls per message block. This is negligible compared to the complexity of the other steps of the attack for messages up to 2^{15} blocks. We need two compression function calls since we have to invert two rounds in order to check if the guess of the inserted message block is valid. The last unknown column which can be recovered is the column before inserting M_1^2. Now, with all this information we can recover 4 bytes from the 8 of the last unknown message block we encounter (the first when computing backward), which is part of the secret key. The rest of the secret can be then computed exhaustively (at a lower cost than brute force without slide attacks) or we can use a trick.[2] Indeed, we know that the initial internal state is equal to zero and one can accelerate the secret recovery with a meet-in-the-middle attack: we compute forward from the known initial internal state and we compute backward as described earlier. Note that we do not look for a collision in the internal state. The meet-in-the-middle attack is used only to test wether a guessed value of the internal state is valid or not.

Slide Attacks on GRINDAHL-256

Finding Slid Pairs

Building the challenge that generates a slid pair works as follows. We choose a message $M_1 = M_1^0||M_1^1||\dots||M_1^{l-1}||M_1^l$, where M_1^l is a non complete block which is padded. The MAC therefore processes the hash input

[2]If the size of the key is not too big, we do not even require to do any exhaustive search.

$$Pad(K||M_1) = K||M_1^0||M_1^1||\dots||M_1^{l-1}||M_1^l + P_1^{\text{“10”}}||P_1^{L1}||P_1^{L2},$$

where $P_1^{\text{“10”}}$ is the 10-padding of M_1^l and $P_1^{L1}||P_1^{L2}$ is the two-block of the message length. Before building the second message, we want the condition

$$0^n \neq P_2^{L1} = P_1^{L2}$$

to always hold for M_1. Then, for each M_1 we build the message $M_2 = M_1^0||M_1^1||$ $M_i^2||\dots||M_1^{l-1}||M_1^l + P_1^{\text{“10”}}||R$, where R is an incomplete block which, after 10-padding, is the same as P_1^{L1}. As P_1^{L1} is non-zero, such an R exists. In this case the padded message is

$$\begin{array}{rcllll} Pad(K||M_2) & = & K||M_1^0||M_1^1||\dots||M_1^{l-1}||M_1^l + P_1^{\text{“10”}}|| & R + P_2^{\text{“10”}} & ||P_2^{L1}||P_2^{L2} \\ & = & K||M_1^0||M_1^1||\dots||M_1^{l-1}||M_1^l + P_1^{\text{“10”}}|| & P_1^{L1} & ||P_1^{L2}||P_2^{L2}. \end{array}$$

Note that the one before last word is equal to the last message word of M_1 after padding. This holds because of the conditions fulfilled by P_2^{L1} and P_1^{L2}. In other words, M_1 and M_2 only differ in an additional block at the end. Such a pair (M_1, M_2) is a slid pair with a probability of 2^{-32}. In this case $P_2^{L1} = P_1^{L2}$ holds with probability one. Detecting a slid pair is as simple as in the case of GRINDAHL-512. For a slid pair the condition $B = P(A)$ holds. Let B_i be column i of B and let A_i be column i of A, respectively. T_A leaves enough information to compute the column B^4 by performing one blank round on T_A. In this way the output (T_A, T_B) of a challenge (M_1, M_2) can be checked for a value of B^4. We can further check by using other columns than B^4, even if for them only a subspace of the potential solutions is determined by T_A. On average, we need 2^{32} pairs of length about 2^{32} until we find a slid one. Thus, we need to hash 2^{32} values to obtain and detect a slid pair. Figure 5.5 shows the backward computation of one blank round.

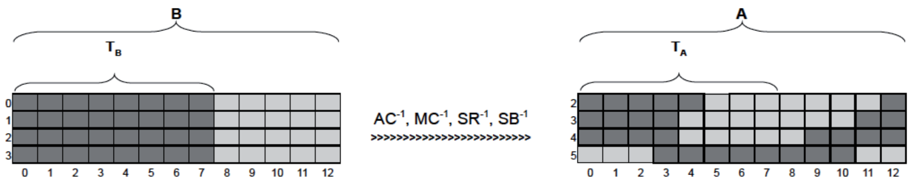


Figure 5.5: Detecting a slid pair of messages for GRINDAHL-256. Cells in dark gray mark known bytes while cells in light gray mark unknown bytes. The inverse Mix-Columns (MC^{-1}) and the inverse ShiftRows (SR^{-1}) are the only two operations which are important for our analysis: AddConstant and SubBytes functions leave a known (respectively, unknown) bytes known (respectively, unknown).

Recovering the Internal State

A challenge (M_1, M_2) which produces a slid pair (T_A, T_B) can be used to recover the final internal state A (corresponding to the computation of M_1) just before the final truncation. Since the columns A^8 to A^{12} are unknown, we need to recover 20 bytes. We can directly recover 10 bytes from A by computing one blank round backward from T_B, exactly as when we tried to detect slid pairs: we can fully invert the MixColumns transformation for the eight first columns (where all the bytes are known), and it is also very easy to invert ShiftRows, SubBytes and AddConstant transformations. So, when considering Figure 5.5, it is obvious that the adversary can directly get 10 unknown bytes from A. The remaining 10 unknown bytes can be recovered in a different way. For each possibility among those bytes ($2^{8 \cdot 10} = 2^{80}$ possibilities), we invert all the blank rounds and check if the last added word (the first encountered when computing backward) is P_1^{L2}. Since we have to invert eight blank rounds the complexity of this step is equivalent to $2^3 \cdot 2^{80} = 2^{83}$ compression function calls. Note that this step is also the most time consuming part of the attack, while the time complexity of the later steps is negligible as shown in the attack on the 512-bit version.

Indeed, when inverting the real internal state A, we surely come to the insertion of the block P_1^{L2} and this can be easily detected since we know this message block and since the message insertion overwrites the first column of the internal state. We can continue to compute backward with the word P_1^{L1} even if some parts of the internal state at this point becomes undetermined due to the truncation when inserting the message words and thus we only have $2^{48-32} = 2^{16}$ possibilities. Finally, we can continue to the message word $M_1^l + P_1^{\text{“10”}}$ which leads to a recovery of the full internal state A. We have to invert three rounds in order to check if the guess of the truncated column is valid. The complexity of this step is equivalent to $2^{33.5}$ compression function calls. In total recovering the internal state costs about 2^{83} compression function calls.

Using only short Messages

Note that the above attack requires that $0^n \neq P_2^{L1} = P_1^{L2}$, i.e., the most significant and the least significant word of the length field of $(K||M_1)$ must be the same – and nonzero. Thus, the smallest possible choice for $P_2^{L1} = P_1^{L2}$ is $P_2^{L1} = P_1^{L2} = 1$, implying a message length (for $(K||M)$, i.e., including the key) of $1 + 2^{32}$ blocks. If dealing with such long messages is an issue, we can modify the attack to use shorter messages. The modified attack goes as follows.

We choose a message $M_1 = M_1^0||M_1^1||\ldots||M_1^{l-1}||M_1^l + P_1^{\text{“10”}}$, where the final block M_1^l is incomplete. The MAC processes the hash input

$$Pad(K||M_1) = K||M_1^0||M_1^1||\ldots||M_1^{l-1}||M_1^l + P_1^{\text{“10”}}||P_1^{L1}||P_1^{L2},$$

with a length-field $P_1^{L1}||P_1^{L2}$. Note that P_1^{L2} holds the 32 least significant bits, while P_1^{L1} holds the 32 most significant bits. We assume short messages, thus $P_1^{L1} = 0^n$. This time, we want the MAC to process the hash input

$$\begin{aligned}
&Pad(K||M_2)\\
&\quad = K||M_1^0||M_1^1||\dots||M_1^{l-1}||M_1^l + P_1^{“10''}||P_1^{L1}||S + P_2^{“10''}||P_2^{L1}||P_2^{L2}\\
&\quad = K||M_1^0||M_1^1||\dots||M_1^{l-1}||M_1^l + P_1^{“10''}||P_1^{L1}||P_1^{L2}||P_2^{L1}||P_2^{L2},
\end{aligned}$$

Thus, M_1 and M_2 differ only in *two* additional blocks at the end. Accordingly, we choose

$$M_2 = M_1^0||M_1^1||\dots||M_1^{l-1}||M_1^l + P_1^{“10''}||P_1^{L1}||S.$$

As P_1^{L2} is nonzero, an incomplete block S with $S + P_2^{“10''} = P_1^{L2}$ does exist.

Now we define M_1 and M_2 as a slid-by-two pair. When processing the shorter message M_1, the first two empty rounds behave exactly as the last two nonempty rounds when processing M_2. This happens with a probability of $(2^{-32})^2$, and on the average, we need 2^{64} pairs to find slid-by-two pair.

A pair of messages is a slid-by-two pair, if and only if the two corresponding states A and B satisfy $B = P(P(A))$. Detecting slid-by-two pairs from $T(A)$ and $T(B)$ and then recovering the internal state A is slightly more complicated, compared to “ordinarily” slid-by-one pairs, but still feasible. By applying a meet-in-the-middle approach we do a forward computation of the output after processing the shorter message. Then we do a backward computation of the output of the longer message and check if there are possible matches of both the remaining states. If so, a slid-by-two pair is found. As our meet-in-the-middle attack uses 19 bytes for comparison, the probability that a wrong pair is detected as a slid pair is $2^{-8\cdot 19} = 2^{-152}$.

Uncovering some Parts of the Secret Key or forge valid MACs

By knowing the whole internal state A it is straightforward to invert the blank rounds. With this information, we can directly generate new valid MACs for messages which contain M as prefix: we just have to continue the computation of the hash function by ourselves.

We can also try to invert the rounds where known message words are inserted. Some parts of the internal state are undetermined because of the truncation when adding message words. We do not know what was in the first column before erasing it with a message word, except for the first undetermined column which is equal to P_1^{L2} as described above. But we can guess those undetermined columns by only keeping those, which lead to the good inserted message words in the first column. This is equal to what we did previously to recover the final internal state. By trying all the possibles values of the truncated column, we can continue going backward and check which one leads to the known correct values of the message blocks inserted a few rounds before. Some trials lead to wrong message blocks inserted and can be discarded. The one leading to the good values have a good chance to be the real erased bytes. Thus, we can go backward for all the known message words and recover the erased columns until we have to stop this procedure when we reach the unknown secret key word. The last unknown column which can be recovered is the column before inserting M_1^3. Now, with all this information we can recover one column of the last unknown message block we encounter (the first when computing backward), which is part of the secret key. The rest of the secret key can be then computed exhaustively (at a lower cost than brute force without slide attacks) or we can use a meet-in-the-middle attack: we compute forward from the known initial internal state and we compute backward as we described before.

Protecting against Slide Attacks

It only takes a negligible effort to protect hash functions against slide attacks. Hash function designers, like block cipher designers, must be aware of possible slide attacks and be on guard for too much self-similarity in their constructions. For sponge-based hash functions, a simple patch would be to just to add a nonzero constant just before running the blank rounds and extracting the hash value. Another option is to marginally change the blank rounds. For example, Grindahl could be changed such that the blank rounds use different rotation amounts (while maintaining the old rotation amounts for all other rounds).

5.2 The SHA-3 Candidate TWISTER

The new hash algorithm, SHA-3 [121] should serve as the new standard for hashing. It should be a replacement to the entire SHA-2 [120] family and must provide message digests of size 224, 256, 384 and 512 bits. The winner of the NIST hash function competition will define the new SHA-3 standard for hash functions. The competition started in october 2008 where NIST received 64 submissions. From these, 51 were selected as meeting the minimum submission requirements and were accepted to participate in round one. An overview of all round one SHA-3 candidates as well as a classification in terms of there underlying primitives and claims made by the authors is given in [52]. One of the candidates accepted for the first round of the SHA-3 competition is TWISTER [50, 51, 53, 56]. In the second round 14 of these 51 candidates have be selected. TWISTER was excluded because of some security issues concerning the 512-bit version of it, which we discuss below.

5.2.1 Introduction

One of the most difficult tasks for cryptographers is to design a hash function which reaches a claimed security level *and* is efficient as well. It is often only a trade off between security and speed. On one hand, a hash function with a huge security margin can offer a high level of security but might be useless for practical applications. On the other hand, a very fast hash function might offer some unexploited weaknesses, while it is used for crucial applications.

TWISTER is a new family of cryptographic hash functions. The core of TWISTER consists of the well studied wide trail strategy known from the AES winner Rijndael. The biggest advantage of this construction is its simplicity and the well studied security analysis. This makes the TWISTER hash family very easy to analyze.

TWISTER is defined for different internal state sizes and a variety of output sizes from 32 bits to 512 bits. This allows TWISTER to be a drop-in replacement for the SHA family of hash functions.

The main advantages of the TWISTER hash function family are:

- portable to many platforms, e.g., 8-, 32-, 64-bit
- using well established and studied design concepts of wide trails
- high speed (comparable to SHA-256/512)
- very fast for short message hashes

All instances of TWISTER make use of the wide-pipe design idea presented in [103]. Its internal state is 512 bits when the output size is smaller then 256 bits, else it is 1024 bits for other cases.

5.2.2 The Design Principles of TWISTER

In this section we explain our design targets and describe why TWISTER was designed as it is.

Security The security of TWISTER is based on well studied concepts known from AES [40]. The wide trails strategy allows us to obtain a maximal diffusion inside each column of the state matrix. Since the message input is orthogonal with the diffusion of the state, we allow the adversary a small control over the state. Introducing local feed-forward as well as `blank rounds` (rounds with no message input) further reduce the influence of an adversary on the state.

Evolutionary During the last decade many hash functions which use a lot of different concepts were broken.

The TWISTER design is in some way evolutionary since we have learned from the AES process in many ways. The well studied and analyzed block ciphers that were in the final round of the competition lead to some well established design principles offering a high level of cryptographic knowledge. Rijndael [39] therefore can be seen as one of the most studied block ciphers during this process and also in the time after. Its concepts of simple byte-wise operations SubBytes, ShiftRows and MixColumns are well analyzed and it turns out that their combination can offer high speed. We adopt some of these concepts for TWISTER and we also learn from recent hash function breaks.

Simplicity and Analyzability A hash function in general should be easy to analyze and without having any trapdoors inside. The easier it is to analyze, the more one can be sure to avoid trivial flaws in its design. We therefore only use very simple components which form our `Mini-Round`. These components are well studied and well known but combining these components to obtain a good hash function is new. Our very simple design and clear structure of TWISTER makes cryptanalysis easy and serves the purpose that there are no simple attacks which cannot be found due to a complex and unreadable algorithm. The security level of TWISTER can be proven for the inner components which is more worth than just a security claim. Using the concept of wide trail, leads to a very fast diffusing after just two (64-bit) input blocks.

Portability and Scalability The main design criterion of TWISTER is its use to a wide range of applications. Due to its byte-wise operations it scales perfectly on 8-, 16-, 32- and 64-bit platforms. TWISTER can be implemented on smart cards with small 8-bit processors very efficiently. The portability is enhanced by its low memory requirements, which makes TWISTER suitable even for low end platforms valuable. For 32 and 64-bit multi-core we offer a high speed parallel mode of operation which scales to reach the optimal level of processor usage.

Speed TWISTER has not only a high security level, it is also very fast. We offer an optimized version for 32-bit and 64-bit environments. Compared with members of the

SHA-2 family TWISTER is at least as fast on 64-bit platforms. This makes a parallel implementation of TWISTER very fast and efficient.

5.2.3 The TWISTER Hash Function Family

In this section we describe the TWISTER family of hash functions [50, 51, 53]. We start with a description of the general design strategy. The design is based on a blockcipher that is used in the Davies-Meyer (DM) mode of operation [29]. TWISTER is byte-oriented and operates on a square state matrix. The building block of the blockcipher is called `Mini-Round`. It takes a sub portion of the message and injects it into the state S whereas S is an $N \times N$ byte-matrix, $N \in \mathbb{N}$. After two `Mini-Round`s, the state is guaranteed to have a full diffusion. Also, two subsequent iterations of `Mini-Round` can be proven to be collision free. See Section 5.2.4 for a detailed discussion of the security properties.

After processing the padded message (*i.e.*, the message is completely absorbed by the state S), a finalization step and the output follows. This technique is inspired by the design ideas of the sponge function [9].

Note that the description of the TWISTER components is slightly more general as they need to be. The reason for this is to give a better understanding of the design principles. The following notations are used:

$S = (S_{i,j})_{1 \le i,j \le N}$	internal state matrix
$S_{(\rightarrow i)}$	the i-th row vector of S
$S_{(\downarrow i)}$	the i-th column vector of S
$C = (C_{i,j})_{1 \le i,j \le N}$	internal checksum matrix
N	number of rows and columns of the internal state matrix
msg_{size}	size of unpadded message (in bits)
R_{msg}	number of N-byte blocks processed per compression function call
m	function calls needed to absorb the message into the state S
$M = (M_1, \dots, M_m)$	padded message to be handled by the TWISTER hash function
M_{unpad}	unpadded message
out	size of the hash value measured in N-byte blocks, i.e., $out = n/(8 \cdot N)$
n	size of the hash value in bits, $n = out \cdot 8 \cdot N$, where $n \le 8 \cdot N^2$
$H = (H_1, \dots, H_{out})$	number of N-byte blocks of the hash output
ϕ	TwistCounter

The State S

TWISTER operates on a square state matrix $S = (S_{i,j})$, $1 \le i,j \le N$, consisting of N rows and columns, where each cell $S_{i,j}$ represents one byte. For all proposed instances of TWISTER we have $N = 8$, *i.e.,* the internal state matrix is 512 bits.

$$S = \begin{array}{|c|c|c|c|}\hline S_{1,1} & S_{1,2} & \dots & S_{1,N} \\ \hline S_{2,1} & S_{2,2} & \dots & S_{2,N} \\ \hline \vdots & \vdots & \ddots & \vdots \\ \hline S_{N,1} & S_{N,2} & \dots & S_{N,N} \\ \hline \end{array}$$

We use the following two notations concerning the state $S_{(i\rightarrow)} := (S_{i,1}, \dots, S_{i,N})$ denotes the i-th row vector and $S_{(j\downarrow)} := (S_{1,j}, \dots, S_{N,j})$ the j-th column vector.

Checksum C

The checksum enlarges the state of TWISTER-384 and TWISTER-512 to address our wide-pipe design [103] decision.

The checksum is, similar to the state matrix S, a square matrix $C = (C_{i,j})$, $1 \leq i, j \leq N$, consisting out of N rows and columns, where each cell $C_{i,j}$ represents one byte. Recall that $N = 8$, and:

$$C = \begin{array}{|c|c|c|c|}\hline C_{1,1} & C_{1,2} & \dots & C_{1,N} \\ \hline C_{2,1} & C_{2,2} & \dots & C_{2,N} \\ \hline \vdots & \vdots & \ddots & \vdots \\ \hline C_{N,1} & C_{N,2} & \dots & C_{N,N} \\ \hline \end{array}$$

TwistCounter ϕ

The TwistCounter ϕ is an unsigned 64-bit integer added to the state within a `Mini-Round` to prevent slide attacks. After that the TwistCounter is decreased.

Processed message counter `hs_ProcessedMsgCounter` Processed message length `hs_ProcessedMsgCounter` is a 64-bit integer. The main purpose of the counter is to compute the length of the hashed message.

The Compression Function

The compression function of TWISTER consists of building blocks called `Mini-Rounds` which are grouped into `Maxi-Rounds`. Each `Mini-Round` is a combination of well studied primitives, which are easy to analyze and fast to implement in software and hardware. The instances of the TWISTER hash function family differ only in their number of `Maxi-Rounds` and the checksum matrix.

The compression function takes a 512-bit message block and processes it into the internal state matrix S. As a `Maxi-Round` only indicates the position of the local feed-forward XOR-operation we normally only discuss a compression function as a set of `Mini-Rounds`.

Mini-Round

The `Mini-Round` is the underlying building-block of the TWISTER hash function. Its main purpose is injecting the message (Message-Injection) and to enhance the diffusion in the state matrix *S*. Each `Mini-Round` processes 64-bit of message data. If a `Mini-Round` is processed without a message input, i.e., an all zero message input, we call it a `Blank-Round`. The `Mini-Round` is visualized in Figure 5.6.

Twister can handle at most 2^{64} `Mini-Rounds`. This limitation is caused by the AddTwistCounter operation where a 64-bit counter is added. We define the maximal number of message blocks for TWISTER-224/256 to be $2^{64}/9 \approx 2^{60.8}$ and for TWISTER-384/512 to be $2^{64}/10 \approx 2^{60.7}$. For longer messages the `Mini-Round` will not have a unique constant XOR'ed into the state. This might weaken the construction for slide attacks and thus we limit the length of the messages. A `Mini-Round` consists of the

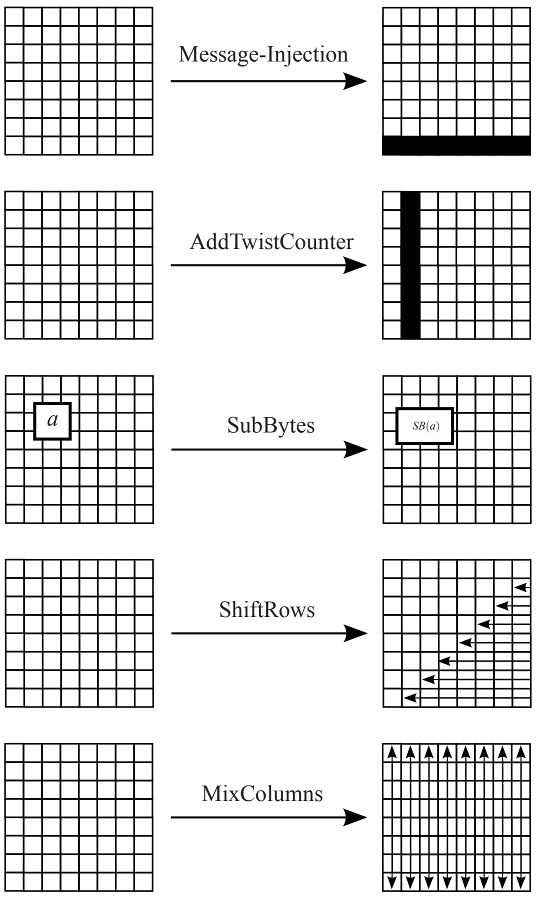


Figure 5.6: A `Mini-Round`

following operations:

- Message Injection injects a 64-bit message block into the last row of the state matrix,
- SubBytes applies a non-linear S-box on each byte of the state matrix in parallel,
- AddTwistCounter XORs a round dependent counter into the second column of the state matrix,
- ShiftRows rotates the i'th row *i* by *i* positions to the left,
- MixColumns applies a linear diffusion on each column of the state matrix in parallel.

Message injection A 64-bit message block m is inserted via XOR into the last row. By using the notation $m = (m[N], \dots, m[1])$ whereas the length of $m[i]$ is one byte, and let

$$S_{(\rightarrow j)} \oplus m = (S_{N,j} \oplus m[N], \dots, S_{1,j} \oplus m[1])$$

we define the message injection process by

$$S_{(\rightarrow N)} := S_{(\rightarrow N)} \oplus m.$$

Add TwistCounter The TwistCounter ϕ is an unsigned 64-bit integer. The initial state is the maximal value (`0xFFFF FFFF FFFF FFFF`). By using the notation $\phi = (\phi[N], \dots, \phi[1])$, where the length of $\phi[i]$ is one byte, and $\phi[1]$ is the most significant byte of ϕ. The counter is byte-wise XORed into the second column of the state S:

$$S_{2,\downarrow} \oplus \phi = (S_{2,N} \oplus \phi[N], \dots, S_{2,1} \oplus \phi[1])$$

and set

$$S_{2,\downarrow}, \phi := S_{2,\downarrow} \oplus \phi \text{ and } \phi := \phi - 1.$$

SubBytes SubBytes uses a bijective function

$$\text{S-box} : \{0,1\}^8 \longrightarrow \{0,1\}^8.$$

This S-box is applied in parallel to each byte of the state matrix, which forms the SubBytes operation. It is highly non-linear. A discussion on how to obtain such cryptographically strong S-boxes (for 8x8 S-boxes) can be found in [154]. TWISTER uses the well known and studied AES S-box. The S-box is given in Appendix A.

We define the SubBytes operation by

$$S_{i,j} := SB(S_{i,j}) \quad i,j \in \{1,2,\dots,N\}.$$

ShiftRows ShiftRows is a cyclic left shift similar to the ShiftRows operation of AES. It rotates the j'th row by $j-1$ bytes to the left.

We define the ShiftRows operation by

$$S_{(i,(j-i+1 \bmod N)+1)} := S_{(i,j)} \quad i,j \in \{1,2,\dots,N\}.$$

MixColumns The MixColumns step is a permutation operation on the state. It applies an $N \times N$-MDS matrix A (a maximum distance separable matrix as defined below) to each column, i.e., performs the operation

$$S_{\downarrow} := A \cdot S_{(j\downarrow)} \quad j \in \{1,2,\dots,N\}$$

Definition 2 *An [n,k,d] code with generator matrix*

$$G = [I_{k\times k}\, A_{k\times(n-k)}]$$

is an MDS code if every square submatrix of A is nonsingular. The matrix A is called an MDS-matrix.

Our chosen MDS matrix is cyclic, i.e., its i-th row can be obtained by a cyclic right rotation of (02 01 01 05 07 08 06 01) by i entries. It has a branch number of 9, meaning that if two 8-byte input vectors differ in $1 \leq k \leq 8$ bytes, the outputs of MixColumns differ in at least $9-k$ bytes. The 8×8-MDS matrix used in all proposed instances of TWISTER is:

$$MDS = \begin{pmatrix} 02 & 01 & 01 & 05 & 07 & 08 & 06 & 01 \\ 01 & 02 & 01 & 01 & 05 & 07 & 08 & 06 \\ 06 & 01 & 02 & 01 & 01 & 05 & 07 & 08 \\ 08 & 06 & 01 & 02 & 01 & 01 & 05 & 07 \\ 07 & 08 & 06 & 01 & 02 & 01 & 01 & 05 \\ 05 & 07 & 08 & 06 & 01 & 02 & 01 & 01 \\ 01 & 05 & 07 & 08 & 06 & 01 & 02 & 01 \\ 01 & 01 & 05 & 07 & 08 & 06 & 01 & 02 \end{pmatrix}$$

All of the byte entries are considered to be elements of $\mathbb{F}_{2^8}$. An element $\sum_{i=0}^{7} a_i x^i \in \mathbb{F}_{2^8}$ is represented by $\sum_{i=0}^{7} a_i 2^i$. The reduction polynomial $m(x)$ of $\mathbb{F}_{2^8}$ is defined as

$$m(x) = x^8 + x^6 + x^3 + x^2 + 1. \tag{5.2}$$

Note that this field is different from the one used in the AES for SubBytes. Some properties of MDS matrices/codes can be found in [105].

Maxi-Round

A `Maxi-Round` contains three `Mini-Rounds` in the case of TWISTER-224/256 and three or four in the case of TWISTER-384/512. In TWISTER-224/256 the last

`Mini-Rounds` in the third `Maxi-Round` is a `Blank-Round`. In TWISTER-384/512 the second `Mini-Rounds` of the second `Maxi-Round` and the last `Mini-Round` of the last `Maxi-Round` is blank, i.e., `Blank-Round`.

The checksum matrix is used only in TWISTER-384/512 and is updated before each `Mini-Round` takes place. We define a checksum update operation as

$$\forall i: \quad C_{(i,\downarrow)} = C_{(i,\downarrow)} \oplus C_{(i+1,\downarrow)} \boxplus S_{(i,\downarrow)}.$$

The state matrix is added to the checksum column-wise.

After each `Maxi-Round` a feed-forward operation is used where the state before a `Maxi-Round` S^k is fed-forward with the state after the current `Maxi-Round` S^{k+1}. We defined the feed-forward operation as

$$S_{(i,j)} := S^k_{(i,j)} \oplus S^{k+1}_{(i,j)} \quad \forall i,j.$$

Figure 5.7 illustrates a `Maxi-Round`.

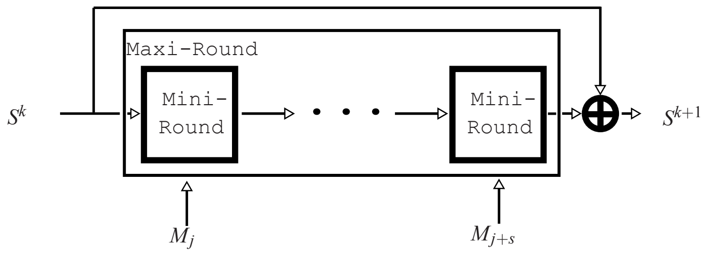


Figure 5.7: A `Maxi-Round`

Compression Function of TWISTER-224 and TWISTER-256

The compression functions of TWISTER-224 and TWISTER-256 consist of three `Maxi-Rounds`. Each `Maxi-Round` contains three `Mini-Rounds`. Figure 5.8 illustrates the compression function.

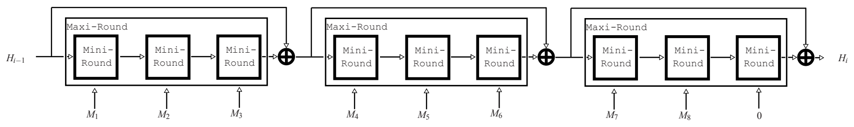


Figure 5.8: The compression function of TWISTER-224 and TWISTER-256

Compression Function of TWISTER-384 and TWISTER-512

The TWISTER-384 and TWISTER-512 compression functions consist of three `Maxi-Rounds`. Where the first and the second `Maxi-Round` contain three `Mini-Rounds` and the third one is built out of four `Mini-Rounds`. Figure 5.9 illustrates the compression function. The second and third `Maxi-Round` contains also a `Blank-Round`, to strength the instance against collision attacks. As we explained before, the checksum C is updated with the internal state before a message injection takes place, i.e., three times in the first `Maxi-Round`, twice in the second, and 3 times in the third. After the compression function call for the last message word, the checksum is regarded as 8 message blocks (i.e., one block for each column) and is inserted into the internal state in the same way as an original message block but without updating the checksum.

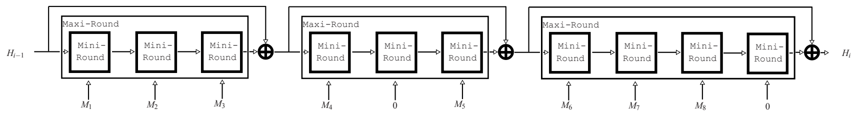


Figure 5.9: The compression function of TWISTER-384 and TWISTER-512

Post-processing

Post-processing takes place after the last 'full' message block (i.e., message block length being divisible by 512-bits) is completely absorbed in the state and, for TWISTER-384/512, also in the checksum matrix. This post-processing consists of three steps:

1. *Padding* the message M and processing the last message block into the state matrix (and checksum matrix if appropriate)

2. TWISTER *state finalization* (processing the message length and the checksum – if appropriate – into the state matrix)

3. *Message digest computation* (performing the `Output-Round`)

Padding The purpose of this padding is to ensure that the padded message has a length which is a multiple of 512 bits. Suppose that the length of the message M is l. Then the message can be divided into $\lceil l/512 \rceil$ 512-bit message blocks $m_1, \ldots, m_{f-1}$ and one final message block m_f of length between 0 and 511 bits. The 512-bit message blocks $m_1, \ldots, m_{f-1}$ are processed by the compression function one by one. The final message block m_f is *always* padded by appending an $'1'$ to the end (of the message) followed by $k = 511 - (l \bmod 512)$ zero bits. Afterwards, the length of m_f is exactly 512-bits and it can processed by the compression function like all other message blocks before. If the last message block is a complete 512-bit block an additional block is attached containing a one followed by 511 zeros.

State Finalization After the padding procedure, the message length, stored in a 64-bit unsigned integer, is injected like a message by XORing it byte by byte to the last row of the state S. This is done after processing the message blocks.

Note that the following steps depend on the TWISTER instantiation. For TWISTER-384/512, the checksum is processed as follows. The checksum is transformed into a 64-byte message block m where $m = m[f], \dots, m[1]$ column by column.

$$m[i] = C_{(\lfloor (i-1)/N \rfloor, i-1 \, mod \, N)} \qquad \forall i = 1, \dots, N^2$$

Each $m[i]$ enters the internal state being processed by a `Mini-Round`, where $m[i]$ is treated as a message block.

Finally, the state is updated via eight `Blank-Rounds`, while the twist counter is not used anymore. This is done in all instances of TWISTER.

Message Digest Computation The message digest is computed using a so called `Output-Round`. The `Output-Round` of TWISTER contains a global feed-forward as well as some `Mini-Rounds` depending on the size of the hash output. First, a `Mini-Round` is applied to the state S^{i-1}, then the resulting state is XORed with S^{i-1}. Then, another `Mini-Round` is applied which gives the state S^i. Let S^i be the final state after the last compression function call (for TWISTER-384/512 this is after the processing of the check sum). A 64-bit output stream out_i is then obtained by XOR-ing the *first column* of S^i with the first column of S^{i-1}. This procedure takes place until the needed amount of output bits is obtained. The last output stream can be varied between 32 bits and 64 bits by taking only the first half (lowest order first) of out_i. Figure 5.10 gives a high level description overview of an `Output-Round`.

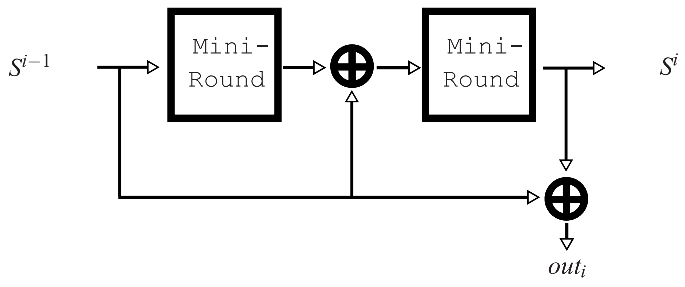


Figure 5.10: An `Output-Round`

Randomized Hashing

TWISTER supports randomized hashing in the following way. One can choose a so called salt value of size at most 64 bits. If the salt is zero then the usual hash computation is performed. For salt values which are not equal to zero and shorter than 64 bits as many zero's are padded to form a 64-bit block. The salt s is introduced in the hash computation as a first input block followed by three `Blank-Round`s. This protects TWISTER against any chosen salt attacks.

5.2.4 Security

Generic Attacks

In this section we describe a few generic attacks on hash functions and show that they do not apply to TWISTER.

Length-Extension Attacks In a length extension attack, given a hash value $H(M)$, the adversary can compute the hash value $H(M||X||Y)$ for any non-empty suffix Y, if the length of unknown message M is known as well as the padding X of M. We protect TWISTER against this kind of attack, since the adversary has no direct access to the internal state after the last message input takes place. By knowing only the hash value an adversary cannot easily recover the internal state S, since the `final()` function outputs only a small fraction of the internal state.

Multi-Collision Attacks Joux's mullti-collision attack [82] works as follows. In a multi-collision $x \geq 2$ messages hash to the same value. Let H be a MERKLE-DAMGÅRD hash function that computes an n-bit hash value. Joux [82] shows that finding a set of 2^k messages all colliding to the same hash value (a 2^k-multi-collision) is as easy as finding k single collisions for H's compression function. The adversary starts by finding a single collision in the compression function then, he iteratively finds an additional collision starting from the previously colliding chaining value. We want to find messages blocks M_i and $M_i' \neq M_i$ such that $C(h_{i-1}, M_i) = C(h_{i-1}, M_i')$, where $C(\cdot)$ represents the compression function and h_i represents the chaining value. It is then possible to construct 2^k messages with the same hash value by choosing for each block i either the message block M_i or M_i'. This attack can find 2^k-way multi-collisions with a complexity of $k \cdot 2^{n/2}$.

TWISTER fully resists the multi-collision attack due to the wide-pipe design [103], as the internal state is at least of size $2n$, and thus every internal collision takes 2^n time.

Herding Attacks The herding attack [85] is a chosen prefix target preimage attack on MERKLE-DAMGÅRD hash functions, i.e., the adversary commits to a digest value h and is then presented with a challenge prefix P. Now, the adversary computes a suffix S, such that $H(P|| S) = h$.

The preprocessing of the herding attack works as follows. Let H be a MERKLE-DAMGÅRD hash function that computes an n-bit chaining value. The adversary takes 2^k chaining values which are fixed or randomly chosen. Then he chooses $O(2^{n/2-k/2})$ message blocks and computes the output of the compression function for each chaining value and each block. It is expected that for each chaining value there exists another chaining value, such that both collide to the same value. The adversary stores the message block that leads to such a collision in a table and repeats this process again with the newly found chaining values. The preprocessing time is $2^{k/2+n/2+2}$ applications of the compression function.

In the online phase, after receiving the challenge P, the adversary exhaustively searches for message blocks X such $P||X$, for some message P forced by the adversary, collides with on of the starting chaining values. Having found such message block X, the adversary can construct a sequence of message blocks Q, with $S = X||Q$ such that the hash value $H(P||S) = h$. This step requires a negligible amount of work, and the resulting suffix Y will be $k+1$ blocks long. The online part of the attack requires trying $O(2^{n-k})$ possibilities.

All of our proposed instances of TWISTER resist this kind of attack due to the large internal state.

Long second pre-image Attacks Dean [42] showed for MERKLE-DAMGÅRD hash functions that easily found fix-points in the compression function can be used for a second-pre-image attack against long messages in time $O(n \cdot 2^{n/2})$ and memory $O(n \cdot 2^{n/2})$. Kelsey and Schneier [88] extended this result and provided an attack to find a second pre-image on a MERKLE-DAMGÅRD construction much faster than the expected workload of 2^n even when it is hard to find fixed points in the compression function. These attacks use messages of variable sizes that collide internally given a fixed IV. These *expandable messages* can be found either using fixed points as in [42] or as a series of internal collisions between a one-block message and an α-block message for varying values of α [88]. The *expandable messages* are used for producing messages of varying length, which all result in the same internal state. It takes $2^{n/2+1}$ work to discover 2^k-*expandable message* blocks using the method of [88] and 2^{n-k+1} work to build a full second preimage. Thus, the total complexity is about $k \cdot 2^{n/2+1} + 2^{n-k+1}$ compression function calls. The complexity of this attack to find a second pre-image for a k-message block depends on the complexity of finding *expandable messages* and the message length k.

Andreeva et al. [2] showed that a combination of the attacks from [42, 85, 88] can be mounted to dithered hash functions. This gives the adversary a greater control on the second pre-images, since he can choose most parts of the second message in an arbitrary way. The attack is based on the diamond like structure, which is build in the herding attack [85]. If the diamond is a 2^l-multicollison, one can obtain a second pre-image of a message of length 2^k blocks with $2^{n/2+l/2+2}+2^{n-l}+2^{n-k}$ compression function computations. The attack works as follows. A diamond of size l is a multicollision that has the shape of a complete converging binary tree of depth l, with 2^l leaves. The nodes of this tree are labeled by chaining values over n bits, and its edges are labeled by message blocks over m bits, which map between the chaining value at the two ends of the edge by the compression function. From any one of the 2^l leaves, there is a path labeled by l message blocks that leads to the same target value h labeling the root of the tree. Now, let M be a target message of length 2^k blocks. The main idea of this attack is that connecting a message to a colliding tree can be done in less than 2^n work and connecting the root of the tree to one of the 2^k chaining values encountered during the computation of $H(M)$ takes only 2^{n-k} compression function calls.

To make TWISTER secure against such attacks we incorporate several security mechanisms. The first countermeasure is a large internal state. We have to avoid small cycles in the dithering sequence. Therefore, the counter should be of the same size as the maximal message block length is. Thus, each round function is unique due to the different round constant being used. As the twist counter never repeats, it does not allow finding *expandable messages*. We note that the twist counter as well as the wide pipe design prevent the earlier types of second preimage attacks.

Slide Attacks Slide attacks are common in block cipher cryptanalysis, but they are also applicable to hash functions as shown in Section 5.1. TWISTER is resistant against this type of attack since we include a counter ϕ. It guarantees that a sliding property of two or more consecutive states is impossible due to the different twist counters.

Differential Attacks The idea is to exploit a high-probability collision producing differential over some component of the hash function. This means that the adversary searches for such differentials under the form of a ”perturb-and-correct” strategy[3] for the hash function. In the design of the TWISTER framework, we applied the following countermeasures against differential attacks:

- *High-Speed-Diffusion.* The strong diffusion capabilities of the `Mini-Round` in combination with the non-linear S-box make it hard to find good differential characteristics.
- *Nested feed-forward.* The internal feed-forward operations aim at strengthening the function against differentials, since it is harder to cancel a difference out once it has entered the internal state.

[3]Or any other strategy, as long as it is high probably and produces a collision.

- *Internal wide-pipe.* this makes internal collisions harder to find, and the `Output-Round`s make the differences much harder to predict in the hash value.
- Using different operators (e.g., $\oplus$, $\boxplus$) highly complicates the computation of good differential paths, which makes TWISTER stronger against some classes of attacks.

A Collision Attack Method The first step in finding a collision for any instance of the TWISTER hash family is to analyze `Mini-Round`. We claim that an internal collision needs at least three applications of a `Mini-Round`, i.e., three input blocks. To find a collision we need the differential properties of MixColumns which are shown in Table 5.1.

D_I \ D_O	0	1	2	3	4	5	6	7	8
0	0	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
1	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	0
2	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	-8	0
3	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	-16	-8	0
4	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	-24	-16	-8	0
5	$-\infty$	$-\infty$	$-\infty$	$-\infty$	-32	-24	-16	-8	0
6	$-\infty$	$-\infty$	$-\infty$	-40	-32	-24	-16	-8	0
7	$-\infty$	$-\infty$	-48	-40	-32	-24	-16	-8	0
8	$-\infty$	-56	-48	-40	-32	-24	-16	-8	0

Table 5.1: The approximate probability that two 8-byte input words with D_I different bytes in predefined positions maps to two 8-byte output words with D_O different bytes in predefined positions by the *MixColumns* operation. The probabilities are in base-2 logarithms. Column Properties of the state matrix after multiplication with the MDS matrix.

The following lemma gives the number of active bytes after two `Mini-Rounds` in the worst case starting from the all zero internal state:

Lemma 1 *Starting from the all zero internal state and inserting a one byte difference in the first input block, then at least* 49 *bytes are active after the second input round.*

Proof 1 *To prove the lemma we assume that it is possible to generate an internal state after the second message input which contains less than* 49 *active bytes.* 49 *active bytes can be constructed by 7 columns with 7 active bytes and one column with zero active bytes or* 8 *columns with at least one column with strictly less than seven active bytes. We start with the first case. Assume that we reduce one column by one active byte, which leaves* 48 *active bytes in total. Thus, one column with 6 active bytes must be a column containing 3 to 8 bytes after MC^{-1}. This means that after SR^{-1} there must be at least 3 different rows where at least 3 active bytes have a different index. This is impossible since there can only be two different indices for the active bytes. After* *the first round the first column contains 8 active bytes. The following message input can only introduce active bytes in the last row of the state matrix. This contradicts our assumption and so at least* 49 *bytes are active after the second input round. Now we analyze the second case, where* 8 *columns with less than seven active bytes are less than* 49 *active bytes after the second round in total. If a column after the second round contains at most six active bytes it contains at least three active bytes after* MC^{-1}. *There are at least three different columns and three different rows after* SR^{-1} *which contain one active byte. This state cannot be constructed from the all zero state after one round.*

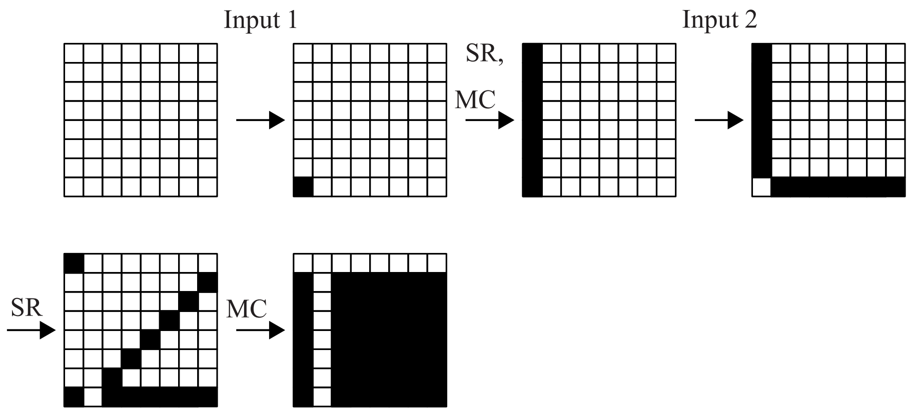


Figure 5.11: Minimal number of active bytes after two `Mini-Rounds`

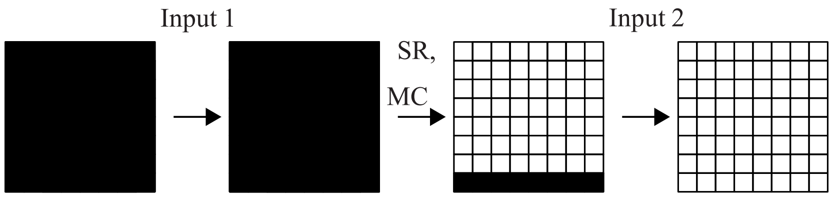
■

The following lemma holds only for TWISTER-224 and TWISTER-256.

Lemma 2 *i) Starting from the all difference internal state at least two input rounds are needed to obtain the all zero internal state. ii) Any non-zero differential starting from the all zero internal state and ending in the all zero internal state needs at least three input rounds. iii) Such a three round differential with the desired difference has probability* 2^{-512} *if all the bytes of the first input block are non-zero. Note that this analysis applies for random messages.*

Proof 2 *i) Assume that we can obtain the all zero internal state after one input round. Thus, the internal state after MixColumns must be all zero. This is also true for the internal state before MixColumns and ShiftRows. Since we have started from the all different internal state the message input must cancel all active bytes out. This is not possible since it takes place only in the last row of the internal state. Thus, we cannot have the all zero internal state after one round.*

By starting from the all difference internal state and inserting any difference which does not cancel any state byte difference out we apply a first `Mini-Round`*. According to Table 5.1 MixColumns may generate a non-zero difference in the last byte of a column and a zero difference in the remaining bytes with probability* 2^{-56}*. This occurs for eight columns with probability* 2^{-448}*. The input block in of the second round will cancel the difference in the first row out. This happens for a randomly chosen input block with probability* 2^{-64}*. Thus, the all zero internal state occurs after at least two input rounds with probability* 2^{-512}*.*



ii) Assume that we can construct the all zero internal state from the all zero internal state in two rounds. Therefore, the internal state after the second round is all zero. This means that the internal state before the ShiftRows and MixColumns of the second round is all zero. Thus, the message input in the second round must cancel out a difference in the last row of the internal state. In other word there must be a difference only in some bytes of the last row. Thus, there are 8 rows, which have at least one active byte before the ShiftRows and MixColumns of the first round. The message injection of the first round can only insert a difference into the last row, which contradicts the assumption.

iii) follows from i) and ii).

■

Note, that finding internal collisions can be improved with message modification techniques. These lemmas offer upper bounds on the differential probabilities of collision-producing differential characteristics.

Pseudo Collisions on `Mini-Rounds`

It is easy to construct a pseudo-collision of a `Mini-Round` by the following scenario. Choosing two states IV_1 and $IV_2 \neq IV_1$ such that for a message block M_i

$$\texttt{Mini-Round}(IV_1, M_i) = \texttt{Mini-Round}(IV_2, M_i),$$

holds. Since a `Mini-Round` is a bijection, it is invertible. Such an attack cannot lead to a collision of the `Maxi-Round` due to the local feedforward of a `Maxi-Round`. Thus, although pseudo collisions can be found easily for a `Mini-Round` they do not lead to a collision of a `Maxi-Round`.

5.2.5 On (Second) Pre-image Resistance

The `Mini-Round`s are permutations and are easily invertible. This is beneficial for collision-resistance but might lead to problems with (second) pre-image resistance.[4] We solve this problem by applying feed-forward XORs in the internal state. So any adversary that tries to mount a pre-image attack has to recover the state from the hash value. For this, the adversary has to guess one bit for every hash output bit (this is due to the `Output`-function). Let $H = (h_1, \ldots, h_{out \cdot N})$ be the hash output for hashing a message M, $|h_i| = 1$ byte. Assume that an adversary tries to mount a (second) pre-image attack. In order to invert a `Mini-Round`, the adversary has to recover the entire internal state S. As the adversary has only 1/N-th of the internal state $S_{(1,\downarrow)} \oplus T_{(N,\downarrow)}$ he has to guess $T_{(N,\downarrow)}$ in order to recover one column $S_{(1,\downarrow)}$ of S. Furthermore, he has to guess all of the other columns to be able to invert a round. The complexity for guessing essentially the whole matrix is $O(2^{N \cdot 8})$. Due to the feed forward, a fix point in a `Mini-Round` will not lead to a fix point in a `Maxi-Round` (unless it is the all-zero value). Pre-images as well as (second) pre-images can be found if an adversary can easily compute fix points in at least one `Maxi-Round`. Then, he can combine different fixed points to obtain a message for a certain hash value.

5.2.6 A Collision attack on TWISTER-512

In this section we describe a collision attack on TWISTER-512 which was proposed by Mendel et al. [107]. This attack combines a semi-free-start collision in the compression function using the rebound attack [108] with Wagner's generalized birthday attack [145]. The attack works only against TWISTER-512.

A Semi-free-start Collision Attack on the Compression Function of TWISTER

A semi-free-start collision in the compression function of TWISTER can be found in 2^8 compression function evaluations. This attack works for all instances of TWISTER. The attack can be summarized as follows. Let r_1, r_2, and r_3 denote three `Mini-Rounds` in a `Maxi-Round` and let S_i be the state after the 64-bit message word M_i is proceeded in r_i. The initial state is denoted by S_0. Note that this attack targets the three `Mini-Rounds` in a `Maxi-Round` of each block independently.

Let an input difference ΔX and an output difference ΔY, where $\Delta X = x_1 \oplus x_2$, $\Delta Y = y_1 \oplus y_2 = S(x_1) \oplus S(x_2)$ and $S(\cdot)$ represents the S-box of TWISTER. A good S-box as the one proposed for the AES, which is also used in TWISTER, has a near uniform distribution of differentials which do exist, i.e., each row of the difference distribution table contains

[4]Note that not all second pre-image attacks rely on the ability to invert the `Mini-Rounds`.

very low values which indicate the occurrence of a one-round differential characteristic. In the case of the AES S-box each row, besides the zero one, contains only zeros, twos and fours. Counting the number of zeros in the difference distribution table of the AES S-box, there are 33150 instances where the input difference cannot become a certain output difference. In the remaining entries there are 32130 entries of twos, 255 entries of fours and one entry of 256, which mark that certain input differences lead to certain output differences. The probability that a fixed input, output difference $(\Delta X, \Delta Y)$ has non-zero probability can be expressed as $\Pr[(\Delta X \to \Delta Y) \neq 0] \sim 1/2$. We note that this is not the differential probability, but it only expresses that an input difference can be transformed into a certain output difference with a probability greater than zero.

The rebound attack [108] can be split into two phases. First, an inbound phase where the differences propagate backward and forward through the MixColumns operation with a probability of one. Second, the outbound phase that tries to find matches for the resulting input and output differences of the SubBytes operation of r_2 and propagates outwards.

1. Let $S_0, S_1, S_1', S_2, S_2'', S_2'''$ and S_3 be internal states of TWISTER as shown in Figure 5.12. Starting with 8 active bytes in S_1' and S_2 that are injected into r_1 and r_3 by M_1 and M_3. Then computing from S_2 backward through MC^{-1} and SR^{-1} to obtain the state S_2''. Compute forward from S_1' through ATC, SB, SR, MC and obtain S_2'''. The states S_2'' and S_2''' contain 64 active bytes.

2. In the second Step of the attack the adversary has to find a match for the input and output differences of the 64 S-boxes in r_2. As explained above a fixed S-box input/output difference may be satisfied with probability of about $1/2$. Starting with a random difference for the last byte in the first column of S_1 he can compute the corresponding column of S_2''. The adversary finds a differential characteristic which has a non-zero probability with probability $(1/2)^{-1} = 2^{-8}$. Once a valid differential is found, he continues with the other columns. This takes about 2^8 compression function calls.

There are 2^{64} possible states for S_2'', so the adversary can choose the one to get the correct input/output difference for SubBytes in the 64 bytes of the internal state. Each of these states can be computed in forward and backward direction and lead to a semi-free-start collision of a `Maxi-Round`. Note that the value of M_2 can be freely chosen in the attack. Once a semi-free collision in a `Maxi-Round` is found this transforms into a collision for the compression function of TWISTER. Figure 5.12 outlines the attack.

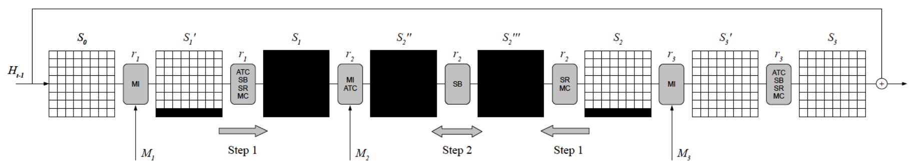


Figure 5.12: A semi-free-start collision for the TWISTER compression function. Source: [107]

A Collision Attack on TWISTER-512

The semi-free-start collision attack presented in the previous section can be transformed into a collision attack on TWISTER-512. To find a collision in TWISTER-512 one has to find a collision in the internal chaining value as well as in the value of the checksum. This can be done by constructing multi collisions [82] and applying Wagner's generalized birthday attack [145]. Mendel et al. [107] showed that one can construct a collision in the compression function of TWISTER-512 with complexity of 2^{223} compression function calls. They first compute a large number of semi-free start collisions for the last `Maxi-Round`. After that computation, they exploit the degrees of freedom in the last row of the internal state to find a collision in the compression function. So, the adversary can construct a 2^{256}-collision with a complexity of about $256 \cdot 2^{223} \approx 2^{231}$ compression function calls. To apply this attack, the adversary needs memory of about 2^9 to store the 2^{256} collisions. In this way he gets 2^{256} values for the checksum C that all lead to the same chaining value H_{256}. Using 257 message blocks which produces the same value in the checksum the adversary can find a collision in the checksum for TWISTER-512. In other words, 256 message blocks are used to find a multi collision and an additional message block is needed for the padding of TWISTER. Using an ordinary birthday attack this would cost about 2^{256} checksum computations and memory requirements of 2^{256} to find these to colliding messages. Using a memory-less variant of the birthday attack [129] the adversary can find a collision for TWISTER-512 with a complexity of about 2^{231} compression function calls and about 2^{256} checksum computations. Normally, the cost for the computation of the checksum is much smaller than one computation of the compression function. If he assumes that the computation of the checksum takes at most $1/16$ of the computation of the compression function, then he can find a collision in TWISTER-512 with a complexity of 2^{252} compression function calls.

Protecting TWISTER against this Attack We can defend TWISTER-512 against this kind of collision attack by expanding the internal state. Another possibility is to introduce a kind of non-linearity into the checksum. However, both patches decrease the speed dramatically, but this does not really matter for most of the applications of hash functions.

5.2.7 Second Pre-image Attack on TWISTER-512

The second preimage attack on TWISTER-512 of [107] works as follows. Let $m = m_1||m_2||\cdots||m_{513}$ be a message for which the adversary wants to find a second preimage and let $C = C_1\ ||C_2||\cdots||C_8$ be the checksum where C_i is a 64-bit word.

1. Construct a 2^{512} collision for the first 512 message blocks. This step takes about $512\cdot 2^{256} \approx 2^{265}$ compression function calls. In this way he gets 2^{512} values for the checksum which all lead to the same chaining value H_{512}.
2. Choose m_{513} arbitrarily with a correct padding and compute H_{513}.
3. For the last compression function evaluation $H_{514} = f(H_{513}, C)$, he first chooses arbitrary values for $C_1||C_2||\cdots||C_5$, computes the state $S_6^F = H_{513} \oplus S_3 \oplus S_6$. This determines $S_{10} = H_{514} \oplus S_6^F$. The value of H_{514} is already known from the given message m as it is.
4. For all the 2^{64} possibilities of C_8 the adversary computes backward from S_{10} to the injection of C_7 and stores the 2^{64} candidates for the state S_7' which results after injecting C_7 into S_7 in a list L.
5. For all 2^{64} choices of C_6 the adversary computes forward from S_6 to the injection of C_7 and checks for match of S_7' in the list L. The adversary only needs to match 448 out of 512 bits, since he can still choose C_7. The adversary gets 2^{128} pairs in total. This step succeeds with probability $2^{-448+128} = 2^{-320}$. Repeating Steps 3 to 5 about 2^{320} times leads to a match. The overall complexity is then bounded by $2^{320+64} = 2^{384}$ compression function calls.
6. After finding a second preimage in the internal chaining value, he still has to ensure that the value of the checksum is correct. Using the fact that the checksum is invertible he can use all 2^{512} values for the checksum which all lead to the same chaining values H_{512}, H_{513} and H_{514}. Using a meet-in-the-middle attack, he constructs the values in the checksum that are needed. The time complexity of this step is about 2^{257} compression function calls and the memory requirement is of 2^{256}. The memory requirement can be reduced significantly by using a memory-less variant of the meet-in-the-middle attack.

Thus, the adversary can construct a second preimage for TWISTER-512 with a time complexity of about 2^{384} compression function calls and memory of about 2^{64}. The attack needs a message of at least 513 message blocks to succeed. This attack cannot be adopted to a preimage attack on TWISTER-512 due to the output transformation.

Protecting TWISTER **against this Attack** TWISTER-512 can be tweaked such that this second preimage attack cannot be applied. Enlarging the internal state would strengthen TWISTER-512 against (second) pre-image attacks as well as against collision attacks. Also the checksum can be designed such that it is not invertible. This can be done by applying one-way functions, e.g., feedforward. The non-linear checksum makes it harder to find single block collisions in the internal state as well as in the checksum value. This is necessary to form a second message, which hashes to the same value as the first one.

5.2.8 Pre-image Attack on TWISTER-512

In this section we discuss the preimage attack on TWISTER-512 suggested in [107]. To find a preimage for TWISTER-512 the adversary has to invert the output transformation. Once this is done he can apply the second preimage attack from the previous section. The preimage of $h = out_1||out_2||\cdots||out_8$ found by the adversary consists of 513 message blocks. In this case, the adversary has to find the internal state H_{514} before the output transformation proceeds h. The adversary can find H_{514} such that out_1 is accurate with a complexity of about 2^8 instead of 2^{64} as one could expect. The complexity of inverting the output transformation is therefore 2^{456}. This process is shown in Figure 5.13.

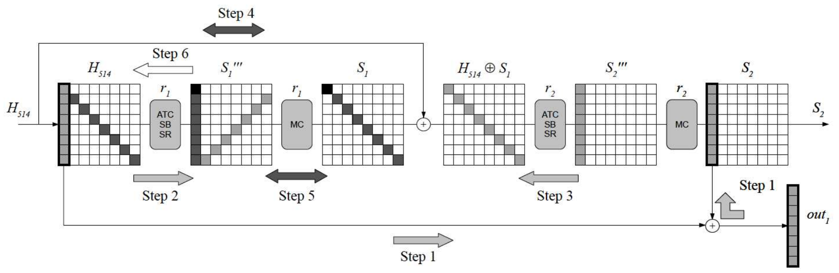


Figure 5.13: The inversion of the first part of the output transformation of TWISTER. [107]

1. The adversary chooses a random value for the first column of H_{514}. He uses out_1 and the first column of H_{514} to compute the first column of S_2.

2. He computes 8 bytes $S_1'''[i][i+(9-i) \bmod 8]$ for $(1 \leq i \leq 8)$ of the state S_1''' using the first column of H_{514}.

3. He computes backwards through r_2 for the first column of S_2 to get the 8 diagonal bytes of $S_1 \oplus H_{514}$.

4. He chooses random values for the 8 diagonal bytes of H_{514}. Note that this determines the first column of S_1'''. Then the adversary computes the 8 diagonal bytes of S_1 from the diagonal bytes of $S_1 \oplus H_{514}$ and H_{514} using the feedforward.

5. Now, the adversary needs to find a match of S_1''' and S_1 through MixColumns in r_1. Note that the first column of S_1''' is already fixed from Step 4. If the first byte of $S_1[1][1]$ does not match, he needs to go back to Step 1 again. After repeating Steps 1 to 4 about 2^8 times he expects to find a match for $S_1[1][1]$. Once, a match is found, modify columns 2 to 8 of S_1''' such that the remaining 7 bytes match as well.

   a) For each column $i = 2,3,\ldots,8$ he chooses random values for the bytes $S_1'''[i][k]$ with $k \neq 9-i$. Note that the bytes $S_1'''[i][k]$ with $k = 9-i$ are already fixed due to Step 2 of the attack.

   b) Next, he computes the MixColumns operation and checks if the byte $S_1[i][i]$ matches. If not, he repeats the previous step. This step has a complexity of about 2^8 operations.

   Each column can be modified independently in the attack and so this step runs in about $8 \cdot 2^8 \cdot (1/10) \approx 2^{7.6}$ compression function calls.

6. When a match for all columns is found, the adversary computes backwards from S_1''' to determine H_{514}. Note that values fixed in Step 1 and Step 4 do not change anymore.

By repeating the attack 2^{448} times, the adversary can invert the output transformation of TWISTER-512 with a complexity of about $2^{448} \cdot 2^{7.6} \approx 2^{456}$ compression function calls and memory requirement of 2^{10}. Once the output transformation is inverted, i.e., H_{514} is found, he can apply the second preimage attack of the previous section to construct a preimage for TWISTER-512 which consists of 513 message blocks. The overall time complexity of this attack os about $2^{448} + 2^{456} \approx 2^{456}$ compression function calls.

Protecting TWISTER **against this Attack** This preimage attack on TWISTER-512 can be defended against in the same way we discussed for the second preimage attack. Enlarging the internal state will defend against this attack. Another patch on TWISTER-512 could be to strengthen the output transformation by squeezing less than 8 bytes during each output step or add more `Blank-Rounds` in the output process. The third way to enhance the security of TWISTER-512 against pre-image attacks is to enlarge the internal state and use a 16×16 byte matrix, but this would lead to a decrease in the performance in two ways. First, it decreases the speed of the hash computation. Second, it is hard to find a 16×16 MDS-matrix which has low values in most of the bytes. Thus, the performance of the hash computation decreases dramatically due to a lack of good MDS-matrixes of this size. We believe that enlarging the internal state would be a good option since the decrease in speed is not crucial for many applications.

5.2.9 Implementational Aspects and Performance

In this chapter we discuss issues related to the implementation of TWISTER on different platforms. In essence, our techniques for implementing TWISTER rely on the following key sources of information due to the similarity to the AES-round function:

- Optimization techniques given by Daemen and Rijmen in [40].
- The techniques for reducing the number of instructions for an AES implementation [7, 74, 83, 122].

Finite Field Multiplication

In the algorithm of TWISTER there are no multiplications of two variables in $GF(2^8)$, but only the multiplications of a variable with a constant. The latter is easier to implement than the former – especially in the context of hardware and high-speed software implementations.

64-Bit Platforms

All the different steps for the round transformation can be combined in a single set of look-up tables including the ShiftRows operation, allowing very fast implementations on processors with word lengths of 64 bits (or more). We use the following notations (for $1 \leq x,y \leq 8$):

$a_{y,x}$ is the byte of the state matrix at the position (x,y),
$b_{y,x}$ is the byte of the state matrix after the S-box at position (x,y),
$c_{y,x}$ is the byte of the state matrix after SHIFTROWS at position (x,y),
$d_{y,x}$ is the byte of the state matrix after MIXCOLUMNS at position (x,y).

After the MIXCOLUMNS-Operation, we have for $1 \leq j \leq 8$:

$$\begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \\ d_{4,j} \\ d_{5,j} \\ d_{6,j} \\ d_{7,j} \end{bmatrix} = \begin{bmatrix} 02 & 01 & 01 & 05 & 07 & 08 & 06 & 01 \\ 01 & 02 & 01 & 01 & 05 & 07 & 08 & 06 \\ 06 & 01 & 02 & 01 & 01 & 05 & 07 & 08 \\ 08 & 06 & 01 & 02 & 01 & 01 & 05 & 07 \\ 07 & 08 & 06 & 01 & 02 & 01 & 01 & 05 \\ 05 & 07 & 08 & 06 & 01 & 02 & 01 & 01 \\ 01 & 05 & 07 & 08 & 06 & 01 & 02 & 01 \\ 01 & 01 & 05 & 07 & 08 & 06 & 01 & 02 \end{bmatrix} \times \begin{bmatrix} S[a_{0,j}] \\ S[a_{1,j+1\,mod\,8}] \\ S[a_{2,j+2\,mod\,8}] \\ S[a_{3,j+3\,mod\,8}] \\ S[a_{4,j+4\,mod\,8}] \\ S[a_{5,j+5\,mod\,8}] \\ S[a_{6,j+6\,mod\,8}] \\ S[a_{7,j+7\,mod\,8}] \end{bmatrix}$$

where $S: \{0,1\}^8 \longrightarrow \{0,1\}^8$ denotes the S-box operation.

The matrix multiplication can be interpreted as a linear combination of all eight column vectors:

$$\begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \\ d_{4,j} \\ d_{5,j} \\ d_{6,j} \\ d_{7,j} \end{bmatrix} = \begin{bmatrix} 02 \\ 01 \\ 06 \\ 08 \\ 07 \\ 05 \\ 01 \\ 01 \end{bmatrix} S[a_{0,j}] \oplus \begin{bmatrix} 01 \\ 02 \\ 01 \\ 06 \\ 08 \\ 07 \\ 05 \\ 01 \end{bmatrix} S[a_{0,j}] \oplus \ldots \oplus \begin{bmatrix} 01 \\ 06 \\ 08 \\ 07 \\ 05 \\ 01 \\ 01 \\ 02 \end{bmatrix} S[a_{0,j+7\,mod\,8}]$$

We define now eight T-tables: $T_0, T_1, \ldots, T_7$:

$$T_0[\alpha] = \begin{bmatrix} 02 \\ 01 \\ 06 \\ 08 \\ 07 \\ 05 \\ 01 \\ 01 \end{bmatrix} S[\alpha], \quad T_1[\alpha] = \begin{bmatrix} 01 \\ 02 \\ 01 \\ 06 \\ 08 \\ 07 \\ 05 \\ 01 \end{bmatrix} S[\alpha] \quad \ldots \quad T_7[\alpha] = \begin{bmatrix} 01 \\ 06 \\ 08 \\ 07 \\ 05 \\ 01 \\ 01 \\ 02 \end{bmatrix} S[\alpha].$$

$$\begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \\ d_{4,j} \\ d_{5,j} \\ d_{6,j} \\ d_{7,j} \end{bmatrix} = T_0[a_0, j\,mod\,8] \oplus T_1[a_1, j+1\,mod\,8] \oplus T_1[a_1, j+2\,mod\,8] \oplus \ldots$$

$$\oplus T_0[a_0, j+7\,mod\,8].$$

All operations are 64-bit XOR operations that can be implemented quite efficiently on 64-bit platforms. Unlike AES, we do not require decryption support, which makes the implementation footprints smaller.

32-Bit Platforms

By splitting the 64-bit look-up tables, $T_0, \dots, T_7$, into 32-bit chunks we take presumably about twice the time for a MINI-ROUND as compared to the 64-bit implementation, i.e., the speed linearly scales down with the bandwidth available on a specific platform.

Specific Remarks for 8-Bit Platforms

The performance on 8-bit processors is an important issue, since most smart cards with cryptocraphic applications have such a processor. There are several options for implementing TWISTER, depending on whether the requirements demand for minimum space (i.e., no tables can be stored) or maximum speed. If minimum space is required, then the multiplication of two elements in $GF(2^8)$ has to be performed in software and can not be stored as a lookup table. There are three possible options of how to handle lookup tables for the simple field multiplication. First, if no memory is available the operations have to be computed instantly without any table lookups. Second, if there is only little memory on the devices such that these tables for multiplication can be stored and third, if the devices have plenty of memory, we can even store our full optimized lookup tables. Specific details for such issues can be found in [40, Chapter 4.1.1]. If space limitations is not an issue, the technique for implementing TWISTER via lookup-tables should be chosen as discussed in this case, if there is no conflict with the word size in the data path. As nearly all of the operations scale linearly down (i.e., a 64-bit XOR can be easily implemented via 8 times an 8-bit XOR) we simply have the running time of 8 times running time on 64-bit.

Dedicated Hardware

TWISTER is suited to be implemented in dedicated hardware. There are several possible trade-offs between chip area and speed. Because the implementation in software on general-purpose processors is already fast, the need for hardware implementation is very probably limited to very specific cases like:

1. Extreme high-speed chips with no area restrictions: the T-tables can be hard-wired and the XOR operations can be conducted in parallel.

2. Compact coprocessors on smart cards, there can either be only the S-box hard-wired or, additionally (and if enough memory is available) the T-tables generated at runtime.

Decomposition of the S-box

As we use the Rijndael S-box, we can assemble it using two transformations:

$$S[\alpha] = f(g(\alpha)),$$

where $g(\alpha)$ is the transformation

$$\alpha \to \alpha^{-1} \text{ in } GF(2^8)$$

and $f(\alpha)$ is an affine transformation.

The problem of designing efficient circuits for inversion in finite field has been studien extensively before; e.g., by C. Paar and M. Rosner in [123] or, for a short summary, in [40].

Performance Measurements

TWISTER was especially designed with 64-platforms in mind by making it possible to aggregate 8 times an 8-bit table lookup into one single 64-bit table lookup. The following performance measurements were conducted for 32-bit and 64-bit measurements:

Processor: Core2Duo $T7300$, 2 MB L2 Cache

Clock Speed: 2000 MHz

Memory: 2048 MB

Operating System: Linux, GNU Debian *Lenny*, Kernel 2.6.26-1 x64

Compiler: GCC 4.3, Optimization settings: -Os

For comparison, performance measurement results for SHA2 on the this platform are given in the following table.[5]

Table 5.2: Performance comparison of TWISTER and SHA-2

Algorithmn	**Hash size (in bit)**	**Platform**	**Time (in cycles/byte)**
SHA-256	256	64-bit	20.1
SHA-512	512	64-bit	13.1
TWISTER-224/256	224/256	64-bit	15.8
TWISTER-384/512	384/512	64-bit	17.5
SHA-256	256	32-bit	29.3
SHA-512	512	32-bit	55.2
TWISTER-224/256	224/256	32-bit	35.8
TWISTER-384/512	384/512	32-bit	39.6

[5]The reference source code for SHA-2 is taken from the Linux, GNU Debian *Lenny*, Kernel 2.6.26-1 x64, OpenSSL version: 0.98 implementation.

Chapter 6

Summary and Conclusions

This thesis focuses on cryptanalysis of various kinds of block ciphers and hash functions. We have shown flaws in a broad range of block ciphers, which outlines structural weaknesses. We presented some analyses of the building blocks of several hash functions and outlined some flaws of them. Some of these internal building blocks are shown to be broken in their use for encryption but remain secure as a component for a compression function of a hash function. This is for example the case for HAS-160, where the hash function remains still unbroken but the compression function used as a block cipher is not. We also introduced a new attack method for hash functions. Finally, we proposed a new hash function for the SHA-3 competition, which is called TWISTER.

To summarize we obtained the following results:

- the first known attack on reduced round SHACAL-2 which works with very little memory,
- attacks on reduced-round variant of the block ciphers AES-192, AES-256 and ARIA,
- the first known attack on the full Tiger in encryption mode,
- the first known attack on 77-round HAS-160 in encryption mode,
- a new attack method called *slide attacks* on a class of hash functions,
- a new proposal for the SHA-3 competition called TWISTER.

The designers of block ciphers and hash functions should consider the attacks presented in this thesis to make future constructions more secure against these kinds of attacks. As shown by the presented attacks, cryptographic primitives can be weak in one scenario and strong in another one. Therefore, cryptographic primitives have to be designed keeping the scenario of application in mind.

Further research should deal with the application of our new attack to other hash functions. It might be possible to adopt more attacks, which are well studied in a block cipher scenario to hash functions. For example, one should try to combine the slide attack with other attacks to increase the efficiency of our attack on hash functions.

There are many improvements, which can be applied based on the TWISTER hash function. If one can find good MDS-matrixes of size 16×16 the TWISTER-512 design could beware more consistent compared with the TWISTER-256 design. Also a good way for parallel versions of TWISTER could be approached.

The main results in this thesis have been published in [34, 50, 51, 52, 53, 56, 60, 64, 61, 62, 47, 72]. Other results in symmetric cryptography that are not considered in this thesis and have been published during my studies, can be found in [58, 59, 63].

List of Publications

Lecture Notes in Computer Science

1. **Michael Gorski**, Thomas Peyrin and Stefan Lucks, *Slide Attacks on a Class of Hash Functions*. In Josef Pieprzyk, editor, ASIACRYPT 2008, volume 5350 of Lecture Notes in Computer Science, pages 143 – 160. Springer, 2008.

2. **Michael Gorski** and Stefan Lucks, *New Related-Key Boomerang Attacks on AES*. In Chowdhury et al., editors, Progress in Cryptology – INDOCRYPT 2008, volume 5365 of Lecture Notes in Computer Science, pages 266 – 278, Springer, 2008.

3. Ewan Fleischmann, Christian Forler, **Michael Gorski** and Stefan Lucks, *Twister – A Framework for Secure and Fast Hash Functions*. In Bao et al., editors, Information Security Practice and Experience, 5th International Conference, ISPEC 2009, volume 5451 of Lecture Notes in Computer Science, pages 257 – 273, Springer, 2009.

4. Ewan Fleischmann, **Michael Gorski** and Stefan Lucks, *Memoryless Related-Key Boomerang Attack on 39-Round SHACAL-2*. In Bao et al., editors, Information Security Practice and Experience, 5th International Conference, ISPEC 2009, volume 5451 of Lecture Notes in Computer Science, pages 310 – 323, Springer, 2009.

5. Ewan Fleischmann, **Michael Gorski** and Stefan Lucks, *Memoryless Related-Key Boomerang Attack on the Full Tiger Block Cipher*. In Bao et al., editors, Information Security Practice and Experience, 5th International Conference, ISPEC 2009, volume 5451 of Lecture Notes in Computer Science, pages 298 – 309, Springer, 2009.

6. Ewan Fleischmann, **Michael Gorski** and Stefan Lucks, *On the Security of Tandem-DM*. In O. Dunkelman, editor, Proceedings of Fast Software Encryption 2009,

volume 5665 of Lecture Notes in Computer Science, pages 84 – 103, Springer, 2009.

7. Ewan Fleischmann, **Michael Gorski** and Stefan Lucks, *Attacking 9 and 10 Rounds of AES-256*. In C. Boyd and J. González Nieto, editors, ACISP, volume 5594 of Lecture Notes in Computer Science, pages 60 – 72, Springer, 2009.

8. Ewan Fleischmann, **Michael Gorski** and Stefan Lucks, *Security of Cyclic Double Block Length Hash Functions*. In M. G. Parker, editor, Proceedings of Cryptography and Coding, 12th IMA International Conference 2009, volume 5921 of Lecture Notes in Computer Science, pages 153 – 175, Springer, 2009.

9. Orr Dunkelmann, Ewan Fleischmann, **Michael Gorski** and Stefan Lucks, *Related-Key Rectangle Attack of the Full 80-Round HAS-160 Encryption Mode*. In R. Roy and N. Sendrier, editors, Progress in Cryptology – INDOCRYPT 2009, volume 5922 of Lecture Notes in Computer Science, pages 157 – 168, Springer, 2009.

10. Ewan Fleischmann, **Michael Gorski**, Jan-Hendrik Hühne and Stefan Lucks, *Key Recovery Attack on full GOST Block Cipher with Negligible Time and Memory*. Procedings of WEWoRC 2009, to appear in Lecture Notes in Computer Science.

11. Ewan Fleischmann, **Michael Gorski** and Stefan Lucks, *Some observations on Indifferentiability*. Information Security and Privacy, 15th Australasian Conference, ACISP 2010, to appear in Lecture Notes in Computer Science.

12. Ewan Fleischmann, Christian Forler, **Michael Gorski** and Stefan Lucks, *Collision Resistant Double-Length Hashing*. The 4th International Conference on Provable Security, ProvSec 2010, to appear in Lecture Notes in Computer Science.

13. Ewan Fleischmann, Christian Forler, **Michael Gorski** and Stefan Lucks, *New Boomerang Attacks on ARIA*. Progress in Cryptology – INDOCRYPT 2010, to appear in Lecture Notes in Computer Science.

International Publications in Journals

1. Ewan Fleischmann, Christian Forler, **Michael Gorski** and Stefan Lucks, *Twister – A Framework for Secure and Fast Hash Functions*. International Journal of Applied Cryptography, ISSN: 1753-0563, 2010. (to appear)

IACR ePrint Reports

1. **Michael Gorski** and Stefan Lucks, *New Related-Key Boomerang Attacks on AES (Extended Version)*, Cryptographic ePrint, 2008/263.
2. Ewan Fleischmann, Christian Forler and **Michael Gorski**, *Classification of SHA-3 Candidates*, Cryptology ePrint, 2008/511.
3. Ewan Fleischmann, **Michael Gorski** and Stefan Lucks, *On the Security of Tandem-DM (Extended Version)*, Cryptology ePrint, 2009/054.
4. Ewan Fleischmann, **Michael Gorski** and Stefan Lucks, *Attacking Reduced Rounds of the ARIA Block Cipher*, Cryptology ePrint, 2009/334.
5. Ewan Fleischmann, Christian Forler and **Michael Gorski**, *Some Observations on SHAMATA*, Cryptology ePrint, 2008/501.
6. Ewan Fleischmann, Christian Forler and **Michael Gorski**, *Classification of the SHA-3 Candidates*, Cryptology ePrint, 2008/511.
7. Ewan Fleischmann, **Michael Gorski** and Stefan Lucks, *Cryptanalysis of the full 80-Round HAS-160 Encryption Mode*, Cryptology ePrint, 2009/335.

International Conferences/Workshops without Proceedings

1. Ewan Fleischmann, Christian Forler and **Michael Gorski**, *The Twister Hash Function Family*, Submission for the NIST SHA-3 Competition. Presented at NIST first hash function workshop, Leuven, 2009.

Bibliography

[1] Ross J. Anderson and Eli Biham. TIGER: A Fast New Hash Function. In Gollmann [70], pages 89–97.

[2] Elena Andreeva, Charles Bouillaguet, Pierre-Alain Fouque, Jonathan J. Hoch, John Kelsey, Adi Shamir, and Sébastien Zimmer. Second Preimage Attacks on Dithered Hash Functions. In Smart [142], pages 270–288.

[3] Feng Bao, Hui Li, and Guilin Wang, editors. *Information Security Practice and Experience, 5th International Conference, ISPEC 2009, Xi'an, China, April 13-15, 2009, Proceedings*, volume 5451 of *Lecture Notes in Computer Science*. Springer, 2009.

[4] Mihir Bellare. New Proofs for NMAC and HMAC: Security without Collision-Resistance. In Cynthia Dwork, editor, *CRYPTO*, volume 4117 of *Lecture Notes in Computer Science*, pages 602–619. Springer, 2006.

[5] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying Hash Functions for Message Authentication. In Neal Koblitz, editor, *CRYPTO*, volume 1109 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 1996.

[6] Mihir Bellare and Phillip Rogaway. Random Oracles are Practical: A Paradigm for Designing Efficient Protocols. In *ACM Conference on Computer and Communications Security*, pages 62–73, 1993.

[7] Daniel J. Bernstein and Peter Schwabe. New AES Software Speed Records. In Chowdhury et al. [34], pages 322–336.

[8] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. RadioGatún, a belt-and-mill hash function. Presented at Second Cryptographic Hash Workshop, Santa Barbara, 2006. Available online at `http://radiogatun.noekeon.org/`.

[9] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. Sponge Functions. Ecrypt Hash Workshop, Barcelona, 2007. Available online at `http://gva.noekeon.org/papers/bdpv07.html`.

[10] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. On the Indifferentiability of the Sponge Construction. In Smart [142], pages 181–197.

[11] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. Sponge Functions, presented at ECRYPT Hash Workshop, Barcelona, 2007.

[12] Eli Biham. New Types of Cryptoanalytic Attacks Using related Keys (Extended Abstract). In Tor Helleseth, editor, *EUROCRYPT 1993*, volume 765 of *Lecture Notes in Computer Science*, pages 398–409. Springer, 1993.

[13] Eli Biham, Alex Biryukov, and Adi Shamir. Cryptanalysis of Skipjack Reduced to 31 Rounds Using Impossible Differentials. In Jacques Stern, editor, *EUROCRYPT 1999*, volume 1592 of *Lecture Notes in Computer Science*, pages 12–23. Springer, 1999.

[14] Eli Biham and Rafi Chen. Near-Collisions of SHA-0. In Franklin [66], pages 290–305.

[15] Eli Biham, Rafi Chen, Antoine Joux, Patrick Carribault, Christophe Lemuet, and William Jalby. Collisions of SHA-0 and Reduced SHA-1. In Cramer [38], pages 36–57.

[16] Eli Biham, Orr Dunkelman, and Nathan Keller. The Rectangle Attack - Rectangling the Serpent. In Birgit Pfitzmann, editor, *EUROCRYPT 2001*, volume 2045 of *Lecture Notes in Computer Science*, pages 340–357. Springer, 2001.

[17] Eli Biham, Orr Dunkelman, and Nathan Keller. Related-Key Boomerang and Rectangle Attacks. In Cramer [38], pages 507–525.

[18] Eli Biham, Orr Dunkelman, and Nathan Keller. Related-Key Impossible Differential Attacks on 8-Round AES-192. In David Pointcheval, editor, *CT-RSA 2006*, volume 3860 of *Lecture Notes in Computer Science*, pages 21–33. Springer, 2006.

[19] Eli Biham, Orr Dunkelman, and Nathan Keller. Improved Slide Attacks. In Biryukov [23], pages 153–166.

[20] Eli Biham and Adi Shamir. Differential Cryptanalysis of DES-like Cryptosystems. In Menezes and Vanstone [112], pages 2–21.

[21] Eli Biham and Adi Shamir. Differential Cryptanalysis of the Full 16-Round DES. In Ernest F. Brickell, editor, *CRYPTO 1992*, volume 740 of *Lecture Notes in Computer Science*, pages 487–496. Springer, 1992.

[22] Eli Biham and Adi Shamir. *Differential Cryptanalysis of the Data Encryption Standard*. Springer, 1993.

[23] Alex Biryukov, editor. *Fast Software Encryption, 14th International Workshop, FSE 2007, Luxembourg, Luxembourg, March 26-28, 2007, Revised Selected Papers*, volume 4593 of *Lecture Notes in Computer Science*. Springer, 2007.

[24] Alex Biryukov, Orr Dunkelman, Nathan Keller, Dmitry Khovratovich, and Adi Shamir. Key Recovery Attacks of Practical Complexity on AES-256 Variants with up to 10 Rounds. In Henri Gilbert, editor, *EUROCRYPT*, volume 6110 of *Lecture Notes in Computer Science*, pages 299–319. Springer, 2010.

[25] Alex Biryukov and Dmitry Khovratovich. Related-Key Cryptanalysis of the Full AES-192 and AES-256. In Mitsuru Matsui, editor, *ASIACRYPT*, volume 5912 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2009.

[26] Alex Biryukov and Dmitry Khovratovich. Feasible Attack on the 13-round AES-256. Cryptology ePrint Archive, Report 2010/257, 2010. `http://eprint.iacr.org/`.

[27] Alex Biryukov and David Wagner. Slide Attacks. In Knudsen [96], pages 245–259.

[28] Alex Biryukov and David Wagner. Advanced Slide Attacks. In Bart Preneel, editor, *EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 589–606. Springer, 2000.

[29] John Black, Phillip Rogaway, and Thomas Shrimpton. Black-Box Analysis of the Block-Cipher-Based Hash-Function Constructions from PGV. In Yung [156], pages 320–335.

[30] Andrey Bogdanov. Linear Slide Attacks on the KeeLoq Block Cipher. In Dingyi Pei, Moti Yung, Dongdai Lin, and Chuankun Wu, editors, *Inscrypt*, volume 4990 of *Lecture Notes in Computer Science*, pages 66–80. Springer, 2007.

[31] Gilles Brassard, editor. *Advances in Cryptology - CRYPTO 1989, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*, volume 435 of *Lecture Notes in Computer Science*. Springer, 1990.

[32] Florent Chabaud and Antoine Joux. Differential Collisions in SHA-0. In Hugo Krawczyk, editor, *CRYPTO 1998*, volume 1462 of *Lecture Notes in Computer Science*, pages 56–71. Springer, 1998.

[33] Hong-Su Cho, Sangwoo Park, Soo Hak Sung, and Aaram Yun. Collision Search Attack for 53-Step HAS-160. In Min Surp Rhee and Byoungcheon Lee, editors, *ICISC 2006*, volume 4296 of *Lecture Notes in Computer Science*, pages 286–295. Springer, 2006.

[34] Dipanwita Roy Chowdhury, Vincent Rijmen, and Abhijit Das, editors. *Progress in Cryptology - INDOCRYPT 2008, 9th International Conference on Cryptology in India, Kharagpur, India, December 14-17, 2008. Proceedings*, volume 5365 of *Lecture Notes in Computer Science*. Springer, 2008.

[35] Christophe Clavier and Kris Gaj, editors. *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*, volume 5747 of *Lecture Notes in Computer Science*. Springer, 2009.

[36] Jean-Sébastien Coron, Yevgeniy Dodis, Cécile Malinaud, and Prashant Puniya. Merkle-Damgård Revisited: How to Construct a Hash Function. In Shoup [141], pages 430–448.

[37] Nicolas Courtois, Gregory V. Bard, and David Wagner. Algebraic and Slide Attacks on KeeLoq. In Nyberg [116], pages 97–115.

[38] Ronald Cramer, editor. *Advances in Cryptology - EUROCRYPT 2005, 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Aarhus, Denmark, May 22-26, 2005, Proceedings*, volume 3494 of *Lecture Notes in Computer Science*. Springer, 2005.

[39] Joan Daemen and Vincent Rijmen. AES Proposal: Rijndael. NIST AES, Available online at `http://csrc.nist.gov/encryption/aes/round2/r2algs.htm`, 1999.

[40] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer, 2002.

[41] Ivan Damgård. A Design Principle for Hash Functions. In Brassard [31], pages 416–427.

[42] Richared D. Deam. Formal Aspects of Mobile Code Security. Ph.D. thesis, Princeton University, 1999.

[43] Hüseyin Demirci and Ali Aydin Selçuk. A Meet-in-the-Middle Attack on 8-Round AES. In Nyberg [116], pages 116–126.

[44] Hans Dobbertin. Cryptanalysis of MD4. *Journal Cryptology*, 11(4):253–271, 1998.

[45] Ali Doganaksoy, Onur Ozen, and Kerem Varıcı. On the Security of the Encryption Mode of Tiger. private communications.

[46] Orr Dunkelman, editor. *Fast Software Encryption, 16th International Workshop, FSE 2009, Leuven, Belgium, February 22-25, 2009, Revised Selected Papers*, volume 5665 of *Lecture Notes in Computer Science*. Springer, 2009.

[47] Orr Dunkelman, Ewan Fleischmann, Michael Gorski, and Stefan Lucks. Related-Key Rectangle Attack of the Full HAS-160 Encryption Mode. In Bimal Roy and Nicolas Sendrier, editors, *INDOCRYPT 2009*, volume 5922 of *Lecture Notes in Computer Science*, pages 157–168. Springer, 2009.

[48] Orr Dunkelman, Nathan Keller, and Adi Shamir. Improved Single-Key Attacks on 8-round AES. Cryptology ePrint Archive, Report 2010/322, 2010. `http://eprint.iacr.org/`.

[49] Niels Ferguson, John Kelsey, Stefan Lucks, Bruce Schneier, Michael Stay, David Wagner, and Doug Whiting. Improved Cryptanalysis of Rijndael. In Schneier [137], pages 213–230.

[50] Ewan Fleischmann, Christian Forler, and Michael Gorski. The TWISTER Hash Function Family. Available online at `http://csrc.nist.gov/groups/ST/hash/sha-3/`.

[51] Ewan Fleischmann, Christian Forler, and Michael Gorski. The TWISTER Hash Function Family. Available online at `http://www.twister-hash.com`.

[52] Ewan Fleischmann, Christian Forler, and Michael Gorski. Classification of the SHA-3 Candidates. Cryptology ePrint Archive, Report 2008/511, 2008. Available online at `http://eprint.iacr.org/`.

[53] Ewan Fleischmann, Christian Forler, Michael Gorski, and Stefan Lucks. TWISTER – A Framework for Secure and Fast Hash Functions. In Bao et al. [3], pages 257–273.

[54] Ewan Fleischmann, Christian Forler, Michael Gorski, and Stefan Lucks. Security of Cyclic Double Block Length Hash Functions. Cryptography and Coding, 12th IMA International Conference, Cirencester (UK), 2009, M. G. Parker (Ed.) LNCS 5921, pages 153–175, Springer 2009.

[55] Ewan Fleischmann, Christian Forler, Michael Gorski, and Stefan Lucks. New Boomerang Attacks on ARIA. In *INDOCRYPT 2010*, 2010, to appear.

[56] Ewan Fleischmann, Christian Forler, Michael Gorski, and Stefan Lucks. TWISTER - A Framework for Secure and Fast Hash Functions. *International Journal of Applied Cryptography*, 2010, to appear.

[57] Ewan Fleischmann, Christian Forler, Michael Gorski, and Stefan Lucks. Collision Resistant Double-Length Hashing. In *ProvSec 2010*, to appear in Lecture Notes in Computer Science.

[58] Ewan Fleischmann and Michael Gorski. Some Observations on SHAMATA. Cryptology ePrint Archive, Report 2008/501, 2008. `http://eprint.iacr.org/`.

[59] Ewan Fleischmann, Michael Gorski, Jan-Hendrik Hühne, and Stefan Lucks. Key Recovery Attack on full GOST Block Cipher with Negligible Time and Memory. WEWoRC 2009, 2009, to appear in Lecture Notes in Computer Science.

[60] Ewan Fleischmann, Michael Gorski, and Stefan Lucks. Attacking 9 and 10 Rounds of AES-256. In C. Boyd and J. González Nieto, editors, *ACISP 2009*, volume 5594 of *Lecture Notes in Computer Science*, pages 60–72. Springer, 2009.

[61] Ewan Fleischmann, Michael Gorski, and Stefan Lucks. Memoryless Related-Key Boomerang Attack on 39-Round SHACAL-2. In Bao et al. [3], pages 310–323.

[62] Ewan Fleischmann, Michael Gorski, and Stefan Lucks. Memoryless Related-Key Boomerang Attack on the Full Tiger Block Cipher. In Bao et al. [3], pages 298–309.

[63] Ewan Fleischmann, Michael Gorski, and Stefan Lucks. On the Security of Tandem-DM. In Dunkelman [46], pages 84–103.

[64] Ewan Fleischmann, Michael Gorski, and Stefan Lucks. Attacking Reduced Rounds of the ARIA Block Cipher. WeWORC, 2009, to appear in Lecture Notes in Computer Science.

[65] Ewan Fleischmann, Michael Gorski, and Stefan Lucks. Some observations on Indifferentiability. In *ACISP 2010*, to appear in Lecture Notes in Computer Science.

[66] Matthew K. Franklin, editor. *Advances in Cryptology - CRYPTO 2004, 24th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 2004, Proceedings*, volume 3152 of *Lecture Notes in Computer Science*. Springer, 2004.

[67] Soichi Furuya. Slide Attacks with a Known-Plaintext Cryptanalysis. In Kwangjo Kim, editor, *ICISC 2001*, volume 2288 of *Lecture Notes in Computer Science*, pages 214–225. Springer, 2001.

[68] Henri Gilbert and Helena Handschuh, editors. *Fast Software Encryption: 12th International Workshop, FSE 2005, Paris, France, February 21-23, 2005, Revised Selected Papers*, volume 3557 of *Lecture Notes in Computer Science*. Springer, 2005.

[69] Henri Gilbert and Marine Minier. A collision attack on 7 rounds of Rijndael. *In The Third AES Candidate Conference*, 2000.

[70] Dieter Gollmann, editor. *Fast Software Encryption, Third International Workshop, Cambridge, UK, February 21-23, 1996, Proceedings*, volume 1039 of *Lecture Notes in Computer Science*. Springer, 1996.

[71] Michael Gorski and Stefan Lucks. New Related-Key Boomerang Attacks on AES. In Chowdhury et al. [34], pages 266–278.

[72] Michael Gorski, Stefan Lucks, and Thomas Peyrin. Slide Attacks on a Class of Hash Functions. In Josef Pieprzyk, editor, *ASIACRYPT 2008*, volume 5350 of *Lecture Notes in Computer Science*, pages 143–160. Springer, 2008.

[73] Jian Guo, San Ling, Christian Rechberger, and Huaxiong Wang. Advanced Meet-in-the-Middle Preimage Attacks: First Results on Full Tiger, and Improved Results on MD4 and SHA-2. Cryptology ePrint Archive, Report 2010/016, 2010. `http://eprint.iacr.org/`.

[74] Mike Hamburg. Accelerating AES with Vector Permute Instructions. In Clavier and Gaj [35], pages 18–32.

[75] Helena Handschuh and David Naccache. SHACAL: A Family of Block Ciphers. Submission to the NESSIE project, 2002. Available online at `http://www.cosic.esat.kuleuven.be/nessie/tweaks.html`.

[76] Seokhie Hong and Tetsu Iwata, editors. *Fast Software Encryption, 17th International Workshop, FSE 2010, Seoul, Korea, February 7-10, 2010, Revised Selected Papers*, volume 6147 of *Lecture Notes in Computer Science*. Springer, 2010.

[77] Seokhie Hong, Jongsung Kim, Guil Kim, Jaechul Sung, Changhoon Lee, and Sangjin Lee. Impossible Differential Attack on 30-Round SHACAL-2. In Thomas Johansson and Subhamoy Maitra, editors, *INDOCRYPT 2003*, volume 2904 of *Lecture Notes in Computer Science*, pages 97–106. Springer, 2003.

[78] Seokhie Hong, Jongsung Kim, Sangjin Lee, and Bart Preneel. Related-Key Rectangle Attacks on Reduced Versions of SHACAL-1 and AES-192. In Gilbert and Handschuh [68], pages 368–383.

[79] Sebastiaan Indesteege and Bart Preneel. Preimages for Reduced-Round Tiger. In Stefan Lucks, Ahmad-Reza Sadeghi, and Christopher Wolf, editors, *WEWoRC 2007*, volume 4945 of *Lecture Notes in Computer Science*, pages 90–99. Springer, 2007.

[80] Takanori Isobe and Kyoji Shibutani. Preimage Attacks on Reduced Tiger and SHA-2. In Dunkelman [46], pages 139–155.

[81] Goce Jakimoski and Yvo Desmedt. Related-Key Differential Cryptanalysis of 192-bit Key AES Variants. In Mitsuru Matsui and Robert J. Zuccherato, editors, *Selected Areas in Cryptography 2003*, volume 3006 of *Lecture Notes in Computer Science*, pages 208–221. Springer, 2003.

[82] Antoine Joux. Multicollisions in Iterated Hash Functions. Application to Cascaded Constructions. In Franklin [66], pages 306–316.

[83] Emilia Käsper and Peter Schwabe. Faster and Timing-Attack Resistant AES-GCM. In Clavier and Gaj [35], pages 1–17.

[84] Selçuk Kavut and Melek D. Yücel. Slide Attack on Spectr-H64. In Alfred Menezes and Palash Sarkar, editors, *INDOCRYPT 2002*, volume 2551 of *Lecture Notes in Computer Science*, pages 34–47. Springer, 2002.

[85] John Kelsey and Tadayoshi Kohno. Herding Hash Functions and the Nostradamus Attack. In Serge Vaudenay, editor, *EUROCRYPT 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 183–200. Springer, 2006.

[86] John Kelsey, Tadayoshi Kohno, and Bruce Schneier. Amplified Boomerang Attacks Against Reduced-Round MARS and Serpent. In Schneier [137], pages 75–93.

[87] John Kelsey and Stefan Lucks. Collisions and Near-Collisions for Reduced-Round Tiger. In Robshaw [133], pages 111–125.

[88] John Kelsey and Bruce Schneier. Second Preimages on n-Bit Hash Functions for Much Less than 2^n Work. In Cramer [38], pages 474–490.

[89] Auguste Kerckhoffs. La cryptographie militaire. *Journal des sciences militaires*, 9(1):5–38, 1883.

[90] Jongsung Kim, Seokhie Hong, and Bart Preneel. Related-Key Rectangle Attacks on Reduced AES-192 and AES-256. In Biryukov [23], pages 225–241.

[91] Jongsung Kim, Guil Kim, Seokhie Hong, Sangjin Lee, and Dowon Hong. The Related-Key Rectangle Attack — Application to SHACAL-1. In Wang et al. [147], pages 123–136.

[92] Jongsung Kim, Guil Kim, Sangjin Lee, Jongin Lim, and Jung Hwan Song. Related-Key Attacks on Reduced Rounds of SHACAL-2. In Anne Canteaut and Kapalee Viswanathan, editors, *INDOCRYPT 2004*, volume 3348 of *Lecture Notes in Computer Science*, pages 175–190. Springer, 2004.

[93] Lars R. Knudsen. Cryptanalysis of LOKI91. In Jennifer Seberry and Yuliang Zheng, editors, *ASIACRYPT 1992*, volume 718 of *Lecture Notes in Computer Science*, pages 196–208. Springer, 1992.

[94] Lars R. Knudsen. Practically Secure Feistel Cyphers. In Ross J. Anderson, editor, *FSE 1993*, volume 809 of *Lecture Notes in Computer Science*, pages 211–221. Springer, 1993.

[95] Lars R. Knudsen. Truncated and Higher Order Differentials. In Bart Preneel, editor, *FSE 1994*, volume 1008 of *Lecture Notes in Computer Science*, pages 196–211. Springer, 1994.

[96] Lars R. Knudsen, editor. *Fast Software Encryption, 6th International Workshop, FSE 1999, Rome, Italy, March 24-26, 1999, Proceedings*, volume 1636 of *Lecture Notes in Computer Science*. Springer, 1999.

[97] Lars R. Knudsen, Christian Rechberger, and Søren S. Thomsen. The Grindahl Hash Functions. In Biryukov [23], pages 39–57.

[98] Kaoru Kurosawa, editor. *Advances in Cryptology - ASIACRYPT 2007, 13th International Conference on the Theory and Application of Cryptology and Information Security, Kuching, Malaysia, December 2-6, 2007, Proceedings*, volume 4833 of *Lecture Notes in Computer Science*. Springer, 2007.

[99] Daesung Kwon, Jaesung Kim, Sangwoo Park, Soo Hak Sung, Yaekwon Sohn, Jung Hwan Song, Yongjin Yeom, E-Joong Yoon, Sangjin Lee, Jaewon Lee, Seongtaek Chee, Daewan Han, and Jin Hong. New Block Cipher: ARIA. In Jong In Lim and Dong Hoon Lee, editors, *ICISC 2003*, volume 2971 of *Lecture Notes in Computer Science*, pages 432–445. Springer, 2003.

[100] Jiqiang Lu, Orr Dunkelman, Nathan Keller, and Jongsung Kim. New Impossible Differential Attacks on AES. In Chowdhury et al. [34], pages 279–293.

[101] Jiqiang Lu and Jongsung Kim. Attacking 44 Rounds of the SHACAL-2 Block Cipher Using Related-Key Rectangle Cryptanalysis. *IEICE Transactions*, 91-A(9):2588–2596, 2008.

[102] Jiqiang Lu, Jongsung Kim, Nathan Keller, and Orr Dunkelman. Related-Key Rectangle Attack on 42-Round SHACAL-2. In Sokratis K. Katsikas, Javier Lopez, Michael Backes, Stefanos Gritzalis, and Bart Preneel, editors, *ISC 2006*,

volume 4176 of *Lecture Notes in Computer Science*, pages 85–100. Springer, 2006.

[103] Stefan Lucks. A Failure-Friendly Design Principle for Hash Functions. In Bimal K. Roy, editor, *ASIACRYPT 2005*, volume 3788 of *Lecture Notes in Computer Science*, pages 474–494. Springer, 2005.

[104] Stefan Lucks. Attacking Seven Rounds of Rijndael under 192-bit and 256-bit Keys. In *AES Candidate Conference*, pages 215–229, Springer, 2000.

[105] Florence J. MacWilliams and Neil J. A. Sloane. The Theory of Error-Correcting Codes, North-Holland, 1977.

[106] Florian Mendel, Bart Preneel, Vincent Rijmen, Hirotaka Yoshida, and Dai Watanabe. Update on Tiger. In Rana Barua and Tanja Lange, editors, *INDOCRYPT 2006*, volume 4329 of *Lecture Notes in Computer Science*, pages 63–79. Springer, 2006.

[107] Florian Mendel, Christian Rechberger, and Martin Schläffer. Cryptanalysis of Twister. In Michel Abdalla and David Pointcheval, editors, *Applied Cryptography and Network Security 2009*, Lecture Notes in Computer Science, pages 342–353. Springer, 2009. in press.

[108] Florian Mendel, Christian Rechberger, Martin Schläffer, and Søren Steffen Thomsen. The Rebound Attack: Cryptanalysis of Reduced Whirlpool and Grøstl. In Dunkelman [46], pages 260–276.

[109] Florian Mendel and Vincent Rijmen. Colliding Message Pair for 53-Step HAS-160. In Nam and Rhee [115], pages 324–334.

[110] Florian Mendel and Vincent Rijmen. Cryptanalysis of the Tiger Hash Function. In Kurosawa [98], pages 536–550.

[111] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of applied cryptography*. CRC, 1997.

[112] Alfred J. Menezes and Scott A. Vanstone, editors. *Advances in Cryptology - CRYPTO 1990, 10th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1990, Proceedings*, volume 537 of *Lecture Notes in Computer Science*. Springer, 1991.

[113] Ralph C. Merkle. One Way Hash Functions and DES. In Brassard [31], pages 428–446.

[114] Nicky Mouha, Gautham Sekar, Jean-Philippe Aumasson, Thomas Peyrin, Søren S. Thomsen, Meltem Sönmez Turan, and Bart Preneel. Cryptanalysis of the ESSENCE Family of Hash Functions. In *Inscrypt*, 2009, to appear in Lecture Notes in Computer Science.

[115] Kil-Hyun Nam and Gwangsoo Rhee, editors. *Information Security and Cryptology - ICISC 2007, 10th International Conference, Seoul, Korea, November 29-30, 2007, Proceedings*, volume 4817 of *Lecture Notes in Computer Science*. Springer, 2007.

[116] Kaisa Nyberg, editor. *Fast Software Encryption, 15th International Workshop, FSE 2008, Lausanne, Switzerland, February 10-13, 2008, Revised Selected Papers*, volume 5086 of *Lecture Notes in Computer Science*. Springer, 2008.

[117] National Bureau of Standards. FIPS 46: Data Encryption Standard. 1977.

[118] National Institute of Standards and Technology. FIPS 180-0: Secure Hash Standard. 1993. Available online at `http://csrc.nist.gov`.

[119] National Institute of Standards and Technology. FIPS 180-1: Secure Hash Standard. 1995. Available online at `http://csrc.nist.gov`.

[120] National Institute of Standards and Technology. FIPS 180-2: Secure Hash Standard. 2002. Available online at `http://csrc.nist.gov`.

[121] National Institute of Standards and Technology. Cryptographic Hash Project. Available online at `http://csrc.nist.gov/groups/ST/hash/index.html`.

[122] Dag Arne Osvik, Joppe W. Bos, Deian Stefan, and David Canright. Fast software aes encryption. In Hong and Iwata [76], pages 75–93.

[123] Christof Paar and Martin Rosner. Comparison of arithmetic architectures for reed-solomon decoders in reconfigurable hardware. In *FCCM 1997*, pages 219–225. IEEE Computer Society, 1997.

[124] Raphael Chung-Wei Phan. Impossible differential cryptanalysis of 7-round Advanced Encryption Standard (AES). *Information Processing Letters*, 91(1):33–38, 2004.

[125] Raphael Chung-Wei Phan. Advanced Slide Attacks Revisited: Realigning Slide on DES. In Ed Dawson and Serge Vaudenay, editors, *Mycrypt 2005*, volume 3715 of *Lecture Notes in Computer Science*, pages 263–276. Springer, 2005.

[126] Raphael Chung-Wei Phan and Soichi Furuya. Sliding Properties of the DES Key Schedule and Potential Extensions to the Slide Attacks. In Pil Joong Lee and Chae Hoon Lim, editors, *ICISC 2002*, volume 2587 of *Lecture Notes in Computer Science*, pages 138–148. Springer, 2002.

[127] Raphael Chung-Wei Phan and M.U. Siddiqi. Generalized Impossible Differentials of the Advanced Encryption Standard (AES). *IEE Electronics Letters*, 37(14):896–898, 2001.

[128] Bart Preneel and Paul C. van Oorschot. On the Security of Two MAC Algorithms. In Ueli M. Maurer, editor, *EUROCRYPT 1996*, volume 1070 of *Lecture Notes in Computer Science*, pages 19–32. Springer, 1996.

[129] Jean-Jacques Quisquater and Jean-Paul Delescaille. How Easy is Collision Search. New Results and Applications to DES. In Brassard [31], pages 408–413.

[130] Vincent Rijmen and Elisabeth Oswald. Update on SHA-1. In Alfred Menezes, editor, *CT-RSA 2005*, volume 3376 of *Lecture Notes in Computer Science*, pages 58–71. Springer, 2005.

[131] Ron Rivest. The MD4 Message Digest Algorithm. In Menezes and Vanstone [112], pages 303–311.

[132] Ron Rivest. The MD5 Message-Digest Algorithm. Request for Comments: 1321, Available online at http://tools.ietf.org/html/rfc1321, April 1992.

[133] Matthew J. B. Robshaw, editor. *Fast Software Encryption, 13th International Workshop, FSE 2006, Graz, Austria, March 15-17, 2006, Revised Selected Papers*, volume 4047 of *Lecture Notes in Computer Science*. Springer, 2006.

[134] Peng Zhang Ruilin Li, Bing Sun and Chao Li. New Impossible Differential Cryptanalysis of ARIA. Cryptology ePrint Archive, Report 2008/227, 2008. Available online at `http://eprint.iacr.org/`.

[135] Markku-Juhani Olavi Saarinen. Cryptanalysis of Block Ciphers Based on SHA-1 and MD5. In Thomas Johansson, editor, *FSE 2003*, volume 2887 of *Lecture Notes in Computer Science*, pages 36–44. Springer, 2003.

[136] Yu Sasaki, Lei Wang, Kazuo Ohta, and Noboru Kunihiro. Password Recovery on Challenge and Response: Impossible Differential Attack on Hash Function. In Serge Vaudenay, editor, *AFRICACRYPT 2008*, volume 5023 of *Lecture Notes in Computer Science*, pages 290–307. Springer, 2008.

[137] Bruce Schneier, editor. *Fast Software Encryption, 7th International Workshop, FSE 2000, New York, NY, USA, April 10-12, 2000, Proceedings*, volume 1978 of *Lecture Notes in Computer Science*. Springer, 2001.

[138] Bruce Schneier and John Kelsey. Unbalanced Feistel Networks and Block Cipher Design. In Gollmann [70], pages 121–144.

[139] Claude Shannon. Communication theory of secrecy systems. *Bell Systems Technical Journal*, 28(4):656–715, 1949.

[140] YongSup Shin, Jongsung Kim, Guil Kim, Seokhie Hong, and Sangjin Lee. Differential-Linear Type Attacks on Reduced Rounds of SHACAL-2. In Wang et al. [147], pages 110–122.

[141] Victor Shoup, editor. *Advances in Cryptology - CRYPTO 2005: 25th Annual International Cryptology Conference, Santa Barbara, California, USA, August 14-18, 2005, Proceedings*, volume 3621 of *Lecture Notes in Computer Science*. Springer, 2005.

[142] Nigel P. Smart, editor. *Advances in Cryptology - EUROCRYPT 2008, 27th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Istanbul, Turkey, April 13-17, 2008. Proceedings*, volume 4965 of *Lecture Notes in Computer Science*. Springer, 2008.

[143] Telecommunications Technology Association. Hash Function Standard Part 2: Hash Function Algorithm Standard (HAS-160). TTAS.KO-12.0011/R1, December 2000.

[144] David Wagner. The Boomerang Attack. In Knudsen [96], pages 156–170.

[145] David Wagner. A Generalized Birthday Problem. In Yung [156], pages 288–303.

[146] Gaoli Wang. Related-Key Rectangle Attack on 43-Round SHACAL-2. In Ed Dawson and Duncan S. Wong, editors, *ISPEC 2007*, volume 4464 of *Lecture Notes in Computer Science*, pages 33–42. Springer, 2007.

[147] Huaxiong Wang, Josef Pieprzyk, and Vijay Varadharajan, editors. *Information Security and Privacy: 9th Australasian Conference, ACISP 2004, Sydney, Australia, July 13-15, 2004. Proceedings*, volume 3108 of *Lecture Notes in Computer Science*. Springer, 2004.

[148] Lei Wang and Yu Sasaki. Finding Preimages of Tiger Up to 23 Steps. In Hong and Iwata [76], pages 116–133.

[149] Xiaoyun Wang, Xuejia Lai, Dengguo Feng, Hui Chen, and Xiuyuan Yu. Cryptanalysis of the Hash Functions MD4 and RIPEMD. In Cramer [38], pages 1–18.

[150] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding Collisions in the Full SHA-1. In Shoup [141], pages 17–36.

[151] Xiaoyun Wang and Hongbo Yu. How to Break MD5 and Other Hash Functions. In Cramer [38], pages 19–35.

[152] Robert S. Winternitz. A Secure One-Way Hash Function Built from DES. In *IEEE Symposium on Security and Privacy*, pages 88–90, 1984.

[153] Wenling Wu, Wentao Zhang, and Dengguo Feng. Impossible Differential Cryptanalysis of Reduced-Round ARIA and Camellia. *Journal of Computer Science and Technology*, 22(3):449–456, 2007.

[154] Xun Yi, Shi Xing Cheng, Xiao Hu You, and Kwok Yan Lam. A Method for Obtaining Cryptographically Strong 8x8 S-boxes. In *IEEE Global Telecommunications Conference, GLOBECOM 1997, Volume 2*, pages 689–693, 1997.

[155] Aaram Yun, Soo Hak Sung, Sangwoo Park, Donghoon Chang, Seokhie Hong, and Hong-Su Cho. Finding Collision on 45-Step HAS-160. In Dongho Won and Seungjoo Kim, editors, *ICISC 2005*, volume 3935 of *Lecture Notes in Computer Science*, pages 146–155. Springer, 2005.

[156] Moti Yung, editor. *Advances in Cryptology - CRYPTO 2002, 22nd Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 2002, Proceedings*, volume 2442 of *Lecture Notes in Computer Science*. Springer, 2002.

[157] Wentao Zhang, Wenling Wu, and Dengguo Feng. New Results on Impossible Differential Cryptanalysis of Reduced AES. In Nam and Rhee [115], pages 239–250.

[158] Wentao Zhang, Wenling Wu, Lei Zhang, and Dengguo Feng. Improved Related-Key Impossible Differential Attacks on Reduced-Round AES-192. In Eli Biham and Amr M. Youssef, editors, *Selected Areas in Cryptography 2006*, volume 4356 of *Lecture Notes in Computer Science*, pages 15–27. Springer, 2007.

[159] Wentao Zhang, Lei Zhang, Wenling Wu, and Dengguo Feng. Related-Key Differential-Linear Attacks on Reduced AES-192. In K. Srinathan, C. Pandu Rangan, and Moti Yung, editors, *INDOCRYPT 2007*, volume 4859 of *Lecture Notes in Computer Science*, pages 73–85. Springer, 2007.

Appendix A

The AES/Twister S-box

The twister S-box is taken from AES [40] which is as follows.

0x63	0x7c	0x77	0x7b	0xf2	0x6b	0x6f	0xc5
0x30	0x01	0x67	0x2b	0xfe	0xd7	0xab	0x76
0xca	0x82	0xc9	0x7d	0xfa	0x59	0x47	0xf0
0xad	0xd4	0xa2	0xaf	0x9c	0xa4	0x72	0xc0
0xb7	0xfd	0x93	0x26	0x36	0x3f	0xf7	0xcc
0x34	0xa5	0xe5	0xf1	0x71	0xd8	0x31	0x15
0x04	0xc7	0x23	0xc3	0x18	0x96	0x05	0x9a
0x07	0x12	0x80	0xe2	0xeb	0x27	0xb2	0x75
0x09	0x83	0x2c	0x1a	0x1b	0x6e	0x5a	0xa0
0x52	0x3b	0xd6	0xb3	0x29	0xe3	0x2f	0x84
0x53	0xd1	0x00	0xed	0x20	0xfc	0xb1	0x5b
0x6a	0xcb	0xbe	0x39	0x4a	0x4c	0x58	0xcf
0xd0	0xef	0xaa	0xfb	0x43	0x4d	0x33	0x85
0x45	0xf9	0x02	0x7f	0x50	0x3c	0x9f	0xa8
0x51	0xa3	0x40	0x8f	0x92	0x9d	0x38	0xf5
0xbc	0xb6	0xda	0x21	0x10	0xff	0xf3	0xd2
0xcd	0x0c	0x13	0xec	0x5f	0x97	0x44	0x17
0xc4	0xa7	0x7e	0x3d	0x64	0x5d	0x19	0x73
0x60	0x81	0x4f	0xdc	0x22	0x2a	0x90	0x88
0x46	0xee	0xb8	0x14	0xde	0x5e	0x0b	0xdb
0xe0	0x32	0x3a	0x0a	0x49	0x06	0x24	0x5c
0xc2	0xd3	0xac	0x62	0x91	0x95	0xe4	0x79
0xe7	0xc8	0x37	0x6d	0x8d	0xd5	0x4e	0xa9
0x6c	0x56	0xf4	0xea	0x65	0x7a	0xae	0x08
0xba	0x78	0x25	0x2e	0x1c	0xa6	0xb4	0xc6
0xe8	0xdd	0x74	0x1f	0x4b	0xbd	0x8b	0x8a
0x70	0x3e	0xb5	0x66	0x48	0x03	0xf6	0x0e
0x61	0x35	0x57	0xb9	0x86	0xc1	0x1d	0x9e
0xe1	0xf8	0x98	0x11	0x69	0xd9	0x8e	0x94
0x9b	0x1e	0x87	0xe9	0xce	0x55	0x28	0xdf
0x8c	0xa1	0x89	0x0d	0xbf	0xe6	0x42	0x68
0x41	0x99	0x2d	0x0f	0xb0	0x54	0xbb	0x16

Table A.1: The AES/Twister S-box

Index