Technical University of Denmark

Cryptanalysis of Block Ciphers

Master's Thesis

Martin M. Lauridsen

✦ Academic Supervisors ✦

Professor Lars R. Knudsen
Associate Professor Gregor Leander

✦ Submission Date ✦

July, 2012

MARTIN M. LAURIDSEN

✦ CRYPTANALYSIS OF BLOCK CIPHERS ✦

CRYPTANALYSIS OF BLOCK CIPHERS

THIS THESIS WAS PREPARED, WRITTEN AND TYPESET BY
Martin M. Lauridsen, s072468

STUDYING AT
The Technical University of Denmark

UNDER SUPERVISION BY
Professor Lars R. Knudsen
Associate Professor Gregor Leander

AFFILIATED WITH
DTU Matematik
Department of Mathematics

Matematiktorvet, building 303 B
DK-2800 Kgs. Lyngby
Denmark

Telephone: (+45) 45 25 30 31
E-mail: mat-instadm@mat.dtu.dk

Project period:	February 6, 2012 – July 16, 2012
Type:	Master's thesis
Study-line:	Mathematical Modelling and Computation
ECTS:	30
Edition:	First

ABSTRACT

This thesis presents an introduction and cryptanalytic treatment of PRINCE, a new lightweight block cipher. PRINCE was designed for the Dutch company NXP Semiconductors by the Crypto Group at the Technical University of Denmark, for use in memory encryption in smaller computing devices.

To provide a basis for the description of PRINCE and the design choices made for its development, we give first an introduction to block ciphers in general, describe the use of lightweight block ciphers and consider their cryptanalysis. Subsequently, we introduce PRINCE by giving its description and presenting arguments for the design decisions. Specifically, we see how PRINCE uses an unrolled design and symmetric arrangement of components, to obtain low latency and overhead of decryption implementation. We also describe a tweak for PRINCE, which allows the encryption of a plaintext to depend on its address in memory. Following the introduction of PRINCE, we give general introductions to the two most effective cryptanalytic attacks on generic block ciphers; differential- and linear cryptanalysis. PRINCE was designed with resistance towards these attacks in mind. Specifically, it applies the wide-trail design strategy, introduced by Rijmen and Daemen, and most notably known from the Advanced Encryption Standard (AES). The purpose of applying this strategy is, to obtain a high degree of diffusion within the cipher components.

The design of PRINCE entails the possibility of reformulation of four rounds of the cipher into a *megabox* description. We use this description to give upper bounds on the probability of characteristics and linear trails, and to look for high-probability differentials and linear hulls. In addition to the megabox analysis, we model the problem of finding the best differentials and linear hulls in the frame of *characteristic trees* and *characteristic graphs*. To those models, we apply search methods and randomized heuristics. We find that latter are especially effective, compared to the megabox approach, for discovering differentials for the tweaked PRINCE. Our results from applying these search methods, argue that PRINCE is insusceptible to differential- and linear attacks.

Finally, we consider two distinguisher attacks on the tweaked PRINCE. The first is a statistical χ^2 analysis, where the attacker is given access to two black boxes; one containing the tweaked PRINCE and one containing a random permutation. Our results indicate that the tweaked PRINCE can not be distinguished from a random permutation using this attack. In another model, we let the attacker control the address at which encryption and decryption takes place. In this case we find that the black boxes can be distinguished, using just six queries to either one of them.

PREFACE

This thesis has been submitted in partial fulfillment of the requirements to receive a Master of Science in Engineering (M.Sc.Eng.) at the Technical University of Denmark in Mathematical Modelling and Computation, under simultaneous compliance with requirements to graduate from the Honors Program of the Technical University of Denmark.

The work detailed herein was carried out in the Spring of 2012, from early February to mid July, and constitutes a total of 30 ECTS for the author, distributed evenly over mentioned period.

The project was carried out under supervision and guidance by Professor Lars R. Knudsen and Associate Professor Gregor Leander of Department of Mathematics, the Technical University of Denmark.

Martin M. Lauridsen

Technical University of Denmark
Kgs. Lyngby
July, 2012

ACKNOWLEDGMENTS

Reaching the finish line of my journey towards completing my Master's degree, there are a number of people to whom I owe my gratitude. First, I would like to thank my supervisors, Professor Lars R. Knudsen and Associate Professor Gregor Leander, for being the ones that kindled my interest in cryptology, starting four years ago. During the project time, their humorous nature, trust and academic direction have helped me tremendously in working towards my goals. Thanks also go out to my former tutor, Søren S. Thomsen, with whom I worked closely on projects, which were a strong motivational factor.

I would also like to mention friends and family, who have always shown a healthy interest in my studies, despite the occasional frown of confusion when the word *cryptology* is brought up. Special thanks go to my girlfriend for her tireless interest and suggestions, moral support and understanding. Finally, I wish to express my gratitude to my parents, for always having supported me in my studies and for having taught me to do my best.

CONTENTS

List of Figures

CHAPTER 1

INTRODUCTION

In 1972, the National Bureau of Standards (NBS) – now the National Institute of Standards and Technology (NIST) – started a program for the protection of governmental data. As part of the program, a request for an encryption method to be standardized, was issued. The algorithm which was eventually adapted as the Data Encryption Standard (DES), was a proposal by IBM in 1974, based on the design of Horst Feistel's Lucifer cipher [28].

The study of the DES has given rise to many revolutionary ideas in cryptanalysis, most notably differential cryptanalysis discovered by Biham and Shamir in the late 1980s, and linear cryptanalysis discovered by Matsui in the early 1990s. Due to the advances in cryptanalysis during the past decades, the DES is no longer considered secure in many applications.

To remedy the insecurity of the DES, NIST announced in 1997 the wish to choose its successor, to be named the Advanced Encryption Standard (AES). Many submissions for cipher suggestions were received, and after several rounds, the Rijndael cipher [7], developed by Rijmen and Daemen, was chosen for the AES in 2001. Since, the AES has superseded the DES as the most used block cipher worldwide. Despite the extensive amount of cryptanalysis performed on the AES, it is still considered secure by current standards.

While the AES will probably be considered secure for years to come, there are applications where the AES is not the block cipher of choice. During the past decade, there has been an increasing ubiquity of smaller computing devices, such as credit cards, smart phones, tablets and passive RFID tags. Applications include road tolling devices, medical implants and biometric passports. The need for encryption of data in these applications has begotten the class of *lightweight* block ciphers.

Lightweight block ciphers are tailored for applications in hardware devices which are restricted with respect to a number of things, e.g. chip area and power consumption. The AES does not fall into this category. Due to the lack of research in the area, early lightweight block ciphers have primarily been proprietary algorithms. However, with the standardization of the PRESENT [5] and CLEFIA [27], the area is not starting to receive the required attention.

This thesis evolves around a lightweight block cipher PRINCE, which was developed by the Crypto Group at the Technical University of Denmark. PRINCE was made for the Dutch company NXP Semiconductors, for the use of memory encryption in small computing devices. Especially, we focus on cryptanalysis of PRINCE, with the intent

to argue the security of the cipher towards attacks, especially from differential- and linear cryptanalysis.

In practice, the size of implementation for lightweight block ciphers is not always as relevant compared to other metrics. Most lightweight block ciphers seen thus far have focused on two objectives; security and minimizing chip area. The development of PRINCE was motivated by the purpose of breaking with this trend. As we shall see, PRINCE is an exciting new lightweight block cipher, which uses an unrolled and symmetrical design, to minimize latency and allow implementation of decryption using a minimal overhead.

1.1 Notation

One of the wonders of cryptology is, that one can do a fair amount of research in the area, without plowing through the Latin, Greek and Cyrillic alphabets in a single article. Notation will be introduced as we progress through the chapters. At this point, we merely state that we shall use `typewriter font` for numbers in base 16 (hexadecimal) notation.

CHAPTER 2

BLOCK CIPHERS

This chapter serves as an introduction to block ciphers from a very general point of view. Understanding the concepts in this chapter provides a basis for the case study of the lightweight block cipher PRINCE, which we introduce in Chapter 3.

First, we discuss the purpose, properties and classification of block ciphers. We then talk about design principles in Section 2.4. Section 2.5 elaborates on what a lightweight block cipher is, and what the goals and challenges are, within this field of study. Cryptanalysis of block ciphers is discussed in Section 2.6.

2.1 Purpose

In the world of cryptography, the goal of *confidentiality* is to disguise a secret message in a way, such that only the message *sender* and its *intended recipient*, are able to know or recover the true message. For anyone else, the garbled message, or *cryptogram*, should look like nonsense. A block cipher is a cryptographic primitive – a sort of building block used in larger applications – for providing, among other things, confidentiality.

Block ciphers are named so, because they operate on *blocks* or *states* of data at a time. Often, the data is binary (i.e. a string of ones and zeros), and hence the blocks are binary strings. The blocks have a fixed size of b bit, where typically $b \in \{64, 128, 256\}$. The plaintext is split into blocks of b bit, while applying a padding scheme to make sure the message constitutes a whole number of blocks, and is encrypted one block at a time.

2.2 Classification

Cryptographic algorithms can be divided into a number of classes. Block ciphers are a type of algorithm in the class of *symmetric-key* algorithms, which contains those algorithms where communicating parties share the same, secret key.

Block ciphers generally have a very good performance when implemented in either hardware or software. However, an obvious obstacle is the need of establishing in a confidential manner a shared, secret key between communicating parties. Thankfully, another class of cryptographic algorithms called *public-key* algorithms, are good at solving this task. These algorithms utilize mathematical structures in a clever way that

allows parties to share with each other the public part of a key, to construct a shared, private key. As block ciphers outperform public-key algorithms, these take over and perform the heavy workload once the key is established [16, Sec. 1.1].

We continue our introduction of block ciphers by presenting their formal properties.

2.3 Properties

A block cipher can be considered as an encryption function

$$E_k : \mathbb{F}_2^b \longrightarrow \mathbb{F}_2^b.$$

The subscript k denotes the secret key, which is shared by communicating parties. We say that the block cipher encrypts plaintexts *under the secret* key k. As a block cipher, the function E_k is required to be a bijection, and since the domain and co-domain are the same, it is in fact a permutation.

To restore the plaintext from a ciphertext, a decryption function D_k is needed. As E_k is a permutation, the decryption function is simply defined as the inverse of the encryption permutation,

$$D_k = E_k^{-1}.$$

Using these two functions, one may encrypt a plaintext message m to obtain a ciphertext c as

$$c = E_k(m)$$

and recover the message as

$$m = D_k(c).$$

As the block cipher E_k operates on b-bit blocks, there are 2^b distinct blocks in its domain and co-domain $\mathbb{F}_2^b$. This, in turn means there are $(2^b)!$ distinct permutations on this set, each representing a potential block cipher. For typical values of b, this is an astronomical amount. However, not all these permutations are used. Since the κ-bit key k allows for 2^κ possible choices, this is in fact the actual number of permutation candidates for the block cipher. The choice of the secret key k represents a choice of a particular block cipher to use, among the vast number of candidates, and under a fixed k the block cipher E_k is completely deterministic.

An important task for a block cipher designer is ascertaining that for any choice of key $k \in \mathbb{F}_2^\kappa$, the permutation E_k should appear to have been chosen uniformly at random from the $(2^b)!$ possible ones.

2.4 Design Principles

Two important objectives in block ciphers are that of *confusion* and *diffusion*, introduced by Shannon in his pioneering work of 1949 entitled *Communication Theory of Secrecy Systems* [26]. The purpose of confusion is to eliminate known plaintext statistics in the encrypted ciphertext. The goal of introducing diffusion to a block cipher, is to complicate, for the attacker, the relation between plaintext bits and key bits in the ciphertext. It is clear that confusion and diffusion are highly desirable properties for a block cipher, and it is the task for the designer to determine how to obtain them, while still keeping the requirements for the block cipher in mind.

2.4.1 Components and Layers

Block ciphers are often composed of *components* or *layers* applied to the b-bit block in a certain order. Naturally, these components need to be permutations themselves for the whole block cipher E_k to be one. Otherwise, encryption and decryption would no longer be well defined. Often, these components of different types are assembled into larger components called *rounds* or *round functions*, denoted $\mathscr{R}$. Typically, the only difference between two round functions $\mathscr{R}_i$ and $\mathscr{R}_j$ used in the same block cipher is, that they use different keys k_i and k_j respectively. Such keys are referred to as *round keys*. A block cipher can consist of some fixed number of such round functions applied in succession. In this case, we speak of an *iterated* block cipher.

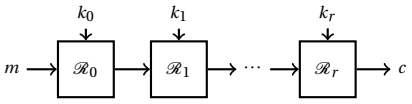


Figure 2.1: Illustration of iterated block cipher using r consecutive round functions $\mathscr{R}_i$, $0 \le i \le r$. Each round function R_i takes as input a round key k_i.

Figure 2.1 shows a generalized example of an iterated r-round block cipher. Each round function R_i is identical, except for the round key k_i taken as input parameter. The most fruitful gain from such a simple design is the implications on cryptanalysis of such a cipher. If the cryptanalyst can analyze and understand the round function, then there is a good chance that something can be deduced about the whole block cipher, based on this knowledge.

2.4.2 Obtaining Confusion and Diffusion

The most commonly used mechanism to provide confusion in a block cipher is a *substitution layer*. The substitution layer is often implemented by a u-bit *substitution box* or *S-box* for short. The S-box replaces each of the $\left(\frac{b}{u}\right)$ consecutive u-bit sub-blocks of the current b-bit state with another u-bit value. In the case of $u = 8$ we call the sub-block a *byte*, and when $u = 4$, which is the case for the PRINCE cipher, we call it a *nibble*.

The choice of S-box can have a great impact on the security and performance of the block cipher [22, 11]. The block cipher designer must be careful in choosing the S-box(es) best suited, under the relevant constraints and requirements. This is always an important task, but in lightweight block ciphers, such as PRINCE, the efficiency of the S-box(es) is of special importance. PRINCE uses a single S-box, but sometimes, as is the case with the Data Encryption Standard (DES), a block cipher uses several different S-boxes $S_0, \ldots, S_k$. Also, as demonstrated by Matsui in [22], using DES as an example, the order of the S-boxes can have an impact on the strength of the cipher.

To ensure diffusion, block ciphers frequently make use of one or more different permutation layers P_i, that simply permute the bits of the block. This might be done on bit level, which is the case for DES [23, 16]. In this case, the block cipher is well suited for hardware implementation. The permutation might also work on u-bit sub-blocks of the b-bit block at a time, just as the S-box does, which might be better suited for software implementation. As an example, consider the AES [7], where optimizations can be made when merging cipher components into four 32-bit tables of 256 entries each. In this case, the AES works with $u = 8 \cdot 4 = 32$ bit words at a time.

Another component often used for providing diffusion is a linear mixing layer, which typically multiplies the b-bit state with a matrix having certain desirable properties. As we shall see in Chapter 3, this is the case with the lightweight cipher PRINCE.

2.4.3 SP-networks and Feistel Ciphers

An important type of iterated block ciphers, termed *substitution-permutation network* or *SP-network* for short, use a design consisting solely of substitution and permutation layers, as well as key addition in each round. These *round keys* are deduced using a process called *key scheduling*, which takes as input the secret key $k \in \mathbb{F}_2^\kappa$. The key schedule is the first step in using the block cipher, both when encrypting and decrypting, and is performed just once by each communicating party. The most well-known and well-studied example of an SP-network is the *Rijndael* block cipher [7], developed by Vincent Rijmen and Joan Daemen[1]. Rijndael won the AES competition announced by NIST[2], and was adopted as a standard in 2001.

Another type of iterated block cipher is a *Feistel* cipher after German physicist Horst Feistel. In a Feistel cipher, the plaintext is $2b$-bit and is broken into parts (L_0, R_0). Over r rounds, the cipher computes a ciphertext (R_r, L_r). A property of a Feistel cipher is, that decryption uses the same r-round process as encryption, but with the order of the round-keys reversed [23]. This concept is similar to what happens in the case of PRINCE, as we shall see in Chapter 3. Another feature of a Feistel cipher is, that only half of the state is operated on in each round, while the other part goes unchanged to the next round. The most prominent example of a Feistel cipher is DES [23, 16].

2.5 Lightweight Block Ciphers

The concept of *lightweight* block ciphers is a fairly new player in the world of cryptology. As with all other classes of cryptographic algorithms, their existence is the consequence of a need of them in applications that are emerging due to new technology.

During the past decades, block ciphers have been used primarily by for applications in (relatively) powerful machines. By a powerful machine, as of today standards, we mean e.g. a desktop computer. With the advent of the Internet in the 1990s, block ciphers starting becoming used on massive scale in various communication protocols on personal computers worldwide, due to cryptographic objectives in the context of communication needing fulfillment. With the increasing ubiquity of smaller computing devices, there is a rising demand for lightweight block ciphers, i.e. block ciphers tailored for use in resource-constrained devices. Examples of such devices include wireless RFID[3] tags, smart cards, medical implants, road-tolling devices and wireless sensor networks.

Traditionally, the metric on which lightweight block ciphers have been measured, other than their security, is the chip area required for a hardware implementation of the cipher. However, there are applications where other metrics, e.g. latency, throughput or power consumption are more relevant. Thus, the designer of a lightweight block cipher faces the challenge of choosing components for the cipher that meet, to their best ability, a potentially complicated combination of constraints and requirements.

[1] Rijndael is a contraction of the authors' names.

[2] National Institute of Standards and Technology.

[3] Radio Frequency Identification.

For a survey of implementations of prominent lightweight ciphers, see [12, 5, 19, 10].

2.6 Cryptanalysis of Block Ciphers

Using a block cipher which has not been properly cryptanalyzed can have dire consequences. It is essential that block ciphers adopted to standards are resistant to attacks. The process of arguing about the strength of a block cipher is what *cryptanalysis* is all about. In this section we give a general introduction to cryptanalysis with focus on block ciphers. As this thesis is an application of cryptanalysis to the lightweight block cipher PRINCE, the following chapters will provide more in-depth descriptions and applications of cryptanalytic techniques.

2.6.1 Assumptions and Types of Attack

When assessing the strength of a block cipher, the cryptanalyst works under two basic assumptions [23]:

1. The attacker has access to listen to encrypted communication which takes place over an insecure channel. This basic concept is illustrated in Figure 2.2. Alice computes a ciphertext c as the encryption of a message m, using encryption function E_k. She sends c to Bob over an insecure channel, where the attacker Eve is listening to the encrypted communication. Bob can compute the message m, using the decryption function D_k, because he too knows the secret key k.
2. The complete description of the cipher used, except the private key k, is available to the attacker. This fundamental assumption is known as *Kerckhoffs' principle* [23, 16]. Another interpretation of this principle is, that the strength of a cipher must *never* depend on an obfuscated design.

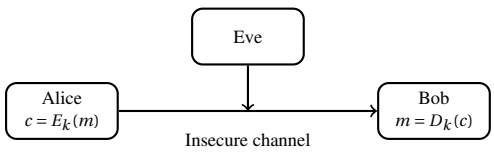


Figure 2.2: Basic setup of symmetric encryption.

In cryptanalysis, several types of attacks are considered. The most fatal type of attack is one in which the attacker (Eve, c.f. Figure 2.2) is able to retrieve the secret key k. If this should happen, Eve is able to retrieve all plaintexts for ciphertexts observed and recorded, or seen in the future, using the same secret key. Also, Eve can intercept ciphertexts and send her own encrypted messages as replacement. For example, Alice needs to make a transaction to send some money to Bob, and asks Bob for his bank account number. When Bob replies, Eve intercepts the message and sends her own bank account details to Alice. Alice thinks the reply came from Bob, and sends the money to Eve, thinking she sent them to Bob.

While key recovery attack, known as a *total break* is clearly fatal, there are other types of breaks of less significance that are still undesirable. One may list common types of breaks in order of their severity [16]:

1. **Total break** where the attacker recovers the whole secret key k.
2. **Global deduction** where the attacker is able to find efficient functions E'_k and D'_k, which yield the same results as E_k and D_k.
3. **Local deduction** where the attacker can encrypt/decrypt previously unseen messages/ciphertexts.
4. **Distinguisher algorithm** where the attacker has access to two black-boxes. One contains an implementation of the block cipher and the other contains a randomly chosen permutation on $\mathbb{F}_2^b$. A distinguisher attack attempts to effectively determine which box contains the block cipher.

From the listing above, it should be clear that a total break implies the other types of break, while it does not hold the other way around. Having classified types of breaks of block ciphers, we turn to the types of attack models.

2.6.2 Cryptanalytic Attack Models

The purpose of an attack is generally to retrieve (parts of) the plaintext from a given ciphertext. Under the assumptions above, the evaluation of the cipher can be analyzed under the following models of attack:

- **Ciphertext-only attack**, where the attacker knows just the ciphertexts observed from the communication channel (plus any knowledge of the type of data being transmitted, especially plaintext statistics).
- **Known-plaintext attack**, where the attacker knows a set of plaintext and ciphertext pairs (m_i, c_i).
- **Chosen-plaintext attack**, also known as the *lunchtime attack*[4]. In this model, the attacker can (for a limited time) request plaintext-ciphertext pairs (m_i, c_i) where m_i is of the attacker's choosing and afterwards utilize these in an attack.
- **Adaptive chosen-plaintext attack**, which is very similar to the prior, except the choice of m_i may depend on the choice of previous plaintext-ciphertext pairs.
- **Chosen-ciphertext attack**, where the attacker has access to a certain number of plaintext-ciphertext pairs for ciphertexts of his choice.
- **Adaptive chosen-ciphertext attack**, which is like the prior attack, except the choice of ciphertexts, for which the attacker is given the corresponding plaintext, may depend on previously chosen ciphertexts.

Note that as we describe the classes above, the strength of the assumptions about the power of the attacker increases, and thus a block cipher which is secure against an attack in the list, is also secure against the others above classes of attacks above it.

2.6.3 Complexity of Attacks

To evaluate the strength of an attack on a block cipher, one typically uses either data complexity (the amount of data required), memory complexity (memory required) or computational complexity (the number of operations, e.g. encryptions, required). Note, that memory complexity can be of great importance for an attack, as with modern computers, input/output operations are often a bottleneck. For a block cipher to be considered secure, it should be infeasible to amount the smallest of the complexities

[4]This name comes from the idea, that the attacker has access to encrypt a set of chosen plaintexts, if the person under attacks leaves his computer unlocked, when going to lunch.

required for an attack. Also, the key length κ should meet some minimum size, because if κ is chosen too small, the attacker may simply try all keys to find the correct one. In this sense, the key length κ always defines an upper bound on the attack complexity.

The type of attack mentioned above is the simplest of them all; the *exhaustive search* attack. Exhaustive search is currently not a threat to any modern block ciphers where $\kappa \geq 64$. Note, that the exhaustive search is an example of known-plaintext attack or known-ciphertext attack, because either the attacker needs plaintext-ciphertext pairs, or must rely on plaintext statistics, to know when the correct key is found. In the known-plaintext case, with about $\lfloor (k+4)/n \rfloor$ plaintext-ciphertext pairs, the *expected* computational complexity of breaking the system is $O(2^{k-1})$ [23, 16]. Note, however, that when mounting an exhaustive search attack, the attacker needs to run the key scheduling for each new key attempted, and this may have a significant impact on the actual time of the attack.

2.7 Summary

In this chapter, we have given a very general introduction to block ciphers. We described their properties, especially from a mathematical point of view, which eases notation for us in the remainder of the thesis. Also, we described shortly the purpose of block ciphers in cryptology.

The main focus of the chapter was design aspects, the concept of a lightweight block cipher, and cryptanalysis. The reason the different components of a block cipher were described in depth is, that it aids the understanding of choices made, when we describe the lightweight block cipher PRINCE in Chapter 3.

Having introduced lightweight block ciphers, we know that for their applications in very resource-constrained environments, the requirements to the block cipher and metrics used play a great role. As we shall see, PRINCE is a step towards lightweight block cipher design for other metrics than just minimizing the chip area.

Finally, our introduction of cryptanalysis will be helpful when we discuss our analysis of PRINCE in Chapters 4 through 10.

CHAPTER 3

THE BLOCK CIPHER PRINCE

In this chapter we present the lightweight block cipher PRINCE [11]. The cipher is an *ultra* lightweight cipher, in the sense that it is tailored for applications in *very* resource restricted environments. The cipher was developed for the Dutch company NXP Semiconductors, by the Crypto Group at the Technical University of Denmark, for use in memory encryption. The cipher is also part of a larger research project on lightweight cryptology, conducted by the Crypto Group.

3.1 Motivation and Design Goals

In this section, we present the motivation behind the development of the PRINCE cipher, and what the design goals are.

To quote the designers, the goal of designing PRINCE is to obtain a block cipher, which is optimized towards meeting the following four criteria, when implemented in hardware [11]:

1. The block cipher should be able to perform *instantaneous* encryption and decryption, i.e. within a single clock cycle, when implemented in hardware.
2. A high clock rate should be achievable, when implemented in a modern chip technology.
3. The costs of hardware implementation should be moderate.
4. Encryption and decryption should be supported with a low overhead.

Criteria 4 above suggests that the fraction of circuit usable by both the encryption and decryption process should be as high as possible. Indeed, this is a major design criteria for the PRINCE block cipher and, as we shall see, the encryption and decryption process uses the same circuit, only with a slightly different key schedule. This criteria is important, because if the block cipher should support instantaneous encryption/decryption (Criteria 1), it needs to be *unrolled*. This means the entire encryption/decryption process must be implemented in the circuit, without values having to be fed back to reapply the same circuit. If the fraction of encryption circuit usable by decryption is low, this implies much increased chip area.

3.1.1 Design Choices

The completely unrolled design of PRINCE implies a much higher level of flexibility, and in particular, the block cipher need not be iterated with a fixed round function $\mathscr{R}$. However, as described, iterated block ciphers are very convenient with respect to cryptanalysis, in the sense that a round function can be analyzed, and conclusions about the strength of the whole ciphers can be made on this basis. With an unrolled block cipher, this is no longer necessarily true, and can hinder the process of showing the strength of a cipher. Indeed, the trust in the security strength of a cipher should always be based upon thorough cryptanalysis, and *never* upon an obfuscated design. For that reason, the choice has been made to have PRINCE be an iterated block cipher with an unrolled implementation.

For PRINCE, the choice has fallen upon an SP-network rather than a Feistel cipher, as the latter operates only on half of the state in each round. This would likely require a higher number of rounds, which is undesirable for a lightweight block cipher [11].

In order to make PRINCE resistant towards differential- and linear attacks (introduced in Chapters 4 and 5 respectively), the design adopts the *wide-trail strategy*, introduced by Daemen and Rijmen, and most prominently known for its use in the AES [7, 8]. The wide-trail design strategy for block ciphers dictates that the round function $\mathscr{R}$ should be constructed in a away, such that it becomes resistant towards differential- and linear cryptanalysis, while still being efficient. This strategy allows to state a lower bound on the diffusion properties of PRINCE over a certain number of rounds, as we shall see in Chapter 6.

3.2 Description of PRINCE

This section presents the description of the PRINCE block cipher. We introduce the cipher in a top-down fashion, by first giving a full overview of the structure (of the encryption process), and then digging into the details of each component as we progress.

3.2.1 Encryption Overview

PRINCE is a $b = 64$-bit iterated block cipher using a private key k of $\kappa = 128$ bit. The key scheduling process splits up k into two parts of 64 bit each, and expands the key to 192 bit total, as illustrated by Figure 3.1.

Figure 3.1: PRINCE key schedule.

The mapping $(k_0 \| k_1) \mapsto (k_0 \| k_0' \| k_1)$ of Figure 3.1 is defined by

$$(k_0 \| k_1) \mapsto (k_0 \| \ (k_0 \ggg 1) \oplus (k_0 \gg 63) \ \| k_1). \tag{3.1}$$

The $\ggg$ symbol of (3.1) denotes a cyclic bitwise right-shift. For example, $1011_2 \ggg 3 = 0111_2$.

With the key schedule in place, the overall encryption process of a 64-bit plaintext block m is depicted in Figure 3.2. The sub-key k_0 is added to m, after which a component, PRINCE_{core} to be described in the following, is applied. To the output of PRINCE_{core} the sub-key k_0' is added, to produce the ciphertext c.

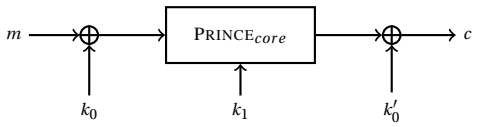


Figure 3.2: Overall encryption using PRINCE.

Note also from Figure 3.2, how the sub-keys are used: first k_0, then k_1 for the PRINCE$_{core}$ component, and then finally k_0'. This justifies the choice of key expansion, as the attacker will now have to guess not 64 but 128 bit, to try and attack either the first two or last two rounds. Also, as we shall see, the specific choice of key expansion makes it possible to decrypt using the same circuit with a simple modification to the key k.

3.2.2 The PRINCE$_{core}$ Component

As was illustrated in Figure 3.2, the overall encryption uses a component, PRINCE$_{core}$ which we turn to describing now. This core component contains mainly applications of the PRINCE round function $\mathscr{R}$ and its inverse $\mathscr{R}^{-1}$.

The PRINCE$_{core}$ component is depicted in Figure 3.3. In the beginning, the 64-bit sub-key k_1 is added, along with a round constant RC_0. Then follows five applications of the round function $\mathscr{R}_i$, $1 \le i \le 5$, which uses the sub-key k_1 and a round constant RC_i. After those, what we will be referring to as the PRINCE *middle part* follows. The middle part is its own inverse, and uses two S-box applications and a linear transformation in the very middle. The middle part is defined by

$$S^{-1} \circ M' \circ S. \tag{3.2}$$

Then follows five applications of the inverse round function $\mathscr{R}_j^{-1}$, $6 \le j \le 10$, using again the sub-key k_1 and round constants RC_j. Finally, the sub-key k_1 is added again along with a round constant RC_{11}.

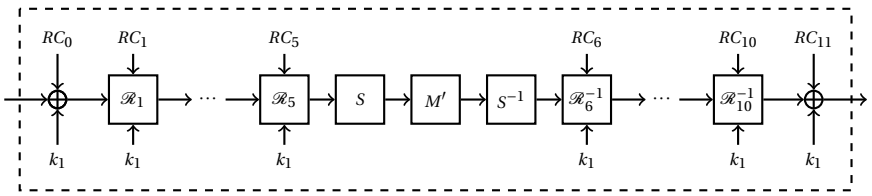


Figure 3.3: The PRINCE$_{core}$ component.

Note from Figure 3.3, that the component is totally symmetric around the middle part, and indeed the very middle component, M', is its own inverse. Due to this fact, and the choice of round constants RC_i, $0 \le i \le 11$, to be specified later, the decryption process uses the exact same process as encryption, only with a different key schedule. The implication of this is, that decryption and encryption can be implemented in hardware with the different key schedule being the only overhead, thus fulfilling Criteria 4 of Section 3.1.

3.2.3 Decryption

As mentioned, the design of PRINCE is such that the construction in Figure 3.2 is used for both encryption and decryption. For this to be possible, the round constants must, in a pairwise manner, add up to the same value. Specifically, it must hold that

$$RC_i \oplus RC_{11-i} = \alpha, \qquad 0 \leq i \leq 11,$$

for some 64-bit constant α. In the case of PRINCE, the choice has fallen upon

$$\alpha = \texttt{c0ac29b7c97c50dd}.$$

The choice of the specific round constants will be explained later. As a consequence of this construction, the action of PRINCE_{core} using key parameter k_1 is equivalent to the action of the inverse of PRINCE_{core} using key parameter $k_1 \oplus \alpha$. This property, referred to as the *α-reflection property* by the designers [11], is essential to the design choice of virtually no overhead for decryption.

Also, the very middle component, the M'-layer, has to be an involution. This implies decryption can use the exact same circuit as encryption, with just a minor change to the master key, such that decryption is defined by

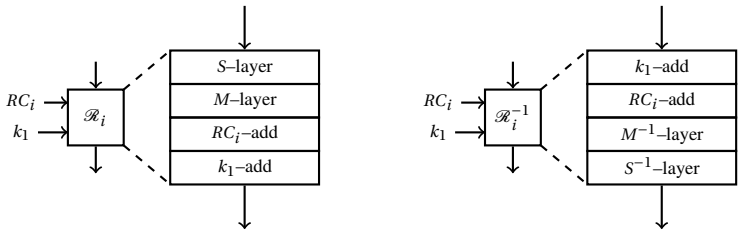
$$D_{(k_0 \| k_0' \| k_1)} = E_{(k_0' \| k_0 \| k_1 \oplus \alpha)}.$$

Having explained how the design of PRINCE manages to accomplish Criteria 4 of the design goals, we continue to describe the PRINCE round function $\mathscr{R}$, which is where diffusion and confusion for the block cipher is obtained.

3.2.4 Round Function

From the description of PRINCE_{core} of Figure 3.3, we see that the encryption consists of ten applications of $\mathscr{R}_i$ or $\mathscr{R}_i^{-1}$ with $i \in \{1, \ldots, 10\}$. Figure 3.4 shows the components making up an individual round $\mathscr{R}_i$ and its inverse $\mathscr{R}_i^{-1}$. A single round consists of:

- A non-linear substitution layer S or S^{-1}, implemented by a single $u = 4$-bit S-box,
- A linear layer M or M^{-1}, and
- Addition of sub-key k_1 and round constant RC_i.



(a) Round $\mathscr{R}_i$. **(b)** Inverse round $\mathscr{R}_i^{-1}$.

Figure 3.4: Description of the PRINCE round function and its inverse.

Having presented the components making up the PRINCE round function, we turn our attention to describing these components next.

S-box Component

For the non-linear substitution layer S, PRINCE uses a single 4-bit S-box. Being a lightweight cipher, focus must be kept on the S-box, as to minimize the chip area needed for implementation, as well as optimizing the critical path of the circuit required for implementation [11].

As the hardware technology used can have a substantial effect on this, one *best* S-box can not be guaranteed to exist. To that end, PRINCE allows flexibility in the choice of S-box, as long as it fulfills the following criteria [11]:

1. The largest differential probability should be $\frac{1}{4}$. The concept of differentials is introduced in Chapter 4.
2. There should be exactly 15 differentials with probability $\frac{1}{4}$.
3. The largest absolute value of any linear approximation bias should be $\frac{1}{4}$. We introduce linear approximations in Chapter 5.
4. There are exactly 30 linear approximations with absolute bias 1/4.
5. Each of the 15 non-zero component functions have algebraic degree 3. For an n-bit S-box, it generally holds that the maximal algebraic degree is $n-1$. The wish to keep the algebraic degree maximal is, in part, to provide resistance towards a type of differential cryptanalysis called *higher-order differential cryptanalysis*, which we will not be considering in this thesis.

A study of 4-bit S-boxes conducted in [18] shows there are just eight 4-bit S-boxes fulfilling the criteria above, up to affine equivalence [11]. One of these S-boxes is given in Table 3.1. This is the S-box we will use for our further studies and implementation of PRINCE in this thesis.

x	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
$S[x]$	b	f	3	2	a	c	9	1	6	7	8	0	e	5	d	a

Table 3.1: The 4-bit PRINCE S-box.

Note that since PRINCE is a 64-bit block cipher, and $Dom(S) = \mathbb{F}_2^4$, the S-layer of PRINCE must apply the S-box to the 64-bit block 16 times in parallel.

Linear Components

In the two linear layers, the 64-bit state is multiplied with a 64×64 matrix M, respectively M', over $\mathbb{F}_2$. The entirety of M and M' can be found in Appendix A, however their construction is described below.

The M' matrix Since the M' component is only used in the middle of what we refer to as the middle part of PRINCE, c.f. Figure 3.3, M' must be an involution, i.e. a function which is its own inverse, in order to maintain the α-reflection property [11].

As PRINCE is a lightweight block cipher, focus must be kept on implementation costs. To that end, M' should be sparse to meet this demand. Another requirement on M' is, that it should provide a certain degree of mixing of the state nibbles. In accordance with the wide-trail design strategy, M' has been chosen such that there will be at least 16 active S-boxes, i.e. input nibbles to S which are non-zero, over 4 rounds. To achieve this effect, M' is constructed such that the Hamming weight of any row or column is at least 3, in order for each output bit of an S-box to affect 3 S-boxes in the

next round [11]. The M' matrix can be constructed as follows. We define four 4×4 matrices over $\mathbb{F}_2$:

$$M_0 = \begin{bmatrix} 0000 \\ 0100 \\ 0010 \\ 0001 \end{bmatrix}, \quad M_1 = \begin{bmatrix} 1000 \\ 0000 \\ 0010 \\ 0001 \end{bmatrix}, \quad M_2 = \begin{bmatrix} 1000 \\ 0100 \\ 0000 \\ 0001 \end{bmatrix}, \quad M_3 = \begin{bmatrix} 1000 \\ 0100 \\ 0010 \\ 0000 \end{bmatrix}.$$

Using these, we may construct two 16×16 matrices over $\mathbb{F}_2$:

$$\hat{M}^{(0)} = \begin{bmatrix} M_0 & M_1 & M_2 & M_3 \\ M_1 & M_2 & M_3 & M_0 \\ M_2 & M_3 & M_0 & M_1 \\ M_3 & M_0 & M_1 & M_2 \end{bmatrix}, \qquad \hat{M}^{(1)} = \begin{bmatrix} M_1 & M_2 & M_3 & M_0 \\ M_2 & M_3 & M_0 & M_1 \\ M_3 & M_0 & M_1 & M_2 \\ M_0 & M_1 & M_2 & M_3 \end{bmatrix}.$$

Finally, using these two matrices, we can define the 64×64 matrix M' having the matrices $\left(\hat{M}^{(0)}, \hat{M}^{(1)}, \hat{M}^{(1)}, \hat{M}^{(0)}\right)$ in as diagonal blocks, with zeros elsewhere. M' is symmetric and is its own inverse, i.e. an involution.

The M matrix As the M-layer is not used in the middle part of PRINCE$_{core}$, it need not be an involution. What we desire from this layer is to have each output bit from an S-box affect as many input bits to the S-box of the next round, as possible. Recalling the definition of diffusion from Chapter 2, this is the way the PRINCE block cipher provides this desirable property.

The M-layer is constructed using the M'-layer described above and an operation SR (short for *shift rows*), such that

$$M = SR \circ M'.$$

The SR operation permutes the state nibbles according to the following permutation, where 0 is the most significant 4-bit nibble of the state:

$$SR = (1\ 13)(2\ 10)(3\ 7)(5\ 1)(6\ 14)(7\ 11)(9\ 5)(10\ 2)(11\ 15)(13\ 9)(14\ 6)(15\ 3).$$

The name *shift rows* comes from the similarity with the AES *ShiftRows* operation [7], and informally if the 16 nibbles are numbered 0 through 15 from most- to least significant, and arranged in a 4×4 matrix state, the permutation operates as indicated in Figure 3.5.

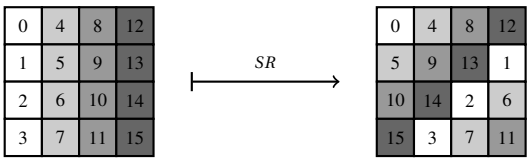


Figure 3.5: Illustration of the effect of SR upon a 64-bit state. Row i is shifted cyclically i positions left, $0 \leq i \leq 3$.

Round Constant Addition

The round constants added in each round of PRINCE$_{core}$ are given in Table 3.2. As mentioned, it holds that

$$RC_i \oplus RC_{11-i} = \alpha, \qquad 0 \leq i \leq 11,$$

with $\alpha =$ c0ac29b7c97c50dd. Note that $RC_0 = 0$, $RC_{11} = \alpha$, and it holds that

$$RC_1 \parallel RC_2 \parallel RC_3 \parallel RC_4 \parallel RC_5 \parallel \alpha$$

corresponds to the first binary digits of π [11]. As such, the round constants are a demonstration of so-called *nothing-up-my-sleeve numbers*, to show that their choice hold no hidden properties. This is very similar to what is done in the key schedule of the Blowfish block cipher [24].

i	RC_i
0	0000000000000000
1	13198a2e03707344
2	a4093822299f31d0
3	082efa98ec4e6c89
4	452821e638d01377
5	be5466cf34e90c6c
6	7ef84f78fd955cb1
7	85840851f1ac43aa
8	c882d32f25323c54
9	64a51195e0e3610d
10	d3b5a399ca0c2399
11	c0ac29b7c97c50dd

Table 3.2: Round constants used by PRINCE$_{core}$.

Round Key Addition

In this part of the round function, the 64-bit round key k_1 is added to the state by the $\oplus$ operation. Note that in the PRINCE$_{core}$ component, the same round key k_1 is used for all rounds. This choice is the result of a tradeoff towards smaller and more efficient implementation, the block cipher being ultra lightweight.

3.3 Implementation and Performance

Let us remind that PRINCE is an ultra lightweight cipher designed for low-cost hardware implementation. However, it does make sense when designing a lightweight block cipher to implement it in software, regardless of the application it was meant for.

It is common practice to start the implementation by a so called *reference implementation*, which follows the definition of the cipher closely. That is, the cipher *primitives*, e.g. the *SR* and S-box operations in the case of PRINCE, are implemented as stand-alone operations. These operations are used as pieces in larger operations, e.g. a round function or the inverse round function.

The reference implementation should be performed very carefully and adhere closely to the cipher definition. Furthermore, it is important that each component of the implementation is tested thoroughly to match the results expected from calculations done by hand or mathematical software. For PRINCE, a sample test could be that a certain set of 64-bit states yield the expected values, when multiplied by the M' matrix.

A reference implementation serves mainly two purposes; one is to give a nice quick introduction to the cipher, to anyone who has not touched upon the design of the cipher; the second is implied by the name, to serve as a reference for other implementations, e.g. a hardware implementation, and to provide test vectors. While the reference implementation is often very manageable when it comes to getting a quick overview, due to its partitioning into meaningful bits, an optimized implementation can be quite the opposite.

3.4 Round-Reduced PRINCE

When considering the strength of an iterated block cipher, cryptanalysts are commonly not able to break the whole cipher, even for those that are just mediocre. Instead, practice is to make statements about *round-reduced* versions of the cipher. For example, an attack on block cipher BC states that s number of rounds for BC are broken, where s is strictly less than the number of rounds r of BC. In the following chapters, we will be considering attacks and cryptanalysis of a reduced number of rounds for PRINCE. Most commonly, the number of rounds considered will be 4. This is due to a way we can describe four rounds of encryption using PRINCE, as we shall see in Chapter 6.

Due to the special symmetric structure and the α-reflection property of PRINCE, a (functional) round-reduced version of PRINCE must necessarily peel off rounds both in the beginning and the end, in order to keep these properties intact. Thus, the number of rounds reduced for PRINCE will always be a multiple of two. Having said that, it is still interesting to consider, for example, the properties of the first four rounds or the last four rounds of PRINCE_{core}, in order to give bounds on the strength of the full cipher, and indeed this shall be one of our main focuses in this thesis.

3.5 Address Encryption Tweak for PRINCE

Recall that PRINCE was designed for NXP Semiconductors for uses in memory encryption. During the development of PRINCE, it was the wish of NXP Semiconductors that an extension should allow the encryption of a certain plaintext to depend on the memory location, i.e. the address, of the plaintext. Thus, the ciphertext c_0 for plaintext-address pair (m, a_0) should be different from (m, a_1) when $a_0 \neq a_1$:

$$E_k(m, a_0) \neq E_k(m, a_1), \qquad \text{for } a_0 \neq a_1.$$

To that end, what we will refer to as a *address tweak* was designed for PRINCE, to accomplish this goal. The tweak is designed such that it can be added without changing the description of the rest of PRINCE, c.f. the above. As such, the tweak can be thought of as an extra module that can be included per wish. When the address tweak is added to PRINCE, we refer to this as the *tweaked* PRINCE.

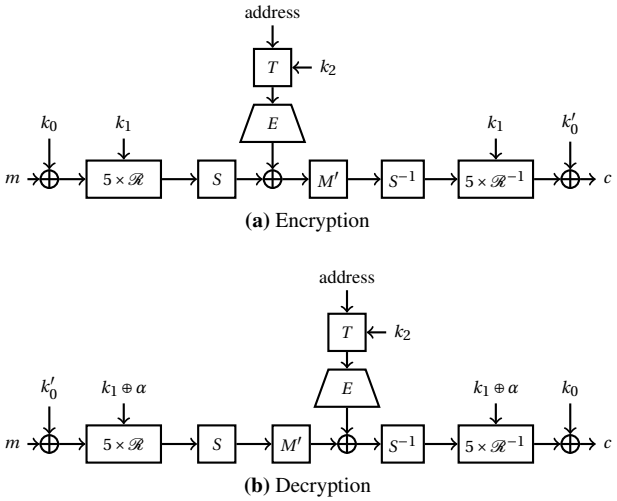
3.5.1 Tweak Description

For the address tweak, the assumption is that addresses are 27-bit values in the application. The overall idea of the address tweak is to encrypt the address using a block cipher T and expand it to 64 bit. The encrypted address is then added to the process of PRINCE_{core} half way through. Encryption and decryption using tweaked PRINCE is outlined in Figure 3.6. Next, we first turn to the address encryption function T.

Tweak Cipher T

The block cipher T used to encrypt the address is a 27-bit block cipher, thus operating on the entire address at a time. It is an iterated r-round SP-network using a round function $\mathbb{T}_i$ depicted in Figure 3.7.

The encryption block cipher T basically consists first of adding key material $k_2^{(0)}$ and then applying the round function $\mathbb{T}_i$ a number of times r. The exact number of rounds is yet to be determined, however it is suggested that $2 \leq r \leq 10$. In the following,



(a) Encryption

(b) Decryption

Figure 3.6: Encrypting and decryption for tweaked PRINCE.

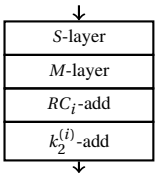


Figure 3.7: Round $\mathbb{T}_i$ of T.

we move on to describe the different components used in the round function $\mathbb{T}$ depicted in Figure 3.7.

The S-box The round function $\mathbb{T}_i$ uses a single 3-bit S-box which is applied in parallel to all nine 3-bit values of the state. The S-box used is given by Table 3.3.

x	0	1	2	3	4	5	6	7
$S[x]$	7	1	3	2	6	5	4	0

Table 3.3: S-box for address tweak encryption round function $\mathbb{T}$.

Linear Mixing Layer M The mixing layer of the address tweak encryption is very similar to that of PRINCE$_{core}$. Here, the 27-bit state is multiplied with a 27×27 matrix

M defined as

$$M = \begin{bmatrix}
1&0&0&0&1&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&1&0&0&0&1&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&1&1&1&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&1&0&1&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&1&0&1&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&1&1&0&0&0&1&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&1&0&0&0&1&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&1&0&0&0&0&1&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&1&0&0&1&1&1&0&0&0\\
0&0&0&0&0&0&0&0&0&1&0&0&0&1&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&1&0&0&0&1&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&1&1&1&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&1&0&0&1&0&0&1&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&1&0&0&1&0&0&1&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&1&0&0&1&0&0&1&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&0&0&0&1&0&1&0&0&1\\
1&0&0&1&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&1&0&0&1&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&1&0&0&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&1&0&1&0&1&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&1&0&1&0&1&0&1&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&0&0&0&1&0&0&0&0&1\\
0&1&0&1&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&1&0&1&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&1&0&0&0&1&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&1&0&0&1&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&1&0&0&1&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&1&0&0&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0
\end{bmatrix}.$$

Round Constants RC_i In each round, the block cipher T adds a round constant RC_i given by Table 3.4.

i	RC_i
1	121FB54
2	22168C2
3	1A62633
4	0A2E037
5	039A252
6	024E088
7	533E63A
8	0082EFA
9	4C76273
10	322514A

Table 3.4: Round constants RC_i for address tweak encryption round $\mathbb{T}_i$.

Key $k_2^{(i)}$ Addition The round function $\mathbb{T}_i$ adds a round key $k_2^{(i)}$, where i is reduced modulo 3. Round keys are defined by splitting the 81-bit master key k_2 into parts such that

$$k_2 = \left(k_2^{(0)} \parallel k_2^{(1)} \parallel k_2^{(2)}\right).$$

Expansion Function E

As is shown in Figure 3.6, after applying the address encryption function $T : \mathbb{F}_2^{27} \longrightarrow \mathbb{F}_2^{27}$, an expansion function E, not to be confused with an encryption function, is used. Just as the trapezoid shape of the E component in the figure suggests, E will elongate a binary string, particularly so the size of it matches that of a PRINCE block state. The definition of the expansion function E is an injective function

$$E : \mathbb{F}_2^{27} \longrightarrow \mathbb{F}_2^{64},$$

where

$$x \longmapsto \big(x \,\|\, x \,\|\, x_{[10]}\big), \quad x \in \mathbb{F}_2^{27}.$$

Here, $x_{[10]}$ denotes the 10 most significant bits of x. Note, that as E is clearly injective, the size of the co-domain equals the size of the domain, which is 2^{27}. As we shall see in Section 6.6.1, this has an implication on the possibilities for intermediate encryption differences (defined in Chapter 4) under certain conditions, when using the address tweak.

3.6 Summary

In this chapter, we have introduced the lightweight block cipher PRINCE, which was developed for the Dutch company NXP Semiconductors. The cipher is different from other lightweight block ciphers seen in the past, such as PRESENT [5], which are commonly optimized towards implementation cost. For PRINCE on the other hand, the main metric is low latency, and this has been obtained by a completely unrolled design.

We saw that PRINCE was chosen to be an iterated SP-network, because this eases the process of arguing the strength of the cipher. Another interesting feature is the α-reflection property, which implies that decryption can be implemented with a minimal overhead. PRINCE uses the wide-trail design strategy, which is a way of obtaining good diffusion properties. The hope in doing so is, that PRINCE will be resistant to differential- and linear cryptanalysis, which we shall introduce in Chapters 4 and 5.

Finally, we described the address tweak for PRINCE, which is a way of obtaining a variant of the cipher, for which encryption of a plaintext depends on its address in memory. Along with the resistance to differential- and linear attacks, this tweak shall also be one of our main cryptanalytic focuses in this thesis.

CHAPTER 4

DIFFERENTIAL CRYPTANALYSIS

Along with *linear cryptanalysis*, described in Chapter 5, *differential cryptanalysis* is one of the most powerful cryptanalytic techniques applied to block ciphers. The technique became known in academia when Biham and Shamir discovered the attack in the late 1980s. Since, it has been applied to a great number of block ciphers.

The attack is an example of a chosen-plaintext attack, and the name is due to its consideration of what was introduced by Lai in [17] as first-order derivatives of Boolean functions.

4.1 Differences and Block Cipher Components

To begin our introduction to differential cryptanalysis, we need the notion of block *difference*.

Definition 1. *The* difference *between two blocks x and y of $\mathbb{F}_2^b$ is defined as*

$$\Delta(x, y) = x \oplus y.$$

Note, that in our treatment of the technique, this is the definition of difference, while in a more generalized setting, differential cryptanalysis can be introduced and applied to other block ciphers not using the $\oplus$ operation.

The hardest part of differential cryptanalysis is, to predict how the difference between a pair of blocks propagate through the remaining components of the block cipher. With that said, there are some parts of a block cipher that behave well with respect to differences, namely addition of round constants and keys, and linear components.

Consider for example a block cipher round function

$$\mathscr{R} : \mathbb{F}_2^b \longrightarrow \mathbb{F}_2^b$$

defined by

$$x \longmapsto L(x) \oplus k,$$

where L is a linear component with respect to the exclusive-or operation, and k is a round key. When this round function is applied to two messages $m_0, m_1 \in \mathbb{F}_2^b$, we

see that the difference across the round function is well defined. Since L is linear, determining the difference across L is easy, since it holds that

$$L(m_0 \oplus m_1) = L(m_0) \oplus L(m_1).$$

Also, we find that

$$(L(m_0) \oplus k) \oplus (L(m_1) \oplus k) = L(m_0) \oplus L(m_1),$$

so key addition actually preserves the difference. Putting the two together, we see that

$$\mathscr{R}(m_0) \oplus \mathscr{R}(m_1) = L(m_0) \oplus L(m_1).$$

In this example, the round function consisted solely of a linear component and a round key addition. As we discussed in Chapter 2, and as is the case with PRINCE, the round function often uses a component which is non-linear with respect to $\oplus$. For PRINCE, and indeed commonly in block ciphers, this non-linear component is an S-box. For such a component S, there are commonly more possible output differences, given a certain input difference. However, the domain and co-domain of the S-box seen as a function S are often so small that we can construct a *difference distribution table*. This table is a tool which defines a probability distribution for the output difference, given a specific input difference, for the non-linear function S.

Definition 2 (Difference distribution table)**.** *A* difference distribution table *for a block cipher component S, which is non-linear with respect to the $\oplus$ operation, is a matrix*

$$D = \begin{bmatrix} d_{0,0} & \cdots & d_{0,j} \\ \vdots & \ddots & \vdots \\ d_{i,0} & \cdots & d_{i,j} \end{bmatrix}$$

where

$$d_{i,j} = |\{x \in Dom(S) \mid S(x) \oplus S(x \oplus i) = j\}|.$$

Using D, we may state that the probability of an input difference i yielding an output difference j has probability

$$p = \frac{D_{i,j}}{|Dom(S)|},$$

when taken across all possible inputs. The hope is, from an attackers point of view, that he can find combinations of input/output differences that yield good probabilities over several rounds of the block cipher, when taking the actions of all cipher components into consideration. We will sometimes use the notation $x \xrightarrow{F} y$ to indicate that a difference x yields a difference y across a function F.

4.2 Characteristics

Next, we continue using the concept of differences to introduce two important new concepts. Note first, however, that when we considered the input difference to a non-linear function S, the difference distribution table considered only a single S-box. However, we described in the introduction of PRINCE in Chapter 3, that the S-layer

applies the S-box 16 times in parallel. To that end, when considering the probability $P\left[x \xrightarrow{S} y\right]$ for specific differences x and y, we consider each nibble of the difference at a time. Formally, if x and y are defined in terms of the 16 nibbles,

$$\begin{aligned} x &= (x_0, x_1, \ldots, x_{15}) \qquad \text{and} \\ y &= (y_0, y_1, \ldots, y_{15}), \qquad x_i, y_i \in \mathbb{F}_2^4, \quad 0 \leq i \leq 15, \end{aligned}$$

then the probability for the entire differences x and y are defined as

$$P\left[x \xrightarrow{S} y\right] = \prod_{i=0}^{15} P\left[x_i \xrightarrow{S} y_i\right]. \tag{4.1}$$

The expression for $P\left[x \xrightarrow{S} y\right]$ of (4.1) holds only because each nibble of x is input to its own S-box independently of the other nibbles.

Having described the probability for a full difference across different cipher components, we now turn to an important definition.

Definition 3 (Characteristic)**.** *An r-round* differential characteristic*, or just* characteristic*, is an $(r+1)$-tuple*

$$(\alpha_0, \alpha_1, \ldots, \alpha_r),$$

where the α_i are differences between two intermediate values of encryption for plaintexts m_0 and m_1, such that $m_0 \oplus m_1 = \alpha_0$. Sometimes, we will also write the characteristic as

$$\alpha_0 \to \alpha_1 \to \cdots \to \alpha_r.$$

The difference α_i, $1 \leq i \leq r$, is the anticipated difference after round i dictated by the characteristic, when using a starting difference α_0.

Having defined r-round characteristics, the obvious next desirable step is to determine them. In a round function such as the one in PRINCE, consisting of an S-box layer, a linear layer, key- and round constant-addition, we note that the probability $P\left[x \xrightarrow{\mathscr{R}} y\right]$ is given by (4.1). This is, as already mentioned, due to the deterministic output difference for linear layers and key/constant additions. In this case, note that (x, y) is in fact a 1-round characteristic. The question that arises is what happens to the probability for characteristics defined over a larger number of rounds.

The probability associated with a characteristic is defined by the probability that $\Delta c = \alpha_r$ and $\Delta x_i = \alpha_i$, $i \leq i < r$, conditioned on $\Delta m = \alpha_0$. Here, Δc is the difference between the encrypted ciphertexts, Δx_i is the difference between intermediate encryption values and $\Delta m = m_0 \oplus m_1$ is the difference between the plaintexts m_0, m_1. Formally, the probability of an r-round characteristic $(\alpha_0, \alpha_1, \ldots, \alpha_r)$ is given by

$$P\left[(\alpha_0, \alpha_1, \ldots, \alpha_r)\right] = P\left[\Delta c = \alpha_r \wedge x_{r-1} = \alpha_{r-1} \wedge \cdots \wedge x_1 = \alpha_1 \mid \Delta m = \alpha_0\right]. \tag{4.2}$$

In (4.2), the probability is taken over all possible plaintext pairs and keys. While (4.2) serves as a definition, the exact computation of the probability for a specific characteristic is impractical in most block ciphers. An exception to this rule of thumb are *Markov ciphers* [16] with independent, randomly chosen round keys, for which it holds that

$$P\left[(\alpha_0, \alpha_1, \ldots, \alpha_r)\right] = \prod_{i=1}^{r} P\left[\Delta x_i = \alpha_i \mid \Delta x_{i-1} = \alpha_{i-1}\right], \tag{4.3}$$

where $\Delta x_r = \Delta c$, $\Delta x_0 = \Delta m$ and Δx_i are the intermediate encryption differences for $1 \leq i < r$. The expression in (4.3) gives the probability of an r-round characteristic as a product of r 1-round characteristics. While generally, (4.3) is not the true expression for (4.2) unless the specified conditions are met, results for e.g. DES [4] show that this is a fair approximation, and indeed for our differential analysis of PRINCE, we will make just this assumption, as is common practice in differential cryptanalysis.

4.3 Differentials

In differential cryptanalysis, when considering an r-round characteristic, the attacker is interested in mainly two things:

1. The starting difference Δm and the ending difference Δc after r rounds, and
2. Maximizing the characteristic probability, because this leads to a better attack success probability, as we shall see when we show how to apply the attack.

To that end, the following definition will potentially help the attacker in boosting the probability of success.

Definition 4 (Differential)**.** *An r-round* differential *is a pair of differences (α_0, α_r) where $\alpha_0 = \Delta m$ and α_r is the anticipated r-round output difference Δc dictated by the differential.*

According to the above, the concept of an r-round differential focuses just on the input and output differences, and thus is like an r-round characteristic, except we do not care about the intermediate differences. Informally, the probability of a differential is thus the sum of probabilities for characteristics $(\alpha_0, *, \ldots, *, \alpha_r)$, where $*$ indicates any value. This probability is given by

$$P\left[(\alpha_0, \alpha_r)\right] = \sum_{x_1} \sum_{x_2} \cdots \sum_{x_{r-1}} P\left[(\alpha_0, x_1, \ldots, x_{r-1}, \alpha_r)\right]. \tag{4.4}$$

Definition 5 (Right and wrong pairs)**.** *When a pair of plaintexts m_0, m_1 such that $\Delta m = \alpha_0$ are encrypted under a key k, we say that the plaintext pair is a* right pair *with respect to a differential (α_0, α_r) if $\Delta c = \alpha_r$ after r rounds of encryption. If $\Delta c \neq \alpha_r$, the pair is said to be a* wrong pair.

4.4 The Attack

We now turn to describing how an $(r-1)$-round differential can be used, as a chosen-plaintext attack, to obtain (parts of) the r^{th} round key in an r-round iterated block cipher. We will again use the notation that m_0 and m_1 are chosen plaintexts for which the attacker requests r-round encryption to obtain c_0 respectively c_1. We also use $x_{j,i}$ to denote the intermediate encryption value for m_j, i.e. the value after adding k_i, for $1 \leq i < r$.

The setup of the attack is as illustrated in Figure 4.1. Having found a good $(r-1)$-round differential (α_0, α_{r-1}) of probability p, the attacker fixes plaintexts m_0, m_1 with $\Delta m = \alpha_0$ and requests the ciphertexts c_0, c_1. He guesses that $\Delta x_{r-1} = \alpha_{r-1}$ with probability p, according to the differential.

To the attacker, the true value of Δx_{r-1} is unknown. However, he does know Δc, and can guess (parts of) the round key k_r and use it to compute

$$\mathscr{R}_r^{-1}(c_0) \oplus \mathscr{R}_r^{-1}(c_1) \tag{4.5}$$

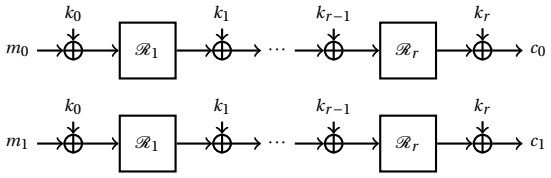


Figure 4.1: Setup for the differential attack.

under the guess for k_r. Given enough such ciphertext pairs, the attacker can compute (4.5) for each ciphertext pair and key guess combination. The attacker then keeps a list of counters T_i, $0 \leq i \leq \tau - 1$; one for each of the τ possible guesses of k_r^i. Whenever a ciphertext pair yields a right pair for (4.5) under k_r^i and the differential in question, the attacker increments T_i by one. In general, if the attacker uses N ciphertext pairs, the expected value of the counter T_i for the correct key guess $k_r = k_r^i$ is $N \cdot p$.

The success of a differential attack relies mainly on two parameters:

1. The probability p associated with the differential, and
2. The number of key guesses τ that each ciphertext pair must be applied to.

As a rule of thumb, a good choice for number of ciphertext pairs N used is $N = m \cdot p^{-1}$ for some small positive constant m. The attacker will need to use these N pairs τ times each, so we can define the actual *work* associated with the attack as

$$W = N\tau.$$

We know that the number of possible choices for k_r is 2^κ. However, depending on the number of non-zero nibbles of α_{r-1} in the $(r-1)$-round differential, the actual τ may be significantly smaller, as we will see next. As an example, consider a differential for $(r-1)$ rounds of PRINCE. We know that α_{r-1} is a 64-bit value, considered as 16 nibbles going into 16 S-boxes in parallel. Naturally, if the input difference to an S-box is `0`, then so is the output difference, because the S-box replaces the same value in each input nibble. In other words, all nibbles with the value `0` of α_{r-1} are guaranteed to have a difference of `0` after applying the S-box of $\mathscr{R}_r$. Consequently, we need only (and can only!) try to recover key bits of k_r equal to four (the number of bits per nibble) times the number of *active* S-boxes of α_{r-1}, i.e. the number of non-zero nibbles of α_{r-1}.

4.5 Filtering

A powerful trick the attacker can apply to both reduce the complexity, but also lower the number of wrong pairs found, is *filtering*. Recall as discussed above, how the active nibbles of α_{r-1} has an influence on the number of key guesses τ considered. Filtering is a trick that uses the same observation; if a certain ciphertext pair does not have precisely the same nibbles active as α_{r-1}, the possibility of a right pair can be ruled out. This is very helpful for the attacker, because wrong pairs do nothing but cloud the purpose of finding the correct key guess.

4.6 Summary

In this chapter we presented differential cryptanalysis, which is one of the most powerful attacks for block ciphers known. We introduced the concepts of characteristics

and differentials, which shall be used heavily throughout this thesis.

After the introduction here, there is still an open question: how does the attacker find characteristics and differentials with good probabilities? We tend to this question in much of the remaining part of this thesis, and specifically we will present various search methods for finding the best possible differentials. As we shall see, there are upper bounds on the probabilities for characteristics, and these help us argue that PRINCE is resistant towards the differential attack.

CHAPTER 5

LINEAR CRYPTANALYSIS

In this chapter we introduce another very powerful attack on block ciphers called *linear cryptanalysis*. The attack was discovered by Matsui [21] in the early 1990s and has had success in application to various ciphers. As we shall see, there are many concepts in linear cryptanalysis that are very similar to concepts from differential cryptanalysis of Chapter 4. In fact, the similarities are striking, and the two methods can be considered siblings in the world of cryptanalytic attacks. A major difference is, however, that linear cryptanalysis is an example of a known-plaintext attack, whereas differential cryptanalysis is a chosen-plaintext attack. Thus, the attack presented here requires fewer assumptions about the attacker.

5.1 Linear Approximations

Linear cryptanalysis is about constructing linear equations, expressing key bits the attacker wants to recover, called *target* key bits, in terms of bits of the plaintext and ciphertext. We give at this point two definitions that are fundamental to our further introduction of the topic.

Definition 6 (Masks and scalar products)**.** *A* mask α*, for a* b*-bit block is a* b*-bit string which acts to* choose *certain bits for an exclusive-or sum of another* b*-bit block. For a mask* $\alpha = (\alpha_0, \ldots, \alpha_{b-1})$ *and a* b*-bit string* $x = (x_0, \ldots, x_{b-1})$*, we use* $\langle x, \alpha \rangle$ *to denote the scalar product between* x *and* α*, s.t.*

$$\langle x, \alpha \rangle = \bigoplus_{i=0}^{b-1} x_i \cdot \alpha_i.$$

For a general b-bit block cipher, the attacker using linear cryptanalysis is looking for masks α, β and γ s.t.

$$\langle m, \alpha \rangle \oplus \langle c, \beta \rangle = \langle k, \gamma \rangle$$

holds with a probability $p \neq \frac{1}{2}$. As was the case with differential cryptanalysis, the introduction starts off by considering how to find linear relations for various cipher components. Then we extend this to single rounds, and finally to several rounds.

Consider first addition of a key k to a block x, i.e. $y = x \oplus k$. The bits of y can be expressed in terms of the bits of x and k as

$$y_i = x_i \oplus k_i, \qquad 0 \leq i \leq b-1.$$

Using the notation of masks and scalar products, this can also be written as

$$\langle k, \alpha^i \rangle = \langle x, \alpha^i \rangle \oplus \langle y, \alpha^i \rangle, \qquad 0 \leq i \leq b-1, \tag{5.1}$$

where α^i is a mask which has a 1 at position i and zeroes elsewhere. Note that the b equations of (5.1) all hold with probability 1. For a function L which is linear with respect to $\oplus$, we can determine the *adjugate* of L, denoted L_{adj}. For matrices, such as the linear layer M' in PRINCE, the adjugate is determined as

$$\left(L^{-1}\right)^T.$$

Having determined the adjugate, we note again that it holds with probability 0 or 1 that

$$\langle x, \alpha \rangle = \langle L(x), L_{adj}(\alpha) \rangle, \qquad \forall x, \alpha \in \mathbb{F}_2^b. \tag{5.2}$$

Just as we saw with differential cryptanalysis, the tricky part are the block cipher components, S-boxes in our case, which are non-linear with respect to $\oplus$. In this case, the attacker tries to find *linear approximations* to relations that hold with some probability $p \neq \frac{1}{2}$. For an S-box, the attacker attempts to find expressions, such that the sum of certain bits in the input matches the sum of certain bits in the output, with probability $p \neq \frac{1}{2}$. Stated otherwise, he looks for masks α, β s.t.

$$\langle x, \alpha \rangle = \langle S[x], \beta \rangle$$

with probability $p \neq \frac{1}{2}$. To determine good masks α and β, we introduce the following definition.

Definition 7 (Linear approximation table)**.** *A* linear approximation table *for a block cipher component S, which is non-linear with respect to $\oplus$, is a matrix*

$$L = \begin{bmatrix} l_{0,0} & \cdots & l_{0,j} \\ \vdots & \ddots & \vdots \\ l_{i,0} & \cdots & l_{i,j} \end{bmatrix}$$

where

$$l_{i,j} = \mid \{x \in Dom(S) \mid \langle x, i \rangle = \langle S(x), j \rangle\} \mid .$$

From Definition 7, we denote the probability that $\langle x, \alpha \rangle = \langle S[x], \beta \rangle$ as

$$p_{\alpha,\beta} = \frac{L_{\alpha,\beta}}{\mid Dom(S) \mid},$$

when taken across all possible inputs. The attacker is generally interested in a high probability $p_{\alpha,\beta}$, but in fact a low probability is equally good. For example, if $p_{\alpha,\beta} = 0.1$ then

$$\langle x, \alpha \rangle = \langle S[x], \beta \rangle \oplus 1$$

holds with probability $1 - p_{\alpha,\beta} = 0.9$. To that end, we introduce the concepts of *bias* and *correlation*.

Definition 8 (Bias and correlation)**.** *The* bias *for a linear approximation*

$$\langle x, \alpha \rangle = \langle S[x], \beta \rangle$$

is defined as $\epsilon_{\alpha,\beta} = p_{\alpha,\beta} - \frac{1}{2}$*. The* correlation *is defined as* $c_{\alpha,\beta} = 2\epsilon_{\alpha,\beta}$*.*

The linear approximation table gives the probability of the a linear approximation over a non-linear function S, however we considered only a single S-box. It is often the case that the block size b is larger than the number of bits in the domain of S. For example, with PRINCE, the block size is $b = 64$ but the S-box only takes as input 4 bit. Again, this means that 16 S-boxes are applied in parallel to the input state, and hence when we talk about the probability of a linear approximation, we must take into account each of the S-boxes applied in parallel. To get the probability for the S-box layer linear approximation, we can multiply the probabilities for each nibble, because the application of the S-boxes are independent of each other. Formally, if $\alpha = (\alpha_0, \ldots, \alpha_{m-1})$ is a mask on the input to the S-box layer defined as a vector of m nibbles, then the probability of the whole linear approximation is defined as

$$P\left[\langle x, \alpha \rangle = \langle S[x], \beta \rangle\right] = \prod_{i=0}^{m-1} p_{\alpha_i,\beta_i} \tag{5.3}$$

for b-bit masks α and β.

5.2 Linear Trails

When the attacker has found good linear approximations for the individual components of the block cipher, the next step is to put them together first for a whole round, and then later on for several rounds. Consider letting c_i denote the state of encryption after i rounds, then $m = c_0$ and $c = c_r$. Assume the attacker has found linear approximations for each round $1 \leq i \leq r$ that each hold with probability p_i (taken over all possible inputs to the round function):

$$\langle c_{i-1}, \alpha_{i-1} \rangle \oplus \langle c_i, \alpha_i \rangle = \langle k_i, \gamma_i \rangle.$$

Adding the r linear approximations together yields a single equation

$$\langle m, \alpha_0 \rangle \oplus \langle c, \alpha_r \rangle = \bigoplus_{i=1}^{r} \langle k_i, \gamma_i \rangle$$

which holds with a certain probability p. In general, computing the probability p is hard because of the highly key-dependent nature of the linear approximations. However, there is a good approximation to the probability p can be computed using the *piling-up lemma*.

Theorem 1 (Piling-up Lemma)**.** *Let* X_i*,* $1 \leq i \leq n$*, be* n *independent random indicator variables and let* $q_i = P[X_i = 0]$*. Then,*

$$P[X_1 \oplus X_2 \oplus \cdots \oplus X_n = 0] = \frac{1}{2} + 2^{n-1} \prod_{i=1}^{n} \left(q_i - \frac{1}{2}\right).$$

The concept of *linear trails* draws a parallel to characteristics from Chapter 4. We consider again an r-round block cipher, and we apply masks to the plaintext, intermediate encryption values and the ciphertext, just as we considered differences in differential cryptanalysis. As such, we define linear trails in the following.

Definition 9 (Linear trail)**.** *An r-round* linear trail*, or just* trail*, is an $(r+1)$-tuple*

$$(\alpha_0, \alpha_1, \ldots, \alpha_r)$$

where the α_i are masks applied to plaintext, intermediate encryption values x_i or ciphertext. The pair of masks α_{i-1} and α_i, $1 \leq i \leq r$, describe a linear approximation for $\mathscr{R}_i$ s.t. the bias for $\langle x_{i-1}, \alpha_{i-1}\rangle = \langle x_i, \alpha_i\rangle$ is large.

When making the assumption that the linear approximations for each of the r rounds are independent, the piling-up lemma can be used to approximate the probability of the linear trail. Finding good approximations for each round individually is easy, but the problems arise when having to combine approximations for several rounds, as the output mask for round i must be the input mask for round $i+1$.

5.3 Linear Hulls

Recall from Chapter 4, that the attacker does not care about the intermediate differences for the attack; the only thing of interest is the differential start and end difference and its associated probability. It should come as no surprise, that exactly the same holds for linear cryptanalysis. To that end, we introduce *linear hulls*.

Definition 10 (Linear hulls)**.** *An r-round* linear hull *is a pair of masks (α_0, α_r) with an associated probability p. The linear hull predicts that*

$$\langle m, \alpha_0\rangle = \langle c, \alpha_r\rangle$$

with probability p.

5.4 The Attack

Next, we turn to describe how the attacker can use an $(r-1)$-round linear hull, as a known-plaintext attack, to obtain (parts of) the r^{th} round key in an r-round iterated block cipher. We use the notation that m is a plaintext and c is a ciphertext.

Having found a good $(r-1)$-round linear hull (α_0, α_{r-1}) of probability p, approximated using the piling-up lemma, the attacker collects N known plaintext-ciphertext pairs for r rounds of the cipher. The attacker will now make a guess at (parts of) the last round key k_r, use this key guess to compute backwards for the ciphertext c to obtain a guess at the intermediate encryption value,

$$x_{r-1} = \mathscr{R}_r^{-1}(c).$$

To the attacker, the true value of x_{r-1} is of course unknown. Assuming there are τ key guesses to make, the attacker can keep *two* lists of counters T_0 and T_1 of length τ; two counters for each guess of k_r^j, $0 \leq j \leq \tau - 1$. For a fixed pair of ciphertext c and key guess k_r^j, the attacker computes x_{r-1} as above and computes

$$v = \langle m, \alpha_0\rangle \oplus \langle x_{r-1}, \alpha_{r-1}\rangle.$$

The attacker increases one of the two counters corresponding to k_r^j by one; $T_0[j]$ if $v = 0$ and $T_1[j]$ if $v = 1$.

Determining how effective a linear attack is going to be is generally hard. The more key bits the attacker tries to determine, i.e. $\log_2 \tau$, the more plaintext-ciphertext pairs

N are needed. While for the differential attack, we were able to make statements about the amount of work needed to determine a correct key guess, based on the probability of the differential and $\log_2 \tau$, there is no straightforward way to do this for the linear attack. In [25], the probability of success for both differential and linear attacks are treated, but for the linear attack it is outside the scope of this thesis.

5.5 Summary

In this chapter we have presented linear cryptanalysis which, besides differential cryptanalysis, is one of the most powerful and generic attacks on block ciphers.

We saw how linear cryptanalysis relates very much to differential cryptanalysis. However, a notable difference is, that linear cryptanalysis is a known-plaintext attack while differential cryptanalysis is a chosen-plaintext attack.

Much of the remaining part of this thesis deals with searching for the best linear trails, linear hulls, characteristics and differentials. By applying various search methods, we argue that the best linear hulls and differentials are not good enough to break PRINCE. This too justifies and shows the strength of the wide-trail design strategy employed.

CHAPTER 6

MEGABOX APPROACH TO DIFFERENTIALS AND LINEAR HULLS

In this chapter we introduce the concepts of *superboxes* and a *megabox* for PRINCE. As we shall see, the megabox description is a clever way to rearrange the components for four rounds of PRINCE.

This rearrangement can often be used for block ciphers utilizing the wide-trail design strategy, and it will benefit us in arguing that PRINCE is resistant to differential- and linear cryptanalysis. To do that, we will use the megabox description of PRINCE to look for good differentials and linear hulls. As will become evident here and in Chapters 8 and 9, the design of PRINCE implies, that finding differentials and linear hulls that are good enough for a practical attack, is infeasible.

The megabox approach to our analysis of this chapter is split into four major parts; first we introduce the PRINCE megabox, then apply it to differential- and linear cryptanalysis, and finally consider the special case of the tweaked PRINCE.

6.1 The PRINCE Megabox

Figure 6.1 shows a reformulation of four rounds of encryption using PRINCE. Note how the application of the M-layer, described in Chapter 3, has been replaced by its definition in terms of the M' matrix and the SR operation, $M = SR \circ M'$. The main observation used in the formulation is, that application of the S-layer and the SR-layer are interchangable without any effect on the result. In other words,

$$(S \circ SR)(x) = (SR \circ S)(x), \qquad \forall x \in \mathbb{F}_2^b.$$

The eight large square boxes in the top and bottom row depict two encryption rounds of 16 consecutive bit of the 64-bit block. These large boxes are called *Super S-boxes* or *superboxes* for short. Note that the layers in each superbox operate on distinct 16-bit values of the 64-bit state, and hence the superboxes operate independently of each other. We will refer to the whole system depicted in Figure 6.1 as the PRINCE *megabox*. Note, that an application of SR, M'-layer and key addition has been left out at the bottom, because these are irrelevant when considering differentials or linear hulls.

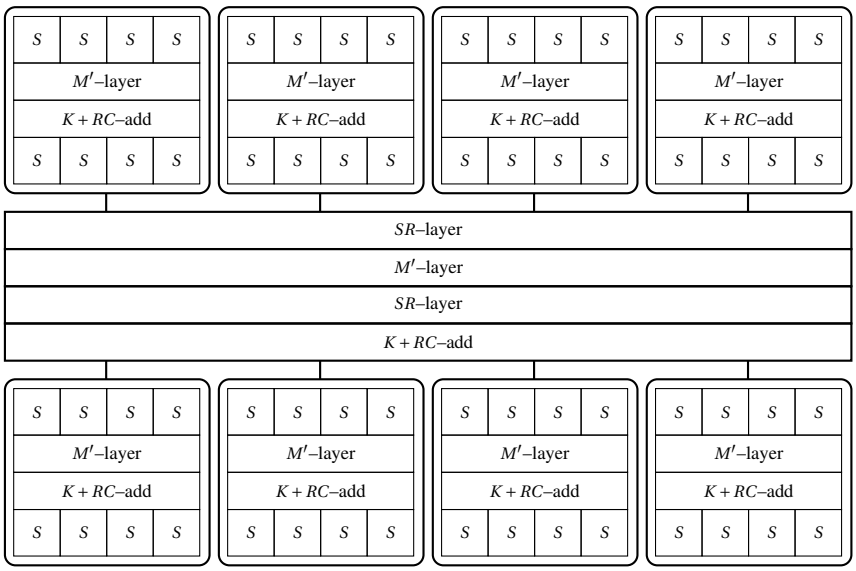


Figure 6.1: The PRINCE megabox for four rounds of encryption.

The reason the encryption can be separated into superboxes is twofold. First, the M' matrix in each superbox operates on 16-bit blocks locally, independently of the other superboxes. This is due to the construction of the M' matrix described in Chapter 3. Second, the SR operation, which mixes all bits of the state and thus can not be part of a superbox, has been pushed down for the first two rounds and up for the last two round. As such, they are collected in the middle, where the entire 64-bit is put together, for what we will refer to as the *middle megabox operations*, after which they are split up again for the lower superbox layer.

As mentioned, the purpose of the wide-trail design strategy employed in PRINCEis to make it resistant towards differential- and linear cryptanalysis. We saw in Chapters 4 and 5, that the probability of success associated with these attacks, depend highly on the number of active S-boxes. To that end, we make the following definition.

Definition 11 (Branch Number)**.** *The* branch number $\mathscr{B}$*, for a certain number of rounds of a block cipher is defined as the smallest number of active S-boxes used by a characteristic or linear trail.*

We shall sometimes use $\mathscr{B}$ as a function mapping characteristics to an integer, such that e.g. $\mathscr{B}((\alpha_0, \ldots, \alpha_r)) = 4$ means the specific characteristic uses 4 active S-boxes.

We present without proof the following lemmas.

Lemma 1. *For two rounds of* PRINCE*, the branch number is $\mathscr{B} = 4$.*

Lemma 2. *For four rounds of* PRINCE*, the megabox will use at least* 4 *active superboxes.*

Theorem 2. *For four rounds of* PRINCE*, the branch number is $\mathscr{B} = 16$.*

Proof. By Lemma 2, a PRINCE characteristic or trail uses at least 4 active superboxes. By Lemma 1, each superbox uses at least 4 active S-boxes. Hence, a megabox

characteristic or linear trail uses at least $4 \cdot 4 = 16$ active S-boxes. Thus, the branch number for four rounds of PRINCE is $\mathscr{B} = 16$. □

6.2 Application to Differential Cryptanalysis

As we saw in Chapter 4, the essential first step in finding a good characteristic for PRINCE is to consider a single S-box. To do so, we present as Table 6.1 the difference distribution table D for the PRINCE S-box.

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	4	0	0	2	0	2	0	4	2	0	2	0	0	0	0
2	0	2	0	4	0	0	0	2	2	0	0	0	0	4	2	0
3	0	0	0	0	0	2	2	0	2	2	2	2	2	0	0	2
4	0	2	2	4	2	2	0	0	2	0	2	0	0	0	0	0
5	0	0	2	2	0	2	0	2	0	2	0	2	2	2	0	0
6	0	0	2	2	0	2	2	0	0	2	0	2	0	0	4	0
7	0	0	2	0	0	0	2	0	2	0	4	0	0	2	2	2
8	0	0	2	0	4	2	0	0	2	2	0	2	0	2	0	0
9	0	0	2	2	0	0	0	0	0	2	2	0	4	2	0	2
a	0	0	0	2	2	4	0	4	2	0	0	0	0	0	0	2
b	0	2	0	0	4	0	0	2	0	0	0	2	2	0	2	2
c	0	4	0	0	0	2	2	0	0	0	2	2	2	0	2	0
d	0	2	0	0	0	0	0	2	0	4	2	0	0	2	2	2
e	0	0	2	0	0	0	4	2	0	0	0	2	2	2	0	2
f	0	0	2	0	2	0	2	2	0	0	2	0	2	0	2	2

Table 6.1: Difference distribution table for the PRINCE S-box. The transposed table corresponds to the difference distribution table for the inverse PRINCE S-box.

Note how each entry, except for entry $(0,0)$, of Table 6.1, has either a 0, 2 or 4. If $d_{i,j} = 4$ it means that $P\left[i \xrightarrow{S} j\right] = 2^{-2}$ while if it is 2, the probability is 2^{-3}. The following definition will serve us throughout the remainder of this thesis, when considering references to the difference distribution table.

Definition 12 (Restricted- and extended candidate sets)**.** *For differential cryptanalysis of* PRINCE*, we define the* restricted candidate set*, for input difference i through the* PRINCE *S-box, to refer be the set of output difference candidates:*

$$\{j \mid d_{i,j} = 4\}.$$

In the case of the extended candidate set*, we allow a larger candidate set for an input difference i:*

$$\{j \mid d_{i,j} \in \{2,4\}\}.$$

In our megabox analysis we focus on the restricted candidate set, because these yield the highest probability. Next, we describe how to use the megabox to find good characteristics for four rounds of PRINCE.

Theorem 2 implies that a 4-round characteristic for PRINCE must use at least 16 active S-boxes. Combining this with the fact that we consider only the restricted candidate sets, in order to get the best probabilities for our characteristics, we introduce the following definition for what we are searching for.

Definition 13 (Good Characteristics)**.** *We define 16-bit characteristics through a superbox, which use a minimal number of 4 active S-boxes, and occur with probability*

$\left(\frac{4}{16}\right)^4 = 2^{-8}$, *as* good *superbox characteristics. Similarly, for the megabox we define* good *megabox characteristics as those that use a minimal of 4 active superboxes, and happen with probability* $\left(2^{-8}\right)^4 = 2^{-32}$.

Trying all 2^{64} input differences α, applying the megabox and checking all output differences β, while filtering out bad characteristics, is computationally infeasible. Luckily, due to the megabox description of four-round encryption shown in Figure 6.1, we may use the independence of the superboxes for two S-box layers to drastically decrease the complexity of our search for the best characteristics, in the following way.

Note that since we are doing a differential attack, we may ignore layers of key- and round-constant addition in the encryption process, because these cancel out when looking at differences.

Recall the construction of M' as a 64×64 matrix with block matrices in the diagonal. The placement of these block matrices was symmetric, and this has an effect on the superboxes in the megabox description. Re-using the notation from Chapter 3, we know that the block matrices in the diagonal of M' are

$$\left(\hat{M}^{(0)}, \hat{M}^{(1)}, \hat{M}^{(1)}, \hat{M}^{(0)}\right).$$

Consequently, when considering either the upper or lower row of superboxes in the megaboxes, the two outer superboxes are equal and the two middle superboxes are equal. To that end, referencing to Figure 6.1, we denote as SB_0 the first and last superboxes of a row, and correspondingly use SB_1 to denote the middle two superboxes.

Due to the independence among the superboxes, we can analyze the two superboxes SB_j, $j \in \{0, 1\}$, once and for all by checking 2^{16} input differences. We now introduce four sets of 16-bit message pair differences in the following way.

Definition 14. *We define four sets* G_j *and* I_j *with* $j \in \{0, 1\}$ *as*

$$I_j = \left\{ \alpha \;\middle|\; P\left[\alpha \xrightarrow{SB_j} \beta\right] = 2^{-8} \right\}, \text{ and}$$
$$G_j = \left\{ \beta \;\middle|\; P\left[\alpha \xrightarrow{SB_j} \beta\right] = 2^{-8} \right\}.$$

In other words, I_j *and* G_j *are the sets of input differences and output differences respectively, for the good superbox characteristics for superbox* SB_j*. This is illustrated in Figure 6.2.*

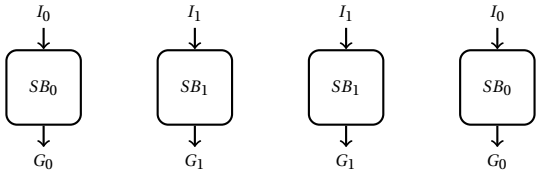


Figure 6.2: A superbox layer showing how the sets I_j and G_j define the good superbox characteristics.

The method introduced above has been summarized as SUPERBOX(j) in Algorithm 1. Having determined I_j and G_j, we may concatenate 16-bit superbox output differences from the G_i sets into a 64-bit block state. To this 64-bit block state we will

Algorithm 1 SUPERBOX(j)

$G_j = \emptyset$
$I_j = \emptyset$
for all 16-bit input differences Δx **do**
  for all possible outcomes Δy where $\Delta x \xrightarrow{SB_j} \Delta y$ **do**
    if $\mathscr{B}(\Delta x \xrightarrow{SB_j} \Delta y) = 4 \wedge P\left[\Delta x \xrightarrow{SB_j} \Delta y\right] = 2^{-8}$ **then**
      $G_j \leftarrow G_j \cup \{\Delta y\}$
      $I_j \leftarrow I_j \cup \{\Delta x\}$
return (G_j, I_j)

apply the megabox middle operations, $SR \circ M' \circ SR$. Afterwards, we split up the 64-bit block into 16-bit differences again, to be input to the lower layer of superboxes.

Since the two superbox layers are identical, we are interested in the 16-bit differences, which after applying the megabox middle layer, are either 0 or are in the sets I_j, because these are the 16-bit input differences to superbox SB_j that give good characteristics in the bottom superbox layer.

Thus, we define two sets containing 64-bit differences as

$$I = (I_0 \cup \{0\}) \times (I_1 \cup \{0\}) \times (I_1 \cup \{0\}) \times (I_0 \cup \{0\}) \qquad \text{and}$$
$$G = (G_0 \cup \{0\}) \times (G_1 \cup \{0\}) \times (G_1 \cup \{0\}) \times (G_0 \cup \{0\}).$$

It is not all elements obtainable as $\beta \in I$ with $\alpha \xrightarrow{SR \circ M' \circ SR} \beta$, $\alpha \in G$, that give good megabox characteristics. While it is certainly a necessary condition, we may filter out a lot of the starting elements of G for the middle round, and hence reduce the complexity. We do this by using the fact that good megabox characteristics use four active superboxes.

Let $\mathscr{B}_S$ be a function which counts the number of active superboxes (i.e. non-zero 16-bit differences of the 64-bit state) over the middle layer of the megabox. Then we may define the sought subset of I as

$$U = \left\{ \beta \;\middle|\; \alpha \in G \wedge \beta \in I \wedge \mathscr{B}_S\left(\alpha \xrightarrow{SR \circ M' \circ SR} \beta\right) = 4 \right\}$$

Note, that if we are able to determine a non-empty set of elements U, we can put together the pieces, i.e. information about the sets I, G and U, to find good megabox characteristics. The method described is formalized as a procedure MEGABOX which uses SUPERBOX(j) as a sub-routine, and is presented as Algorithm 2.

6.3 Changes for Linear Cryptanalysis

In Section 6.2, we saw how to apply the megabox analysis to four rounds of differential cryptanalysis for PRINCE. This section serves to present the considerations we need to make, to make the analysis work for linear cryptanalysis as well.

Thus far, our use of the PRINCE megabox description has been for considering the propagation of differences through four rounds. However, with our knowledge from Chapter 5 of how linear masks evolve through different cipher components, the megabox approach is obviously equally applicable to finding linear trails and hulls.

Algorithm 2 MEGABOX

$U \leftarrow \emptyset$
$(G_0, I_0) \leftarrow$ SUPERBOX(0)
$(G_1, I_1) \leftarrow$ SUPERBOX(1)
$I \leftarrow I_0 \cup \{0\} \times I_1 \cup \{0\} \times I_1 \cup \{0\} \times I_0 \cup \{0\}$
$G \leftarrow G_0 \cup \{0\} \times G_1 \cup \{0\} \times G_1 \cup \{0\} \times G_0 \cup \{0\}$
for all 64-bit differences $\Delta x \in G$ **do**
    Define Δy as $\Delta x \xrightarrow{SR,M',SR} \Delta y$
    if $\Delta y \in I \wedge \mathscr{B}_S\left(\Delta x \xrightarrow{SR,M',SR} \Delta y\right) = 4$ **then**
        $U \leftarrow U \cup \{\Delta y\}$
return U

As was the case with differential cryptanalysis, the first step in mounting a linear attack on PRINCE is to find a collection of the best linear trails. In this section we focus on finding the linear trails for four rounds of PRINCE encryption, that we will eventually use to construct linear hulls.

In the linear case, the input masks α for the good 16-bit superbox trails are defined by the sets I_j and the corresponding output masks β are defined by the sets G_j. Just as with the differential analysis, the set U defines the set of 64-bit trails over $(SR \circ M' \circ SR)$ or the corresponding inverse layer, that gives a total of 16 active S-boxes for four rounds.

Our implementation of the search method for good differential characteristics is generic enough to change one minor thing to make it work for finding the best linear trails, and that is using the linear approximation table L rather than the difference distribution table D. This table, for the PRINCE S-box, is presented in Table 6.2.

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	−2	−2	2	−2	0	4	−4	0	−2	2	−2	−2	0	0
2	0	0	−4	0	−4	0	0	0	−2	−2	2	−2	−2	2	2	2
3	0	4	−2	−2	2	−2	0	0	2	2	4	0	0	0	2	−2
4	0	0	−4	4	2	2	2	2	2	2	−2	−2	0	0	0	0
5	0	0	2	2	0	−4	−2	2	2	−2	0	−4	−2	−2	0	0
6	0	−4	0	0	−2	−2	2	−2	0	4	0	0	−2	−2	2	−2
7	0	0	2	−2	0	0	−2	2	0	4	−2	−2	0	4	2	2
8	0	−2	−2	0	4	−2	−2	−4	−2	0	0	−2	2	0	0	2
9	0	−2	0	2	2	0	−2	0	2	0	2	4	−4	2	0	2
a	0	2	2	4	0	2	−2	0	−4	2	2	0	0	−2	2	0
b	0	−2	0	−2	2	4	−2	0	0	−2	0	−2	−2	0	2	−4
c	0	−2	−2	0	−2	0	−4	2	0	2	2	0	2	0	−4	−2
d	0	−2	0	2	0	−2	0	2	0	−2	0	2	4	2	4	−2
e	0	−2	2	0	2	0	4	2	−2	0	4	−2	0	2	−2	0
f	0	2	0	2	0	−2	0	−2	−2	0	−2	0	−2	4	−2	−4

Table 6.2: Linear approximation table for the PRINCE S-box. The transposed table corresponds to the linear approximation table for the inverse PRINCE S-box.

As we did when applying the megabox to look for good characteristics, we will again restrict ourselves to use the restricted candidate sets which, for linear cryptanalysis, are redefined in the frame of the linear approximation table.

Definition 15 (Restricted- and extended candidate sets)**.** *For linear cryptanalysis of* PRINCE*, we define the* restricted candidate set*, for input mask i through the* PRINCE

S-box, to refer be the set of output mask candidates:

$$\{j \mid |l_{i,j}| = 4\}.$$

In the case of the extended candidate set*, we allow a larger candidate set for an input mask i:*

$$\{j \mid |l_{i,j}| \in \{2, 4\}\}.$$

Once the candidates are computed and stored by the program for each input mask α, the remaining process is nearly identical to that of the differential attack. The only thing we need to change is to make sure to use the adjugate matrix of M', c.f. Chapter 5, when finding the output mask for the M' layer. It turns out, however, that $M'_{adj} = M'$, since M' is symmetric and an involution, so applying M' to α suffices as the output mask.

6.3.1 Linear Trail Probabilities

Here, we briefly present the expected probability of the linear approximations found for the megabox. When computing the trail probabilities, there are two things to note:

1. We are only using entries from the linear approximation table where the bias is $|\epsilon| = 2^{-2}$.
2. All trails use precisely 16 active S-boxes.

With this in mind, we can easily approximate the probability associated with each linear trail, by applying the piling-up lemma:

$$\begin{aligned} p &= \frac{1}{2} + 2^{16-1} \prod_{i=1}^{16} \left(p_i - \frac{1}{2} \right) \\ &= \frac{1}{2} + 2^{15} \prod_{i=1}^{16} \left(\pm 2^{-2} \right) \\ &= \frac{1}{2} \pm 2^{-17}. \end{aligned}$$

This, in turn, corresponds to a bias of $|\epsilon| = 2^{-17}$.

6.4 Consideration of Middle- and Inverse Rounds

As was described using Figure 6.1, the first four encryption rounds of PRINCE can be described using a megabox. This is also the case for the middle part as well as the last, inverse, part of the cipher. The changes are very simple:

- For the middle part, we substitute the lower layer of superboxes by inverse superboxes which use the inverse S-boxes. As such, these inverse superboxes represent two applications of $\mathscr{R}^{-1}$. We also replace the last SR of the megabox middle operations by SR^{-1}.
- For the megabox for the last four encryption rounds, we replace all superboxes with inverse superboxes, and replace both SR by inverse SR^{-1}.

Using the modified megaboxes, the superbox and megabox analyses are identical to that used for the first four rounds of PRINCE, as described in Section 6.2. When we present our results in the following, we also give results for four rounds of PRINCE taken around the middle, and four inverse rounds, i.e. $(\mathscr{R}^{-1} \circ \mathscr{R}^{-1} \circ \mathscr{R} \circ \mathscr{R})$ and $(\mathscr{R}^{-1} \circ \mathscr{R}^{-1} \circ \mathscr{R}^{-1} \circ \mathscr{R}^{-1})$, respectively.

6.5 Results

In this section we present results for the megabox approach of analysis introduced in this chapter. We have applied the megabox to the following three scenarios, for both differential- and linear cryptanalysis:

- Four regular rounds $\mathscr{R}$ of PRINCE,
- Four rounds of PRINCE, taken around the middle components, i.e. $\mathscr{R}^{-1} \circ \mathscr{R}^{-1} \circ \mathscr{R} \circ \mathscr{R}$, and
- Four inverse rounds $\mathscr{R}^{-1}$ of PRINCE.

Note however, that all results for the latter scenario will be exactly the same as for four regular rounds. This is due to the α-reflection property and the symmetric design of the components implied by it. For example, for any good differential (α, β) for four regular rounds of PRINCE, (β, α) will be a good differential for four inverse rounds of PRINCE. To that end, we present only results on the former two of the scenarios listed above.

Using the MEGABOX procedure described in Section 6.2, we have been able to find all four-round characteristics and linear trails for PRINCE, using 16 active S-boxes. Recall that the set of good inputs and outputs for superbox SB_j, $j \in \{0, 1\}$, were denoted I_j and G_j respectively. In our tests, we found that for a fixed scenario c.f. the above, it holds that $I_0 = I_1$ and $G_0 = G_1$. As such, we simply refer to them as I and G in the results presented in the following. The sets I_j and G_j for four regular rounds of PRINCE are listed in Appendix B.

To present our results, we introduce two auxiliary sets:

- ΔX which is the set of input differences for all good characteristics, and
- ΔY, which is the set of output differences from all good characteristics.

Scenario	# of characteristics/trails	# in best differential/hull	$\lvert\Delta X\rvert$	$\lvert\Delta Y\rvert$
Differential, regular rounds	313,408	1	10,592	15,176
Differential, middle rounds	477,500	2	11,424	11,424
Linear, regular rounds	5,120,000	1	155,372	78,508
Linear, middle rounds	7,619,840	1	145,828	145,828

Table 6.3: Results for megabox analysis for characteristics and linear hulls, for four rounds of PRINCE.

Table 6.3 gives the results found during our megabox analysis for the three scenarios mentioned above, in both the differential- and linear cryptanalysis cases. The most important thing to notice from the result is, that even though we have found many characteristics and linear hulls, using the minimal number of $\mathscr{B} = 16$ active S-boxes, the number of characteristics in the best differential and linear hull respectively, is low. In fact the only case we have a differential with more than one characteristic is for four middle rounds, where we find a differential of probability 2^{-31}. This indicates that the design of PRINCE results in poor *differential- and linear hull effect*, i.e. the attacker can not find differentials or linear hulls with many characteristics or trails, respectively.

6.6 Application to Tweaked PRINCE

In this section, we consider again using the megabox description, this time for analyzing the special case of the tweaked PRINCE. The megabox analysis is applied in Section 6.6.2. First, we make a crucial observation about the tweaked PRINCE, which we shall use both in our analysis here, but also in Chapters 8 and 9.

6.6.1 Consequences From A Differential Perspective

Consider the encryption of plaintexts m_0 and m_1 stored in two different addresses a_0 and a_1 in memory, respectively. Let the intermediate values after five rounds of PRINCE$_{core}$ and an S-layer be denoted by x_0 and x_1. As the first five rounds of PRINCE$_{core}$ encryption does not depend on the address, it trivially holds that

$$x_0 \oplus x_1 = 0.$$

At this point, the difference in the output of the expansion function E is added to the difference $x_0 \oplus x_1 = 0$. Due to the expansion function E used in the address encryption tweak, the difference is of a special form. Namely, if we let

$$t = T(a_0) \oplus T(a_1),$$

then after the application of E, the difference between the intermediate values input to the remaining encryption process is

$$(x_0 \oplus x_1) \oplus (E(T(a_0)) \oplus E(T(a_1))) = \left(t \parallel t \parallel t_{[10]}\right),$$

where $t_{[10]}$ denotes the 10 most significant bits of t. We will refer to a difference of this form as the *repetitive* form. This observation on the address tweak from a differential perspective, is what will form the basis of the analysis in the following.

6.6.2 Finding Characteristics

Due to the expansion function E, we have established that there are 2^{27} possible differences of the repetitive form, as $|Dom(E)| = 2^{27}$. Let us denote the set of all these differences as

$$\Gamma = \{\alpha \mid \alpha \text{ is of repetitive form}\}.$$

As of Section 6.6.1, the remaining layers of the middle part of PRINCE$_{core}$, after adding the output from the address tweak encryption, is $(S^{-1} \circ M')$. Thus, we must make a minor modification to the megabox, to make it describe four rounds for what we denote the *tweak scenario*, i.e. after adding the output of the address encryption. The resulting new megabox is presented in Figure 6.3.

As the set Γ does not hold an incomprehensible amount of differences, a first approach to utilizing the megabox to find good characteristics, would be to apply the feasible (in the sense of the S-layer) top transformations $(S^{-1} \circ M')$ to each $\alpha \in \Gamma$, to arrive at a set of 2^{27} differences,

$$\Theta = \{((S^{-1} \circ M')(x_0) \oplus (S^{-1} \circ M')(x_1)) \mid (x_0 \oplus x_1) \in \Gamma\}.$$

This is the set of possible input differences to the first superbox layer. The hope is then, that the existing megabox analysis completed earlier, can be applied to the set Θ to find the best characteristics.

In the following, we will describe an iterative process in which different approaches have been made to finding good characteristics.

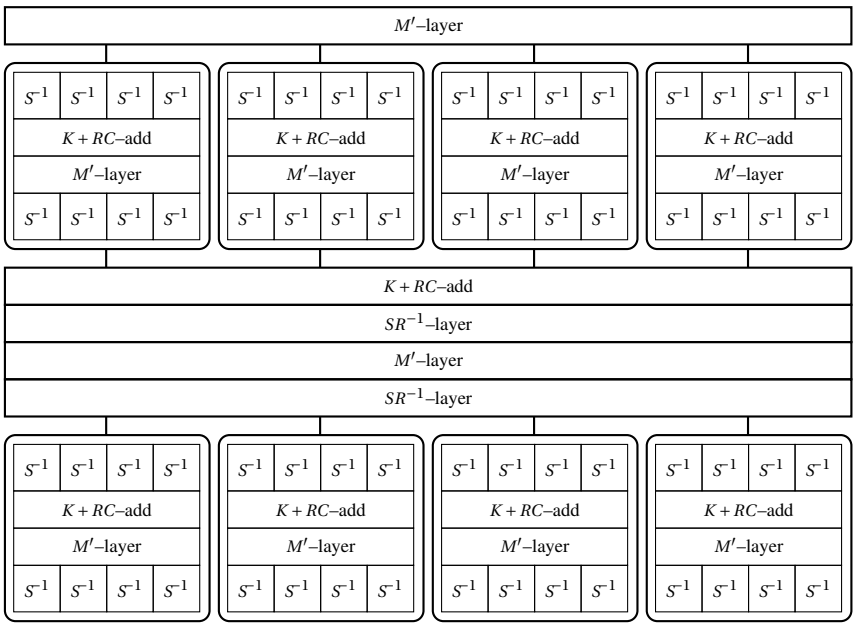


Figure 6.3: Tweaked PRINCE Megabox.

First Approach

The analysis of Section 6.2 considers only characteristics using the minimal number of four active superboxes. At this point, we are just interested in minimizing the number of active superboxes for a characteristic starting with $\alpha \in \Gamma$, rather than requiring a specific number. As Algorithm 3 we present a modified MEGABOX routine. Recall that the original megabox analysis uses the SUPERBOX(J) sub-routine to find the inputs to each superbox, that yield output differences using the minimal number of four active S-boxes.

Algorithm 3 MEGABOX-TWEAKED

1: $(G_0, I_0) \leftarrow$ SUPERBOX(0)
2: $(G_1, I_1) \leftarrow$ SUPERBOX(1)
3: $I \leftarrow I_0 \cup \{0\} \times I_1 \cup \{0\} \times I_1 \cup \{0\} \times I_0 \cup \{0\}$
4: **for all** Differences $\epsilon \in I$ **do**
5: **if** $\epsilon \in \Theta$ **then**
6: **for all** Differences ζ s.t. $\epsilon \xrightarrow{SuperBox\ Layer} \zeta$ **do**
7: $\eta \leftarrow (SR^{-1} \circ M' \circ SR^{-1})(\zeta)$
8: **if** $\eta \in I$ **then**
9: Store characteristic and number of active superboxes used, $\mathscr{A}(\epsilon) + \mathscr{A}(\eta)$

Informally, Algorithm 3 will split each $\beta \in \Theta$ into 16-bit blocks, see for each of these blocks which superbox characteristics will use 4 active S-boxes per block, assemble these into 64-bit blocks again and see which of those, after applying the middle part of the megabox, yield 16-bit blocks that are give good superbox characteristics for the lower layer.

When considering what we defined as the restricted candidate case, for the analysis

of the SUPERBOX sub-routine, the tweaked megabox analysis program completes within reasonable time. However, no characteristics are found using just 4 active S-boxes per superbox.

Second Approach

In the previous approach, we found that when using the restricted candidate set, we are not able to find characteristics using just 4 active S-boxes per superbox for the tweaked PRINCE. This was when the analysis uses only characteristics that use 4 active S-boxes per superbox. From this, two immediate next strategies arise; either we switch to using the extended candidate sets, or we allow a higher number of active S-boxes in each superbox. If either of these strategies turn out not to work, a third strategy might be to combine the two, to relax the constraints even further.

When considering using the extended candidate sets, our results show, for each superbox, that the set of characteristics through a superbox using 4 active S-boxes result from a combination of $|I| = 5502 \approx 2^{12.43}$ input differences and $|G| = 4606 \approx 2^{12.17}$ output differences. These sets are huge to compared to the sets of sizes 130 and 100, respectively, when using the restricted candidate sets. The work required to analyze a single superbox would be in the order of $2^{4 \cdot 12.43} > 2^{49}$, which is an impractical amount.

Instead, we go back to using the restricted candidate sets, but allow more active S-boxes per superbox. Allowing five active S-boxes per superbox in our megabox analysis, we find that each superbox finds characteristics resulting from a combination of 596 input- and 523 output 16-bit differences. Even in this case, the characteristics found use all 8 active superboxes. As we here allow superbox characteristics using up to 5 active S-boxes, the total number of active S-boxes is between 32 and 40, and hence the probability is $2^{-80} \leq p \leq 2^{-64}$, which is worse than exhaustive search.

6.7 Summary

This chapter has introduced the PRINCE megabox, which is a way of remodeling four consecutive rounds of encryption. As we saw, the description is possible due to the way the matrix M' in PRINCE is constructed.

We saw with the introduction of branch number, that for PRINCE, the smallest number of active S-boxes over four rounds is 16. Using the megabox description, we were effectively able to greatly lower the complexity of a search for differentials and linear hulls. In the case where we confine ourselves to the restricted candidate sets, i.e. consider only S-box candidates that occur with maximal probability, we find many characteristics and linear trails using only 16 active S-boxes. However, our results also show that PRINCE exhibits a poor differential effect, in that the best differential found for any placement of four rounds, has just two characteristics and hence a probability of 2^{-32}. The same holds for linear hulls, where none have more than a single trail in them.

Section 6.6.1 showed how, for the tweaked PRINCE, an attacker is able to obtain knowledge about the intermediate encryption value difference, at a certain point during encryption. In Section 6.6.2, we tried applying a modified megabox to the search for finding differentials for this special tweak scenario. The best differentials found had a probability of 2^{-64} at best, which is impractical for an attack on just four rounds. In Chapters 8 and 9, we will introduce better methods for finding good differentials in the tweaked PRINCE scenario.

This chapter served as an introduction to our search for the best differentials and linear hulls for PRINCE. Our results here indicate already, having found the best results to exhibit low probabilities and poor differential- and linear hull effects, that PRINCE is a lightweight block cipher which is resistant to differential- and linear attacks. In Chapter 8 and 9 we shall apply other, very different search methods, for finding good differentials and linear hulls for PRINCE.

CHAPTER 7

STATISTICAL DISTINGUISHER FOR TWEAKED PRINCE

Recall from Chapter 2 that in a distinguisher attack, the attacker is able to tell the difference between the output of two black-boxes, where one box contains the block cipher, with a randomly chosen key, and the other contains a randomly chosen permutation. The distinguisher attack is, in the sense that it does not use any knowledge about the structure or input to the block cipher, a weak type of attack. In this chapter, we consider such an attack on tweaked PRINCE.

7.1 Setup

In our setup we consider the encryption of two equal messages at two distinct addresses in memory, $a_0 \neq a_1$. Thus, we know for sure that the difference α after adding the encryption of the addresses, is of repetitive form. If PRINCE is resistant to distinguisher attacks, this will imply that, even when the difference after five rounds of PRINCE_{core} is known to have a certain form, the outputs c_0 and c_1 (and hence $c_0 \oplus c_1$ too) should look random. In other words, it should not be possible to distinguish the difference between outputs from the black box containing PRINCE from the difference between outputs from the other black box, containing a permutation.

7.2 The χ^2 Analysis

A commonly applied test to determine, based on empirical data, if a source can be considered having a distribution X, is the χ^2 test. To perform a χ^2 test, we first form a hypothesis [15, 14]. Commonly, the hypothesis states that

> "The data source, which we are testing, has distribution X".

In many applications, and certainly in ours, we are interested in seeing if the empirical data can be considered random. As such, it is natural to choose X to be the discrete uniform distribution, and indeed this is the distribution we choose for X in our analysis.

Having determined the hypothesis, two remaining steps constitute the χ^2 test. The first step is, based on the observed and expected data, to compute a χ^2 value given by

$$\chi^2 = \sum_{i=1}^{k} \frac{(o_i - e_i)^2}{e_i}. \tag{7.1}$$

The k in (7.1) is the size of the outcome space. For example, if we consider testing a 20-sided die[1] for fairness, the test would use $k = 20$ and hypothesize that the die is fair. The o_i and e_i of (7.1) are the observed and expected frequencies for the i^{th} outcome value.

The second step of the analysis is to compare the computed χ^2 value to a lookup value $\chi^2_{a,d}$, which is the threshold value for d degrees of freedom at significance level a. If the computed χ^2 exceeds $\chi^2_{a,d}$, the hypothesis is rejected at significance level a, otherwise it is accepted. In our case, the conclusion will be that, at a certain level of confidence, we either accept the output differences as looking random, or we reject it as so.

To start the analysis for the tweaked PRINCE, we need to decide exactly what to measure as the source output. The set of possible outcomes should have at least 5 observations each, in order for the test to be successful [14]. To that end, it does not make sense to measure the whole 64-bit output difference at a time, because this would require, in expectation, at least $5 \cdot 2^{64}$ message pair encryptions. Of course, such a vast number is unrealistic as, for example, exhaustive key search has a time complexity of $O(2^{64})$. Instead, we split the string into smaller blocks and consider them individually.

Assume generally that an n-bit string x is chosen uniformly at random from $\mathbb{F}_2^n$. In this case, it is clear, that as x is random, so is any w-bit substring of x, $w < n$. Stated differently, if some w-bit substring of x is not random, then neither is x. This gives rise to the idea, that rather than testing 64-bit output differences from PRINCE_{core}, we may look at w-bit sub-blocks of the 64-bit difference individually. If one of the w-bit sub-blocks can be distinguished from a random w-bit generator, this implies that the 64-bit output difference can be discarded as looking random. For simplicity, we make sure that w divides 64, so a whole number of sub-blocks make up the whole difference.

7.3 Generating Data

For generating the empirical data on which we will base the χ^2 test, a small C++ program is written. The following describes how we perform a *single* χ^2 test for each w-bit sub-block of the output difference. First, a value $t \in \mathbb{F}_2^{27}$ is chosen, and α is constructed as

$$\alpha = (t \parallel t \parallel t_{[10]}).$$

For a certain number of iterations I (we used $I = 2^{18}$), a 64-bit plaintext m_0 is chosen, and the other plaintext is computed as $m_1 = m_0 \oplus \alpha$. This is done in order to mimic the encryption process of identical plaintexts at distinct addresses in memory, which will have difference α after adding in the address encryption, in the tweak part. Depending on the number of rounds being analyzed, the corresponding remaining part of PRINCE_{core} is applied to m_0 and m_1 separately, leaving out key- and round constant additions for speed, as these play no role in when considering differences. In the end of

[1] 20-sided dice are commonly used in role-playing board games.

β	0	1	2	3	4	5	6	7
0	16490	16285	16334	16306	16407	16295	16420	16488
1	16560	16384	16405	16172	16358	16381	16352	16388
2	16224	16417	16292	16158	16475	16309	16537	16497
3	16212	16286	16547	16384	16515	16478	16545	16193
4	16373	16326	16297	16368	16377	16578	16182	16309
5	16374	16318	16459	16597	16335	16116	16441	16641
6	16468	16423	16211	16365	16557	16362	16274	16232
7	16324	16439	16492	16496	16364	16572	16322	16285
8	16320	16544	16560	16502	16354	16247	16467	16409
9	16458	16344	16394	16383	16302	16467	16335	16440
a	16660	16427	16412	16386	16445	16461	16480	16330
b	16478	16470	16324	16373	16397	16234	16546	16202
c	16365	16325	16287	16301	16191	16317	16365	16508
d	16297	16504	16304	16539	16442	16519	16227	16452
e	16306	16490	16310	16269	16339	16260	16431	16445
f	16235	16162	16516	16545	16286	16548	16220	16325

β	8	9	10	11	12	13	14	15
0	16337	16161	16300	16491	16419	16422	16305	16329
1	16435	16462	16238	16435	16427	16259	16421	16364
2	16412	16285	16298	16225	16246	16277	16475	16305
3	16192	16423	16257	16464	16356	16462	16410	16378
4	16151	16275	16366	16408	16518	16448	16395	16551
5	16338	16233	16453	16169	16460	16392	16411	16521
6	16446	16388	16512	16402	16175	16415	16309	16222
7	16478	16380	16506	16121	16343	16401	16452	16476
8	16550	16667	16290	16505	16322	16274	16327	16423
9	16606	16385	16519	16519	16563	16424	16236	16451
a	16323	16407	16536	16366	16289	16447	16233	16292
b	16557	16294	16293	16332	16216	16408	16496	16167
c	16370	16550	16372	16459	16551	16477	16424	16323
d	16364	16405	16434	16289	16449	16532	16327	16445
e	16374	16434	16357	16516	16561	16213	16491	16317
f	16211	16395	16413	16443	16249	16293	16432	16580

Table 7.1: 4-bit nibble value frequencies for 2^{18} ciphertext differences. Column i gives the observed frequencies for the i^{th} most significant 4-bit nibble.

the iteration, the frequency for the value of each w-bit sub-block of $c_0 \oplus c_1$ is increased in a process-wide table.

Table 7.1 gives an example of empirical data generated, using the method described above, when running for $I = 2^{18}$ iterations, considering sub-blocks of $w = 4$ bit, i.e. 16 nibbles. The data itself seems inane to look upon, but applying the χ^2 test extracts usefulness from the data, and provides a better ground of making a conclusion, as we shall see.

For our test, we are considering each output w-bit sub-block as an individual distribution which is independent of the other sub-blocks. As we consider w-bit sub-blocks, the size of the outcome space is

$$k = 2^w.$$

We will in fact be conducting $\left(\frac{64}{w}\right)$ χ^2 tests; one for each sub-block j with $0 \leq j \leq \frac{64}{w} - 1$ (counting from the most significant end). Consequently, we compute $\frac{64}{w}$ values of χ_j^2, and will eventually compare each of these with a threshold value $\chi_{a,d}^2$.

The procedure described covers a *single* χ^2 test for each sub-block. For our experiments, we have chosen to run several tests for each sub-block. However, the implementation makes sure to only use a plaintext pair (m_0, m_1) once during all tests. This is done to avoid using the pairs (m_0, m_1) *and* (m_1, m_0), which would yield the same results as $\oplus$ is a commutative operation. Furthermore, we make sure to use a fixed, randomly chosen α across all tests. The reasoning behind this is, that the distribution of values on the sub-blocks may vary greatly, depending on the α chosen. So rather than using a random α for each test, we fix it for all tests to eliminate this potential source of error.

Having performed the experiment, running 20 χ^2 tests for each sub-block, we get 20 χ^2 values for each w-bit sub-block. Some of these values are presented in Table 7.2. A complete table showing values for all tests can be found in Appendix C.

Nibble #	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6
0	18.79	10.26	15.43	14.34	12.23	14.12
1	8.29	26.90	20.29	24.90	19.28	20.20
2	13.49	14.95	13.02	15.66	12.57	13.31
3	23.02	8.59	19.01	8.64	15.21	6.73
4	11.90	22.89	18.36	17.65	29.53	24.73
5	27.59	10.31	15.06	14.75	8.75	11.79
6	16.53	21.07	16.00	10.81	10.30	10.97
7	12.14	12.36	18.34	8.36	9.52	16.24
8	11.77	17.99	9.90	25.03	22.63	9.45
9	17.13	9.16	13.41	9.71	28.42	11.67
10	10.17	8.98	6.61	10.47	14.18	19.81
11	16.56	14.78	16.18	15.23	9.04	24.26
12	18.15	16.67	9.76	16.95	10.19	10.44
13	14.23	9.07	14.96	30.70	18.24	17.08
14	10.01	17.16	16.79	9.89	9.05	16.73
15	16.62	6.58	14.91	8.90	19.43	18.09

Table 7.2: Computed χ^2 values for selected experiments using $w = 4$-bit sub-blocks, counting from most significant to least significant.

Having performed the first step of the χ^2 test, we now turn to the threshold value $\chi^2_{a,d}$. To that end, we need to determine the degrees of freedom, d, for our test. In [14, Section 9.5], the degrees of freedom d is defined as

$$d = k - m,$$

where m is

> "the number of quantities, obtained from the observed data, that are needed to calculate the expected frequencies."

In our case, we may sum over the observed frequencies and divide by k to obtain the expected value $E = 2^{18}/k$, according to the uniform distribution. Thus, we find that $m = 1$, and hence

$$d = k - 1.$$

Table 7.3 gives threshold values $\chi^2_{a,d}$ for $d = 15$ and $d = 255$ degrees of freedom and selected values of the significance level a. These degrees of freedom correspond to experiments where we consider sub-blocks of $w = 4$ or $w = 8$ bit, i.e. nibbles and bytes respectively. The interpretation of the threshold values is, that there is a probability a for a χ^2 value in a test with d degrees freedom to go as high as $\chi^2_{a,d}$.

a	0.0001	0.001	0.01	0.05	0.1	0.2	0.3	0.4	0.5
$\chi^2_{a,15}$	44.263	37.697	30.578	24.996	22.307	19.311	17.322	15.733	14.339
$\chi^2_{a,255}$	347.65	330.52	310.46	293.25	284.34	273.79	266.34	260.09	254.33

Table 7.3: Threshold values $\chi^2_{a,d}$ for $d \in \{15, 255\}$ and selected values of a.

7.4 Results

If we compare e.g. the χ^2 values of Table 7.2, for six test runs on nibble basis, we see that most values are less than or equal to 20. However, there are some higher values, e.g.

- $\chi^2_5 = 27.59$ for Test 1,
- $\chi^2_1 = 26.90$ for Test 2,
- $\chi^2_1 = 24.90$, $\chi^2_8 = 25.03$ and $\chi^2_{13} = 30.70$ for Test 4,
- $\chi^2_4 = 29.53$ and $\chi^2_9 = 28.42$ for Test 5,
- $\chi^2_4 = 24.73$ and $\chi^2_{11} = 24.26$ for Test 6.

When considering one test individually, most of the nibbles for the χ^2 listed would be rejected as being uniformly distributed, at a significance level $a = 0.05$, for which the threshold value is $\chi^2_{0.05,15} = 24.996$.

For example, considering Test 2, we conclude that we must reject the hypothesis for nibble 1 (i.e. bits 4-7, counting from the left, starting from 0), at a significance level of $a = 0.05$, i.e. an error probability of at most 0.05. For the other 4-bit nibbles in Test 2, the hypothesis can not be rejected at this significance level, using the χ^2 test based on the observed data.

At this point, the question still stands: can we use the χ^2 test to distinguish the two black-boxes? As mentioned above, we can consider tests individually and make conclusions to reject the hypothesis for most of the tests. This is because some nibbles yield χ^2 values high enough to reject them as randomly distributed, with a low probability of error.

However, as is evident from Appendix C and Table 7.2, it is not the always the same sub-blocks in each test, for which we can reject the hypothesis. This is despite the steps we have taken to rule out sources of error, mainly:

- Using a fixed value for the repetitive form difference α and
- Making sure only to use one of the message pairs (m_0, m_1) and (m_1, m_0), which yield the same statistic.

Another thing to observe, especially from the test data provided in Appendix C is, that the amount of high χ^2 values may not be extraordinarily large. To back this claim, we have, for all 20 tests of Appendix C, partitioned the calculated χ^2 values into intervals and computed the fraction of χ^2 values in each interval. This number (corresponding to the probability for a randomly chosen test χ^2 value falling into a certain interval) is compared to the *expected probability* for a χ^2 value falling into the same interval. The latter number is computed using the cumulative distribution function (CDF) for the χ^2 distribution. The comparison of observation probability and expected probability, for selected intervals and for sub-blocks corresponding to nibbles and bytes, is presented in Table 7.4.

From Table 7.4, we especially highlight the low absolute differences between the observed probabilities and expected probabilities, for each interval. We conclude on these grounds that the differences are so small, that we may *dismiss* the initial indication from the χ^2 tests stating that the highest test χ^2 values indicate the source being non-uniform.

7.5 Summary

In this chapter we have considered using the χ^2 analysis to perform a statistical distinguisher attack on the tweaked PRINCE. The model of attack is a chosen-plaintext attack, in the sense that the attacker must request the encryption of equal plaintexts at different addresses, from each black box.

χ^2 Interval	Count	Obs. prob.	Exp. prob.	Abs. diff.
205-220	6	0.0375	0.0457	0.0082
220-230	13	0.0813	0.0772	0.0041
230-240	18	0.1125	0.1262	0.0137
240-250	21	0.1313	0.1649	0.0336
250-260	28	0.1750	0.1751	0.0001
260-270	32	0.2000	0.1536	0.0464
270-280	20	0.1250	0.1127	0.0123
280-290	15	0.0938	0.0701	0.0236
290-301	7	0.0438	0.0398	0.0039

(a) Data for sub-blocks of byte size (8 bit).

χ^2 Interval	Count	Obs. prob.	Exp. prob.	Abs. diff.
3.4-5	4	0.0125	0.0070	0.0055
6-7	11	0.0345	0.0221	0.0124
7-8	7	0.0219	0.0339	0.0119
8-9	18	0.0564	0.0463	0.0102
9-10	24	0.0752	0.0578	0.0175
10-11	25	0.0784	0.0671	0.0112
11-12	21	0.0658	0.0736	0.0077
12-13	19	0.0596	0.0767	0.0172
13-14	19	0.0596	0.0768	0.0172
14-15	27	0.0846	0.0741	0.0105
15-16	20	0.0627	0.0694	0.0067
16-17	23	0.0721	0.0632	0.0089
17-18	14	0.0439	0.0562	0.0123
18-19	15	0.0470	0.0489	0.0019
19-20	15	0.0470	0.0418	0.0052
20-21	14	0.0439	0.0351	0.0088
21-22	9	0.0282	0.0290	0.0008
22-23	10	0.0313	0.0237	0.0077
23-24	6	0.0188	0.0190	0.0002
24-25	6	0.0188	0.0152	0.0037
25-28	9	0.0282	0.0284	0.0002
28-31	4	0.0125	0.0128	0.0002

(b) Data for sub-blocks of nibble size (4 bit).

Table 7.4: Comparison of probabilities for observed χ^2 values and expected probabilities, for selected intervals. Data is collected over 20 tests.

Rather than considering the whole 64-bit output difference as a data source, we considered smaller sub-blocks as individual data sources. Our initial findings indicated that, in each test, there seemed to be nibbles that failed the χ^2 test, and could be considered non-random. However, comparing with results from other tests, we saw that this did not hold for the same nibbles each time.

By comparing the frequency of the computed χ^2 values, we were able to conclude that the sub-blocks' distribution do indeed follow what is expected, and that the large χ^2 values seen, do not indicate a non-uniform source. As such, we conclude that the attacker is not able to distinguish the two black boxes by using our χ^2 test.

CHAPTER 8

CHARACTERISTIC TREES

In this chapter we define *characteristic trees*, which are a way of modeling, for both differential- and linear cryptanalysis, the characteristics or linear trails for a certain number of rounds r of a block cipher. The purpose of introducing characteristic trees is to make it well-defined, in a mathematical context, what the cryptanalyst wants to obtain when looking for good characteristics or linear trails.

In our case, the application will be searching for differential characteristics

$$(\alpha, \ldots, \beta),$$

related to the tweaked PRINCE, where α is of a special form. While our study focuses on characteristics for 4 rounds, the method is directly applicable to any number of rounds or inverse rounds, and indeed block ciphers in general.

Finally, we make it clear that our focus is on characteristics. However, there are obvious changes to the methods described here, which makes the methods search for good linear trails instead.

8.1 Definition and Relation to Differential Cryptanalysis

In the following, we give a very brief introduction to terms of graph theory, in order to define characteristic trees, and relate them to differential cryptanalysis.

Definition 16 (Graph)**.** *In the mathematical branch of graph theory, a* graph *is an object with a structure consisting of* vertices*, drawn as dots, and* edges*, drawn as lines connecting the dots.*

These fundamental and inordinately simple mathematical structures found the basis of a rich area of study in mathematics and have countless of applications in a wide range of sciences.

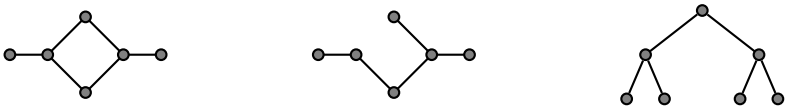
Definition 17 (Path)**.** *A* path *from vertex v to vertex u is defined by a set of edges traversed when going from v to u in the graph.*

Definition 18 (Cycle)**.** *A graph is said to contain a* cycle *if there exists a path starting and ending in some vertex v, which does not use the same edge more than once.*

Definition 19 (Connected graph)**.** *A graph is said to be connected if there exists a path between any two vertices v and u.*

Definition 20 (Tree)**.** *A connected graph which contains no cycle is called a tree. In a tree, a special node can be chosen as a* root *(and the tree is thus said to be* rooted*), and the rest of the tree is typically drawn as descending from this root. The vertices at the bottom of a rooted tree are called* leaves*.*

Examples of mentioned graph types are shown in Figure 8.1.



(a) Graph with cycle

(b) Tree

(c) Rooted tree

Figure 8.1: Selected types of graphs.

In a characteristic tree, we consider a rooted tree, where the root of the tree is labeled with the b-bit input difference α of a differential characteristic $(\alpha, \ldots, \beta)$. The specific characteristic $(\alpha, \ldots, \beta)$ is represented by a path starting in the root α and ending in the leaf β. Recall that an r-round characteristic is represented as a tuple intermediate values, e.g.

$$(\alpha, u_1, \ldots, u_{r-1}, \beta).$$

The natural way of modeling the whole r-round characteristic, including intermediate values, is a path from root to leaf in the characteristic tree with root α. The actions between two differences in the characteristic (typically a round function) is modelled by the edges of the characteristic tree, and the intermediate values will correspond to the vertices occurring on the path in the characteristic tree, which are neither the root α nor the leaf β. To the edges of the characteristic tree we associate the probability for that step of the characteristic occurring.

Recall that given an intermediate value u_i for a differential characteristic

$$(\alpha, \ldots, u_i, u_{i+1}, \ldots, \beta),$$

there are often several possibilities for the next intermediate value u_{i+1}. This is, as we know, due to the S-layer, where an input difference x may have several possible output differences y. In general, if the input to the S-layer for intermediate characteristic value is u_i with m active S-boxes, and active nibble j, $j = 1, \ldots, m$, has c_j possible output differences through the S-layer, then there are

$$C = \prod_{j=1}^{m} c_j$$

possible output differences for the next intermediate value u_{i+1}. Note, that this is a simplified case where we consider only the S-layer, where in reality there are other components of the round function as well. However, it serves the purpose of illustrating the choice of modelling the *characteristic space* for an input difference α as a rooted tree. This is because each vertex of the tree may "spawn" several vertices beneath it

(called *children*), and these naturally model the possible outcomes due to the S-layer. Thus, in the example above, the number of children due to u_i would be C. In this sense, the characteristic tree may potentially grow very large in the number of descendants of the root, the more rounds r the characteristic covers. Note, that the number of rounds r of the characteristic corresponds to the number of edges on a path from root to leaf, which is referred to as the *height* of the tree.

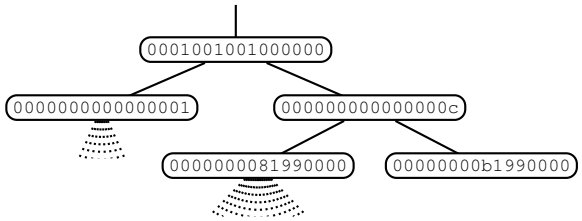


Figure 8.2: Local section of a characteristic tree.

Figure 8.2 shows an example of a local section of a characteristic tree. From the uppermost node we find that

$$(M' \circ SR^{-1})(0001001001000000) = 0000000000000001.$$

The restricted candidate set for nibble difference 1 is {1,c}. Thus, that vertex has two children. The same computation and lookup is performed to find the children of 0000000000000001 and 000000000000000c, etc. Note that at the bottom, the dotted lines indicate the actual number of children, e.g. 0000000081990000 has 16 children. This illustrates well how the size of the characteristic tree can grow quite large over few rounds.

In the past, other approaches to modeling the problem of finding good characteristics as a graph, have been made. Bafghi et. al. model the problem not as a tree, but a *directed* graph where the edges go one way and indicate the start and end state of a characteristic step [3, 2].

So far, we have described how to build a characteristic tree for r-round characteristics using an input difference α. In reality, we often want to find the best characteristic, but care less about what the input difference α is. To that end, it makes sense to talk about a collection of characteristic trees which cover all input differences, one tree for each input.

Having defined characteristic trees, and having made it clear how they model r-round characteristics, we move on to consider what benefits we gain from the definition when it comes to searching for the best characteristics.

8.2 Search Methods for Characteristic Trees

At this point, we want to clarify exactly why the concept of characteristic trees has been introduced. The two main reasons are

1. It serves as a good model to further the conceptual understanding of what characteristics are, and how vast the space of characteristics actually is, and
2. It is defined upon trees, known from graph theory, which have been studied thoroughly for many years, and search methods for trees exist in great numbers.

The hope is that, by taking a problem in the world of cryptanalysis and modeling it as a more generic problem, we may apply known search methods for solving the problem of finding good characteristics. In our application of characteristic trees, we are searching for the best differential characteristics where the input difference α is of the repetitive form. Recall that the expansion E for the address tweak is a bijective function

$$E : \mathbb{F}_2^{27} \longrightarrow \mathbb{F}_2^{64}.$$

Thus, the size of the co-domain 2^{27} is also the number of differences α of the repetitive form, which are of interest for these characteristics. This also means that we get a collection of 2^{27} characteristic trees, one for each possible α. As this is a very large number of trees to look through when the trees potentially can get quite large, a naïve search method will not do, as it will simply take too long to finish.

8.2.1 Related Work

Search methods for characteristics, modeling the search space as a graph, include *ant colony* techniques [2] and *neural networks* [3] proposed by by Bafghi et. al. In [29], Vaudenay considers cryptanalysis of the CS-Cipher, proposed at FSE[1], by modelling the space of characteristics as a graph and using the Floyd-Warshall algorithm [6, Chapter 25] to find the best characteristic. Floyd-Warshall is a dynamic programming algorithm, with complexity cubic in the number of vertices (characteristic states), for solving the *all-pairs shortest path* problem. It translates to finding the best $(\alpha, \ldots, \beta)$ characteristic, but does not guarantee that α is of a specific form, which is required in our case.

8.2.2 Parallelized Brute-Force Search

It should be clear that if an algorithm for a problem such as this can be parallelized, this is desirable, simply due to the speedups obtainable. Unfortunately, not every problem or algorithm lends itself well to parallelization. The naïve approach to solving the characteristic trees problem is a brute-force algorithm, which searches through a full characteristic tree, one at a time, and finds an exact solution to the optimal characteristic. Since all characteristic trees are independent of each other, the brute-force algorithm can be easily parallelized. We have yet to describe how to search through the full characteristic tree, so we turn to that task now.

The method is based on a *depth-first search* [6] (DFS) which is among the first graph algorithms to be taught in any class on the subject. Recall that a characteristic corresponds to a path from root to leaf. In a DFS, the search in a single tree starts at the root and at every branch point, a choice is made and the search continues towards some leaf. Only when a leaf is reached will the search method go back to the previous branch point and try the next child of that vertex. Recall that when traversing a path from root to leaf, we perform transformations on the vertex label (which is a 64-bit difference), and the number of possible S-layer output differences determines the number of children. From looking at the non-zero nibbles of the vertex, the algorithm can use the difference distribution table (Table 6.1) to compute the next factor of the entire characteristics probability, due to that single step, as the weight of the corresponding edge. For example, if the state has 4 active S-boxes and each of the

[1] Fast Software Encryption, a cryptology conference.

substitutions occur with probability 2^{-2}, corresponding to an entry of 4 in Table 6.1, the probability factor contributed by that child is $(2^{-2})^4 = 2^{-8}$. Thus, when traversing the tree during the DFS, the algorithm keeps record of the weight of the current path, and this is what we wish to maximize.

Assume the attacker can execute N threads concurrently, then it is a simple matter of partitioning the 2^{27} characteristic trees into N sets. Each thread of execution works through the trees in its corresponding set, while keeping record of the best characteristic it has found. Finally, when all trees have been searched through, a single thread of execution can compare the results found by the N threads and choose the best one.

When considering this parallelization, it is natural to ask how large a speedup factor is obtainable when using N threads of concurrent execution. To that end, Amdahl's Law [13] gives an upper bound on the speedup based on the *parallel fraction*, the fraction of program code which is executed in parallel. The expected speedup is given by

$$\frac{1}{(1-P)+\frac{P}{N}}, \tag{8.1}$$

where P is the parallel fraction. A plot of Amdahl's law is shown in Figure 8.3. Amdahl's Law can also be used to give an approximation of the parallel fraction, based on experimental data provided by test-runs from a parallelized algorithm implementation.

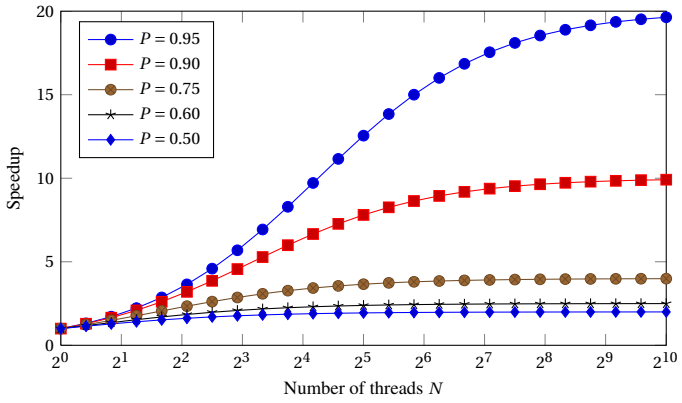


Figure 8.3: Upper bound on speedup as a function of number of threads N for different parallel fractions P.

As is evident from Figure 8.3, the only really good speedups are obtained when approaching a parallel fraction of ≈ 0.90. Parallelizing algorithms with lower parallel fractions often implies that the programmer must be *very* careful when implementing. While concurrent programming potentially offers great speedups of computation, embarking on the task of implementing even easily parallelizable algorithms opens up a world of pitfalls. The programmer must especially be wary of memory shared between threads of execution, and the consequences improper use may lead to [1].

Results

Large as the search space is, with its 2^{27} characteristic trees, our implementation of a parallelized brute-force search is able to completely exhaust the search space in

34 seconds, when using $N = 24$ concurrent threads of execution and considering the restricted candidate sets. This result was obtained on a Linux machine using an AMD Opteron(tm) Processor 6168 processor. The best characteristic found is a 4-round characteristic

$$\alpha \xrightarrow{S^{-1}\circ M'} u_1 \xrightarrow{S^{-1}\circ M'\circ SR^{-1}} u_2 \xrightarrow{S^{-1}\circ M'\circ SR^{-1}} u_3 \xrightarrow{S^{-1}\circ M'\circ SR^{-1}} \beta,$$

with $\alpha \in \Gamma$, using a total of 28 active S-boxes. This result is obtained when considering just the restricted candidate sets. Consequently, we may readily establish that the best found characteristic using this method has probability

$$\begin{aligned} P\left[(\alpha,\ldots,\beta)\right] &= \left(2^{-2}\right)^{28} \\ &= 2^{-56}. \end{aligned}$$

Returning to parallelization, we tried running the same program using just $N = 1$ thread of execution. In this case the, algorithm finished searching through all 2^{27} characteristic trees in 553 seconds. We may now use Amdahl's Law to approximate the parallel fraction of the program, using these experimental data. The speedup obtained here, using 24 threads of execution, is set equal to (8.1), and we may solve for the parallel fraction P,

$$\begin{aligned} \frac{553}{34} &= \frac{1}{(1-P)+\frac{P}{24}} \\ \Leftrightarrow P &= \frac{24\left(\frac{553}{34}-1\right)}{\frac{553}{34}(24-1)} \\ &\approx 0.98. \end{aligned}$$

This shows that the implementation of the brute-force algorithm has a very good parallel fraction.

One thing worth noting from running the brute-force algorithm for $N = 1$ threads was, that the best characteristics found (those with probability 2^{-56}), are found quite fast, in fact within a few seconds, making the order the trees are considered, interesting. In our implementation, the roots of the characteristic trees are constructed by iterating over a counter $t = 1, 2, \ldots, 2^{27} - 1$. The root of the characteristic tree is then constructed as

$$\alpha = (t \parallel t \parallel t_{[10]}).$$

The fact that the best characteristics found are discovered early in the search, indicates that they are found for $\alpha \in \Gamma$ constructed using small values of t. In fact, all the characteristics found, using 28 active S-boxes, use just 3 different starting differences from the set

$$\{0000808000101000, 0000888000111000, 0008800001100000\} \subset \Gamma.$$

The values of t corresponding to the α above are

$$t \in \{1028, 1092, 17408\},$$

confirming that the best characteristics are indeed found early.

The best characteristics found in the megabox search method of Section 6.6.2 used 32 active S-boxes and had a probability of 2^{-64}. While the best characteristic found here certainly have a higher probability, the natural question to ask is: are there any characteristics that are better yet? Recall that for the megabox search method of Section 6.6.2, we attempted to use the extended candidate sets. Doing so, the average number of candidates for a non-zero input differences goes up from 1 to 7, and we make a rough estimate that the number of children of any non-leaf vertex of a characteristic tree goes up by roughly a factor 7.

When the characteristic trees become as large as they do, when using the extended candidate sets, the parallelized brute-force search described in this section, is too naïve. Next, we turn to a slightly more sophisticated search method.

8.2.3 Parallelized Pruning Depth-First Search

The search method we describe here searches through each characteristic tree in the same way, just as it was the case for the brute-force search, and thus offers itself well for parallelization. A thread of execution will find the best result within a single tree. In the brute-force search of the previous section, all threads finished looking at their assigned set of trees, before the best result from each thread was compared to the others. In this algorithm, a *global best* will be maintained, and *during execution* the global best is used by each thread to determine if the current path may prove fruitful. How this is determined will be described in the following.

As the characteristic tree is potentially very large, we want to look at as small a part of it as possible. To that end, the search algorithm may implement the useful observation, that at some point on a path from root to leaf, the accumulated probability for the current path of a characteristic may become lower than the global best. If that should happen, the algorithm can safely discard that part of the tree and everything beneath it, because pursuing it further would always result in a characteristic worse than the global best.

This way of pruning the characteristic tree, in order to identify bad branches at an early stage, is virtually the idea used by Matsui in 1995 to find the best differential- and linear characteristics for DES [22]. One might say that the algorithm described here *learns* about characteristics of improving quality, across several characteristic trees, during its execution. In order to implement the restrictive search which uses the global best characteristic to prune off bad branches, a simple modification to the brute-force approach of the previous section suffices.

Having said this, there is a pitfall when it comes to parallelizing the code. The global best solution should now be accessible by all N threads of execution for both reading and writing. As mentioned, one must be cautious with shared memory when it comes to concurrent programming, because otherwise inconsistent states of shared memory may occur between the threads. Fortunately, with our choice of the OpenMP[2] C++ library, declaring variables as shared memory is relatively simple [9].

Results

For our runs of the pruning DFS-like algorithm, we have considered the extended candidate sets. As such, each non-zero nibble has *a lot* more candidates, in fact 7 on average c.f. the difference distribution table for PRINCE (Table 6.1), and consequently the characteristic trees have grown much larger.

[2] OpenMP API specification home page at `www.openmp.org`.

The result from our implementation show that using the search algorithm on our now much larger characteristic trees, the algorithm runs for a long time without finding a good characteristic. We observe that within seconds, a characteristic of probability 2^{-95} is found, but after ten minutes, no better characteristic is found. Compared to characteristics of probability 2^{-56} found using the former search method, this is not satisfactory.

For his algorithm described in [22], Matsui found a good DES characteristic to use as a starting solution for the search algorithm. This way, the algorithm is able to quickly prune away not just the bad characteristics, but also those that, in the probabilistic sense, are mediocre.

The hope in providing a starting solution for the search method is, that the processing of each characteristic tree will be sped up significantly, in that hopefully many branches can be cut off early. Unfortunately, our results show this is not the case. After having provided the characteristic of probability 2^{-56} as a starting global best, we tried running the algorithm using just $N = 1$ thread. Doing so, it is evident that the executing thread stays within the very first characteristic tree for impractically long time.

8.3 Summary

At this point we conclude, that using the two search methods for characteristic trees described in this chapter, we are not able to find four-round characteristics for the tweaked PRINCE of a probability higher than 2^{-56}. This was when using the parallelized brute-force search method, when considering the restricted candidate sets. This result improves on the one found for the megabox analysis of the tweaked PRINCE scenario in Section 6.6.2.

When we used the extended candidate sets, our search methods were found to be too naïve to yield any results. It seems we need a better approach, which we will turn to in Chapter 9. First, however, we provide in Section 8.4 some results on the sizes of the characteristic trees we are searching through. This should provide an insight into why remodeling the problem is a necessity.

8.4 Sizes of Characteristic Trees

This section serves to shed some light on why the parallelized pruning depth-first search method, described above, did not perform well and would run for a long time without progressing. Specifically, we try to clarify just how large the problem space, i.e. the sizes of the characteristic trees are.

To that end, we give an analysis in the following which gives bounds on the expected sizes of the trees, under the special condition that the linear layer M is taken away from consideration. This is done to make the analysis feasible, but a fair assumption is that having the M layer in place will generally increase the size of the trees.

Lemma 3. *In the setting with the linear layer M removed, for an input difference string $\mathcal{V}$ with m non-zero nibbles, the* expected *number of possible output differences through the S-layer is* 1 *for the reduced candidate set and 7^m for the extended candidate set.*

For a proof of Lemma 3, see Appendix D.

Theorem 3. *If we consider r rounds of* PRINCE*, but with the removal of the linear layer M, a characteristic tree has an* expected *size of*

$$\sum_{i=0}^{r} c^i,$$

where c is the expected number of output differences for an input difference state with m active nibbles.

Proof. Note first, that as we consider an analysis where the linear layer M is not in place, the number of non-zero nibbles m is constant across all vertices. Thus, as of Lemma 3, the expected number of children for any vertex is $c = 1$ in the reduced candidate case and $c = 7^m$ in the extended case. As such, the root itself accounts for 1 vertex. At level 1 the expected number of vertices is c, at level 2 it is c^2, etc. In general, for r rounds, this gives an expected number of vertices of

$$c^0 + c^1 + \cdots + c^r = \sum_{i=0}^{r} c^i$$

□

In the restricted candidate case, Theorem 3 surprisingly implies that the expected size of a characteristic tree for r rounds is 5. In the extended candidate case, we find that in a characteristic tree with $m = 3$ active nibbles in the root, the expected number of vertices in the characteristic tree is 13,881,758,801. Without proof, we state that for a randomly chosen 16-nibble difference, the expected number of active nibbles is 15. In this case, the expected size of the latter characteristic tree is $\approx 2^{168.44}$.

In the same thread, the following Corollary makes a statement about an upper bound on the expected size of characteristic trees.

Corollary 1. *When considering removal of the linear layer M, upper bounds on the expected size of a characteristic tree are* 2^{64} *in the reduced candidates case and* 2^{192} *in the extended candidates case.*

Proof. The proof is a direct application of Theorem 3, where we substitute $c = 2^{16}$ corresponding to 16 active nibbles all with 2 candidates for the reduced candidates case, and $c = 8^{16}$ corresponding to 16 active nibbles with 8 candidates each, for the extended candidates case. □

Corollary 1 helps us understand why switching from the reduced- to the extended candidates set complicates the search through the characteristic trees tremendously. Next, we describe a way of remodeling the characteristic trees into another graph-based model of the characteristic space, which seems better suited for searching, when considering extended candidate sets.

CHAPTER 9

CHARACTERISTIC GRAPHS AND RANDOMIZED SEARCH HEURISTICS

The characteristic tree description of Chapter 8 serves as a nice way of describing differential characteristics (or linear trails, for that matter), and the problem of finding the best ones, in a frame of a well-studied computer scientific problem. However, when dealing with a problem of a size as massive as this one (when considering the extended candidate sets), the characteristic tree description has a fatal weakness when it comes to actually searching for the best characteristics. This weakness is the result of a combination of two things:

- When we consider the extended set of S-layer candidates for our search, the size of each characteristic tree grows tremendously, as we saw in Section 8.4.
- There may potentially be large parts of characteristic trees that are repeated across each tree considered. As we shall see in this chapter, this is a redundancy that can be removed by a remodeling of the problem.

With the motivation for changing to a new representation in place, we turn to describing a transformation from the characteristic *tree* model to a characteristic *graph* model.

9.1 Description of Characteristic Graphs

One of the major issues with characteristic trees was the potentially vast amount of duplicate vertices. Note that indeed within a *single* characteristic tree, there could potentially be a large number of duplicates. Across all $|\Gamma| = 2^{27}$ characteristic trees, the number will be much greater. To that end, we define a new type of structure to search through, a graph which is not a tree. However, the edges have directions, and can only be traversed in the direction indicated by the edge. This does not exclude, however, that both edges (a, b) and (b, a) may be present, for vertices a and b in the graph.

The *characteristic graph* is obtained by taking the description of all characteristic trees and *contracting* them, such that for any two duplicate vertices v and v' representing the same 64-bit difference, we simply reduce them into one vertex having *both* the edge set of v and v'. When doing this, we end up with a *directed* graph by induction, with the important property that it has no duplicate vertices (in the sense that no two

vertices represent the same 64-bit difference). The consequence of this is going from a large set of *very* large trees to a single graph, for which the size is just a fraction of the combined size of the trees.

With the new definition of characteristic graphs, there is no longer a concept of a root. The concept of a direction in the graph still exists, however, due to the directions of the edges. The problem of finding the best differential characteristic $(\alpha, \ldots, \beta)$ is now very similar to a well-studied problem AllPairShortestPath(G), in which the problem is to find the shortest path between all pairs of vertices of a possibly directed graph G, under a certain weight function on the edges. The difference from our problem, in the setting of characteristic graphs, is that:

- We do not want to find the shortest (or lightest in the case of a weight function on the edges), but rather the *heaviest* path of a certain length r (we will use $r = 4$-round characteristics), and
- We are not interested in finding this for *all* pairs of vertices, in the case of characteristics for the tweaked PRINCE, but rather a subset of all pairs

$$\{(\alpha, \beta) \mid \alpha \in \Gamma\}.$$

With the new model in place, we move on to describe a generic way of searching for solutions to hard problems, which in our case will be applied to search for good characteristics, not just for the tweaked PRINCE, but four rounds of PRINCE in general.

9.2 Randomized Search Heuristics

Many problems in real life can be modeled as *optimization problems*. These are problems that are, considering the size of the problem in question, *hard* to solve. What it means for a problem to be hard to solve, is answered within the study of *complexity classes*, which is beyond the scope of this thesis. However, we may very informally state that it involves two important classes of *decision problems* called $\mathcal{P}$ and $\mathcal{NP}$. Some problems have decision versions that can be proven $\mathcal{NP}$-complete, and these problems are generally thought to be harder than other problems. Whether this is really so depends on the famous $\mathcal{P}$ versus $\mathcal{NP}$ question.

Due to the apparent hardness of some optimization problems (i.e. those for which the decision problem is $\mathcal{NP}$-complete), deterministic algorithms for finding exact solutions become impractical for interesting problem sizes. To that end, the study of *randomized search heuristics* deals with the design and analysis of highly efficient algorithms for optimization problems.

Ideally, for some specific optimization problem, one would like to:

1. Analyze the problem,
2. Design an efficient algorithm for solving the problem, and
3. Prove the algorithm's correctness and efficiency.

In real-world scenarios, however, it might be the case that:

- There is a lack of resources for obtaining the ideal situation above, e.g. financing, time or knowledge,
- A good solution, rather than an optimal one, might be satisfactory, or
- The specifics of the problem are unknown, e.g. it may be given as a black box problem.

In any of these cases, an off-the-shelf heuristic which is applicable to generic optimization problems is often a good choice of alternative. In the following, we present three randomized search heuristics which are readily applicable to our optimization problem of finding good characteristics and differentials.

The randomized search heuristics we will describe are defined using two basic concepts:

- A search space S, i.e. a set of possible solutions, and
- An objective function $f : S \longrightarrow \mathbb{R}$.

We make a distinction between minimization- and maximization problems. The general goal for a search heuristic is finding the global minimum respectively maximum of f. The important assumption in the black-box scenario is, that we can only gain information about f by evaluating it in a number of points of the domain S. The randomized search heuristic is a description of which points of S and in which order, for which to evaluate f.

Next, we introduce two concepts of randomized search heuristics which we shall use in our description.

Definition 21. *In the context of a randomized search heuristic, a* state *is an element of the domain S, in which the search heuristic may find itself, at a certain point of time during execution.*

Definition 22. *The* neighborhood *of a state $s \in S$, defined at a certain time (implemented as e.g. iteration number) t, is a subset of S of states, that are feasible as potential states for time $t+1$. The definition of a neighbourhood is highly problem-dependent.*

With the definition of a state and neighborhood in place, we may describe an execution of a randomized search heuristic as a finite sequence of states, where each state is chosen from the neighborhood of the preceding state.

As mentioned, we will consider three search heuristics. While all three are very similar, they can be ranked by simplicity, and we start with the simplest.

9.2.1 RANDOMIZED LOCAL SEARCH

Presented as Algorithm 4 is the first randomized search heuristic, RANDOMIZED LOCAL SEARCH or RLS for short. The heuristic uses a starting state, which may be chosen e.g. at random. At each iteration step it picks a state from the neighborhood of the current state (we shall from now on refer to this as a *candidate state*, to avoid ambiguity between the term *neighbor* used in graphs). If the objective value f for the candidate state is at least as good as that of the current state, the algorithm switches to the new state. This behavior is iterated until a stopping condition is met. The specification of the stopping condition is not in the scope of the search heuristic, but is commonly tuned for each problem.

One important thing to notice about the RLS is, that it does not allow worsening of state objective value over time. In other words, using RLS, the sequence of $f(s)$ for states $s \in S$ will be non-decreasing. At first, this seems like an entirely good thing, however it may be the case that during the course of RLS, it gets stuck in a *local* minimum/maximum in the search to find the *global* minimum/maximum. This can happen because of the way neighborhoods are defined as true subsets of S; not every state it obtainable from every state. Consequently, it may be that from some state s,

Algorithm 4 RANDOMIZED LOCAL SEARCH (RLS)

```
1: s ← starting state
2: while stopping condition not met do
3:     s' ← random state from the neighborhood of s
4:     if f(s') ≥ f(s) then
5:         s ← s'
```

the algorithm will *need* to temporarily go to a state s' with a worse objective function value, to be able to progress further towards the global minimum/maximum.

9.2.2 METROPOLIS ALGORITHM and SIMULATED ANNEALING

To get around the potential problem of getting stuck a local minimum/maximum, two additional randomized search heuristics are presented. The two are *very* similar, and the pseudo-code for both heuristics is presented as Algorithm 5. Note the introduction of the iteration number t in the new heuristics. The stopping condition may, or may not, depend on the value of t by allowing the heuristic to run for a maximum number of iterations. The important difference in Algorithm 5 from RLS is, that the heuristic may actually switch to an inferior state s' at time $t+1$ with probability

$$p = e^{\frac{f(s')-f(s)}{T_t}} \tag{9.1}$$

in the maximization case. In the case of a minimization problem the terms in the numerator of the exponent are swapped. The parameter T_t of (9.1) is called the *temperature* of the search heuristic. It is this parameter that gives rise to two different search heuristics:

- When T_t is constant, the algorithm is called METROPOLIS ALGORITHM, and
- When T_t depends on the current iteration number t of the heuristic, the algorithm is called SIMULATED ANNEALING[1].

Algorithm 5 METROPOLIS ALGORITHM and SIMULATED ANNEALING

```
1: s ← starting state
2: t ← 0
3: while stopping condition not met do
4:     s' ← random state from the neighbourhood of s
5:     if f(s') ≥ f(s) then
6:         s ← s'
7:     else
8:         s ← s' with probability e^((f(s')-f(s))/T_t)
9:     t ← t + 1
```

9.3 Application in Characteristic Search

In this section we describe how we apply the three randomized search heuristics RLS, METROPOLIS ALGORITHM and SIMULATED ANNEALING to the problem of finding

[1] The name comes from similarities with annealing in matallurgy.

good differential characteristics and differentials (or linear trails and hulls). Rather than going into implementation specifics, we focus on how to define the four major components of the search heuristics in the context of our problem:

- States,
- Neighborhood of a state,
- How to evaluate the objective function f, and
- Stopping condition.

9.3.1 Definition of States

Recall that we will be searching for 4-round characteristics, and hence paths of length 4 in characteristic trees. Thus, for our randomized search heuristic, it makes sense to define as the solution space S the set of all directed paths of length 4, starting in a vertex α. For the tweaked PRINCE scenario, α will be of repetitive form. Thus, a state $s \in S$ is going to be such a path, defined by a length 5 sequence of 64-bit differences:

$$s = (s_0, s_1, s_2, s_3, s_4).$$

For a single instance of the search heuristic, we will fix the starting vertex $s_0 = \alpha$, so it is unchanged during the course of the heuristic. The reasoning behind this is, that for tweaked PRINCE, we have 2^{27} possible choices of α. We want to ascertain that by the end of the search, we have considered a portion of characteristics, where the starting vertex is any of the possible α. As such, we run the search heuristic within the scope of a fixed starting vertex until a certain stopping condition is met, before trying the next α. Meanwhile, we iterate over possible α with an outer loop, effectively running an instance of the search heuristic, possibly more than once, for each $\alpha \in \Gamma$.

9.3.2 Objective Function

Recall that the weight of a directed edge (s, s') in the characteristic graph corresponds to the probability of having s' as output difference, through the S-layer, for the input difference $L(s)$, where L is the relevant linear layer. Our sole interest is maximizing the total characteristic probability, so to evaluate a state $s = (s_0, s_1, s_2, s_3, s_4) \in S$, we define as our objective function $f : S \longrightarrow \mathbb{R}$:

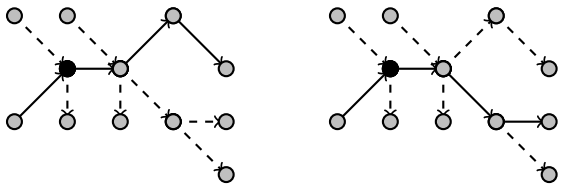
$$f((s_0, s_1, s_2, s_3, s_4)) = \sum_{i=0}^{3} P\left[(s_i, s_{i+1})\right], \tag{9.2}$$

where $P\left[(s_i, s_{i+1})\right]$ is $\log_2$ of the probability associated with that particular 1-round characteristic. Why we use $\log_2$ is explained in Section 9.3.5.

9.3.3 State Neighborhoods

As explained, the randomized search heuristics do not explicitly define neighborhoods of states, as this concept is very problem-dependent. For our problem, states are paths of length 4 of a directed graph. As such, the neighborhood should be a set of such states as well. The definition of a neighborhood is sometimes very straightforward in simple cases. In other cases, however, the choice is not so obvious, and indeed experimenting with definitions of neighborhoods may provide very varied results even when using the same search heuristic.

As described above, for a state s starting in vertex α, we would like our neighborhood to stay within paths starting in α. One immediate idea would be to choose one vertex of the state at random (leaving out the fixed starting vertex α), and replacing it with a randomly chosen vertex reachable from the predecessor vertex of the replaced vertex. This, however may render the state impossible, because the successor vertex of the replaced vertex may not be reachable through any edge from the newly chosen vertex. What we have chosen to do in our implementation is instead to choose a vertex $s_i \neq s_0$ at random, remove the remainder of the path $s_i, \ldots, s_4$, and replacing it by another feasible sub-path of the required length. The choice of a candidate state from a state $s \in S$ is illustrated in Figure 9.1. The solid edges are the edges that are part of the state path, while dashed edges are other edges in the graph. The solid vertex is the vertex randomly chosen, for which to replace the remaining sub-path by a randomly chosen new path.



(a) State s at time t.

(b) Randomly chosen candidate state s' for time $t+1$.

Figure 9.1: Illustration of choosing a candidate state in the randomized search heuristic.

9.3.4 Stopping Condition

As described, our implementation will consider each possible α starting vertex of a path, in turn, possibly trying each more than once. For each fixed α, we will initialize a randomized search heuristic, and thus need a stopping condition, both for each instance of the search heuristic, but also for the outer loop iterating over α. In our case, we have chosen to take as input to the search algorithm two parameters:

- A number of seconds for which to run. This is used to control how long we iterate over α in the outer loop.
- A number of iterations for which to run the randomized search heuristic, for each fixed α, still obeying the running time stopping condition above.

9.3.5 A Note on Implementation

For our search heuristics, we are considering paths of length 4, and want to maximize the weight of the path, equal to the probability of the corresponding characteristic. However, as the probability associated with each edge of the characteristic graph is always a power of two (specifically a product of powers of 2^{-2} and 2^{-3}), we choose to implement our heuristics to just work in $\log_2$ of the probabilities. This allows us to add instead of multiply and work in integers. This will, without a doubt, have an influence of the choices for the temperature parameter later on, but will have no effect on how well the heuristics will perform.

9.4 Results

In this section we present results of using randomized search heuristics in the search for differential characteristics *and* differentials. These apply for four rounds of PRINCE in the following three scenarios:

- Four regular rounds of PRINCE (denoted by $4 \times \mathscr{R}$),
- Four inverse rounds of PRINCE (denoted by $4 \times \mathscr{R}^{-1}$),
- Four rounds in the address tweak scenario.

For the two former scenarios, it does, of course, not make sense to talk about a starting vertex α for the path, of repetitive form. In these two scenarios, the starting vertex α is allowed to take on any value in $\mathbb{F}_2^{64}$, so for them we choose α at random each time.

During the course of each heuristic, the implementation is made such that for each characteristic $(\alpha, \ldots, \beta)$ with a probability p above some predetermined bound, the pair (α, β), along with the probability p, is stored in a hash table. This allows us, in the end, to look for differentials with several characteristics in them, and for our results we will present both the best characteristic and the best differential found.

Our results are presented in sections, organized by the randomized search heuristic used. The first results are for the RLS algorithm, which is the simpler of the three heuristics described.

9.4.1 Results for RLS

Some of the results found here are carried on to the implementations of the METROPOLIS ALGORITHM and SIMULATED ANNEALING, as these heuristics can be considered extensions of RLS.

When testing our implementation of RLS, it quickly became evident that, depending on the scenario tested, the implementation should include a restriction on the starting vertex for some path α, for which we start the randomized search heuristic. Generally, when we let a counter $t = 1, \ldots, 2^{27} - 1$, and construct α as $\alpha = (t \parallel t \parallel t_{[10]})$, many of these starting vertices will have many active S-boxes in the first round. Those heuristic instances are generally off to a bad start, because each characteristic found therein, will have a low probability associated with them, already after the first round.

	Tweak		$4 \times \mathscr{R}$		$4 \times \mathscr{R}^{-1}$	
$\#\mathscr{A}(\alpha)$	Best char.	Best diff.	Best char.	Best diff.	Best char.	Best diff.
2	None[1]	None[1]	2^{-32}	2^{-32}	2^{-37}	2^{-37}
3	None[1]	None[1]	2^{-32}	2^{-32}	2^{-48}	2^{-48}
4	None[1]	None[1]	2^{-34}	2^{-34}	2^{-42}	2^{-42}
5	None[1]	None[1]	2^{-33}	2^{-33}	2^{-40}	2^{-40}
None	None[1]	None[1]	2^{-34}	2^{-34}	2^{-44}	2^{-44}

Table 9.1: Results for varying bounds on the number of active S-boxes in α, using the RLS heuristic.

By experiments, we found that for the $4 \times \mathscr{R}$ and $4 \times \mathscr{R}^{-1}$ scenarios, putting a suitable restriction on the number of active S-boxes in the starting vertex α of at most 3 or 4, gives us better results faster. Table 9.1 gives results for the experiments. Results are shown for each of the three scenarios under consideration, and the entries denote the

[1] Results invalidated due to implementation error. Please contact author if new results are required.

probabilities for the best found characteristics and differentials. The quantity $\#\mathscr{A}(\alpha)$ denotes the upper bound used, on the number of active S-boxes of the starting vertex α. The results were found running RLS for 10 minutes with 60.000 iterations per run of the heuristic in the inner loop. The choice to use 60.000 iterations was found to give good results, by experiments, and we chose to use this number for all our results. We found that in the tweak scenario, better results are found when imposing no restriction on the number of active S-boxes in α. Consequently, we have chosen for this scenario to use no limit, while for the other two scenarios to use the restriction $\#\mathscr{A}(\alpha) = 3$ for all of our results in the following.

	Tweak		$4 \times \mathscr{R}$		$4 \times \mathscr{R}^{-1}$	
Experiment	Best char.	Best diff.	Best char.	Best diff.	Best char.	Best diff.
1	None[1]	None[1]	2^{-32}	2^{-32}	2^{-43}	2^{-43}
2	None[1]	None[1]	2^{-34}	2^{-34}	2^{-41}	2^{-41}
3	None[1]	None[1]	2^{-33}	2^{-33}	2^{-41}	2^{-41}
4	None[1]	None[1]	2^{-34}	2^{-34}	2^{-43}	2^{-43}
5	None[1]	None[1]	2^{-33}	2^{-33}	2^{-45}	2^{-45}
Average	None[1]	None[1]	2^{-33}	2^{-33}	$2^{-41.96}$	$2^{-41.96}$

Table 9.2: Results for the RLS heuristic. The results were found running for 10 minutes with 60.000 iterations per run of the heuristic in the inner loop.

Table 9.2 gives results found for the three scenarios, after running five experiments. We will later compare these results to those found using the two other search heuristics.

9.4.2 Results for METROPOLIS ALGORITHM

Recall that going from RLS to METROPOLIS ALGORITHM, we introduce the possibility of state worsening. As such, the objective function value may now decrease over time. For the METROPOLIS ALGORITHM, the temperature T_t is constant. Note that as T_t tends to zero from above, the METROPOLIS ALGORITHM becomes equivalent to the RLS. In practice, the temperature is adjusted using sample runs, which usually show temperatures which are better than others. The sample runs can, of course, provide outlying results, being provided by a randomized search heuristic. In our case, we have collected several samples to make sure our choice is just.

Table 9.3 shows results found using varying temperatures T_t for the METROPOLIS ALGORITHM in the address tweak scenario. When the temperature T_t approaches 0 from above, the acceptance probability of an inferior state tends to 0. As such, the results of our temperature experiments with the METROPOLIS ALGORITHM of Table 9.3 indicates that RLS will generally perform better than METROPOLIS ALGORITHM. In our case, we chose to use $T_t = 0.1$ as the constant temperature for the METROPOLIS ALGORITHM, because tests indicate this gives the best results.

Table 9.4 gives the probability of accepting an inferior state $s' \in S$ using a temperature $T_t = 0.1$, for certain degrees of inferiority. Please note that the objective function f uses $\log_2$ for the characteristic probabilities. For example, if

$$f(s) = -50, \quad \text{and}$$
$$f(s') = -52,$$

[1] Results invalidated due to implementation error. Please contact author if new results are required.

[1] Results invalidated due to implementation error. Please contact author if new results are required.

Temperature	Best characteristic	Best differential
0.1	None[1]	None[1]
1.0	None[1]	None[1]
5.0	None[1]	None[1]
10.0	None[1]	None[1]
20.0	None[1]	None[1]
40.0	None[1]	None[1]
80.0	None[1]	None[1]

Table 9.3: Results in the address tweak scenario using the METROPOLIS ALGORITHM, for varying temperatures T_t. Each experiment is run for 10 minutes using 60.000 iterations for each run of the heuristic.

the probability of accepting the inferior state s' is approximately $2^{-28.9}$. Here, the probabilities for the states themselves are 2^{-50} and 2^{-52} for s and s', respectively.

$\lvert f(s') - f(s)\rvert$	Approx. acceptance prob.
1	$2^{-14.4}$
2	$2^{-28.9}$
3	$2^{-43.3}$
4	$2^{-57.7}$
5	$2^{-72.1}$

Table 9.4: Probabilities for accepting an inferior state s', using $T_t = 0.1$, from the neighbourhood of state s, as next state.

	Tweak		$4 \times \mathscr{R}$		$4 \times \mathscr{R}^{-1}$	
Experiment	Best char.	Best diff.	Best char.	Best diff.	Best char.	Best diff.
1	None[1]	None[1]	2^{-34}	2^{-34}	2^{-40}	2^{-40}
2	None[1]	None[1]	2^{-32}	2^{-32}	2^{-53}	2^{-53}
3	None[1]	None[1]	2^{-33}	2^{-33}	2^{-45}	2^{-45}
4	None[1]	None[1]	2^{-33}	2^{-33}	2^{-44}	2^{-44}
5	None[1]	None[1]	2^{-33}	2^{-33}	2^{-47}	2^{-47}
Average	None[1]	None[1]	$2^{-32.86}$	$2^{-32.86}$	$2^{-42.18}$	$2^{-42.18}$

Table 9.5: Results for the METROPOLIS ALGORITHM. The results were found using $T_0 = 0.1$ for 10 minutes with 60.000 iterations per run of the heuristic in the inner loop.

In the same style as the results for RLS, we present in Table 9.5 the results found for five experiments, using the METROPOLIS ALGORITHM. The likeness of the results are conjectured to mainly be due to the low temperature $T_t = 0.1$ used. The probability, in a single iteration, that METROPOLIS ALGORITHM will switch to an inferior state is *very low* for $|f(s') - f(s)| > 1$. With the large number of iterations however, the possibility of escaping from a potential local maximum still exists.

9.4.3 Results for SIMULATED ANNEALING

As described, the difference from the METROPOLIS ALGORITHM is, that for SIMULATED ANNEALING, the temperature parameter T_t is actually a function of the iteration number t. Thus, to implement SIMULATED ANNEALING, we need to specify how to define the temperature function T_t. A first step in the approach to this is understanding how the acceptance probability depends on the temperature.

[1] Results invalidated due to implementation error. Please contact author if new results are required.

When in state s at step t, for a candidate state s' with $f(s') < f(s)$ for step $t+1$, we can define the inferiority of s' as $|f(s') - f(s)|$. Figure 9.2 shows how, for a candidate state s', the acceptance probability decreases exponentially, when approaching a temperature of zero. For higher levels of inferiority, the curve looks more linear. Generally, the idea of SIMULATED ANNEALING is, that the overall quality of the solution should improve over time. Also, over time, the current solution should "trace in" on the optimal solution, so we would like in the beginning to allow larger jumps in the objective value for one state to the next. As the heuristic progresses, the jumps should generally become smaller as we come closer to the optimal solution. Referring to Figure 9.2, this implies having the highest temperature in the beginning, and having it decrease over time, eventually reaching nearly zero.

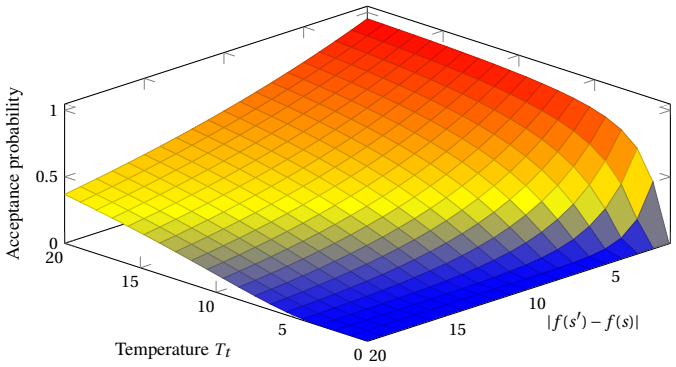


Figure 9.2: Probability of accepting an inferior state for various levels of inferiority $|f(s') - f(s)|$ and varying temperatures.

Two commonly used approaches to reducing the temperature over time, i.e. defining T_t, are:

- Let T_t be linear, in which a certain value is subtracted each time. One can subtract a value corresponding to the starting temperature divided by the number of iterations, so the temperature ends up being zero.
- Let T_t be a function which decreases the current temperature by a certain fraction at each iteration.

For our temperature tuning, we chose the second option by introducing a shaving factor for each iteration. This shaving factor we denote as $\sigma \in]0;1[$, $\sigma \in \mathbb{R}$. We would, however, like that the temperature does not drop too low (specifically to avoid data type underflow), so we set a minimum temperature of 0.1. After each iteration of the heuristic (i.e. after considering each candidate state s'), the current temperature is multiplied by σ. Thus, the series of temperatures during the course of SIMULATED ANNEALING will be defined using a certain starting temperature T_0 and σ as

$$T_{t+1} = \begin{cases} 0.1 & \text{if } \sigma \cdot T_t < 0.1 \\ \sigma \cdot T_t & \text{otherwise} \end{cases}.$$

To tune the temperature parameter, after actually implementing the heuristic with the mentioned function T_t, we applied different combinations of (σ, T_0), and used SIMULATED ANNEALING on the $4 \times \mathscr{R}$ scenario. Our findings, regarding temperature tuning for SIMULATED ANNEALING, were inconclusive. The results did not show

any tendencies as to whether the starting temperature should be high or low, or any hints to what value of σ gives good results. The results from the tests can be found in Appendix E.

When considering the course of the heuristic, it seems however, for SIMULATED ANNEALING that the lower the starting temperature T_0 is, and the faster it drops, the quicker the objective value improves. This again indicates that RLS is superior to SIMULATED ANNEALING for our application, with its inability to switch to inferior states. We conjecture that this is due to two things:

- The massive size of the problem. With characteristic graphs this large, the METROPOLIS ALGORITHM and SIMULATED ANNEALING spend too many iterations on poor solutions while RLS is guaranteed to only improve, and
- The connectedness of the characteristic graph, i.e. the number of vertices reachable from some vertex within a path of length r (for a characteristic of r rounds), may be so large that there is no risk of RLS becoming stuck in a local maximum.

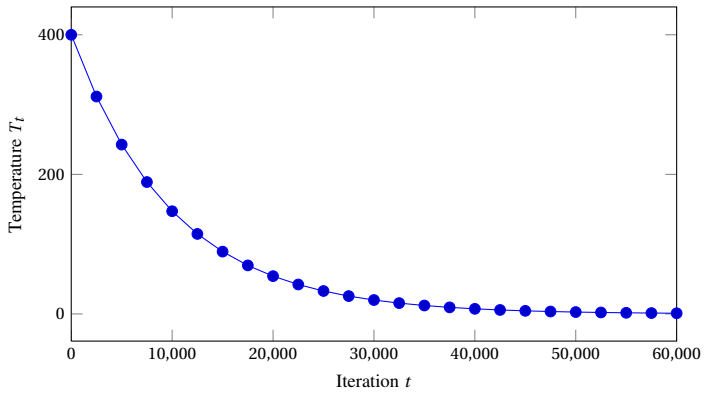


Figure 9.3: Temperature T_t as a function of iteration number t for SIMULATED ANNEALING, using $(T_0, \sigma) = (400.0, 0.9999)$.

For our test runs, we have chosen to use a starting temperature $T_0 = 400.0$ and a shaving factor $\sigma = 0.9999$. Across all three heuristics, we have been using 60.000 iterations for each run of a heuristic with a fixed starting vertex. In this case, the temperature T_t as a function is depicted in Figure 9.3

	Tweak		$4 \times \mathscr{R}$		$4 \times \mathscr{R}^{-1}$	
Experiment	Best char.	Best diff.	Best char.	Best diff.	Best char.	Best diff.
1	None[1]	None[1]	2^{-34}	2^{-34}	2^{-49}	2^{-49}
2	None[1]	None[1]	2^{-35}	2^{-35}	2^{-48}	2^{-48}
3	None[1]	None[1]	2^{-35}	2^{-35}	2^{-48}	2^{-48}
4	None[1]	None[1]	2^{-33}	2^{-33}	2^{-47}	2^{-47}
5	None[1]	None[1]	2^{-34}	2^{-34}	2^{-47}	2^{-47}
Average	None[1]	None[1]	2^{-34}	2^{-34}	$2^{-47.62}$	$2^{-47.62}$

Table 9.6: Results for the SIMULATED ANNEALING heuristic. The results were found using $(T_0, \sigma) = (400.0, 0.9999)$ for 10 minutes with 60.000 iterations per run of the heuristic in the inner loop.

Again, we present results from using SIMULATED ANNEALING in five different experiments. The results are shown in Table 9.6. The general picture when comparing to the METROPOLIS ALGORITHM, is that in the tweak scenario and $4 \times \mathscr{R}^{-1}$ scenario, SIMULATED ANNEALING gives inferior results for both. The results for the $4 \times \mathscr{R}$ scenario is very much like those found using RLS and METROPOLIS ALGORITHM, with an average probability of 2^{-34}.

9.5 Summary

In this chapter we have introduced the concept of characteristic graphs and applied three randomized search heuristics to the problem of finding good characteristics and differentials. Our focus has been on characteristics and differentials, but the methods are equally applicable to linear cryptanalysis as well. In all of our experiments, we have tried to find characteristics for four rounds of PRINCE, in order to compare results with the findings in Chapters 6 and 8.

An important thing to notice is, that even though our search heuristics consider differentials under the extended candidate sets, and thus search through a much denser characteristic graph, the differential effect is poor. While the differential probability is, of course, always as least as large as a corresponding characteristic's probability, we do not see a large improvement anywhere when it comes to the number of characteristics in a differential. This was also the case in the megabox analysis of Chapter 6, where we only considered the restricted candidate sets, and this suggests that obtaining a good differential effect for a differential attack on PRINCE is infeasible.

The differentials found using search heuristics, for the $4 \times \mathscr{R}$ and $4 \times \mathscr{R}^{-1}$ scenarios, are inferior to those of the best possible probability 2^{-32}, found using the megabox approach in Chapter 6. However, we point out that in the tweak scenario, we were able with the RLS to find a characteristic and differential probability of 2^{-472}, which is clearly superior to that of probability 2^{-56} found when searching through the characteristic trees in Chapter 8.

[1] Results invalidated due to implementation error. Please contact author if new results are required.

[2] This result was not presented in the tables in this chapter, but have been found after fixing a minor error in the implementation of the tweak scenario, c.f. the footnotes for results in the tables of Section 9.4

CHAPTER 10

A DISTINGUISHER ATTACK FOR TWEAKED PRINCE

Chapter 7 presented, using a χ^2 statistical analysis, an attempt to distinguish a black box containing the tweaked PRINCE cipher, from a black box containing a random permutation. In practice, the test was to request a certain number of encryptions for a plaintext at different addresses, and seeing if introducing the address tweak introduced some kind of statistical bias in the differences among the output, that could be used to distinguish the tweaked PRINCE from a random permutation. The conclusion was, that this was not the case. In this chapter, we present again a different type of distinguisher attack on the tweaked PRINCE.

10.1 Introduction and Setup

As mentioned, we will again consider a distinguisher attack on the tweaked PRINCE block cipher. The setup is as illustrated in Figure 10.1, where the attacker is given access to two black boxes, BOX_0 and BOX_1. The attacker does not know which of the two boxes contains the tweaked PRINCE cipher, and which contains a random permutation.

An assumption required for the attack is, that the attacker can control at which addresses he wishes to request encryptions and decryptions. In a chosen-ciphertext attack, the attacker should be able to request the plaintext for a chosen ciphertext, but with the address tweak for PRINCE, the plaintext is ambiguous unless the address is specified. As such, the assumption seems fair, however it may be impractical.

Another assumption for the attack is, that the PRINCE black box has chosen its master key k internally, uniformly at random, and that the key is fixed and unchanged during the attack.

As we are considering the tweaked PRINCE, the input to BOX_i, $i \in \{0,1\}$, is a 64-bit plaintext m and a 27-bit address adr, as shown in Figure 10.1. Both of the black boxes produce as output a 64-bit ciphertext c. The assumption mentioned above is reflected in Figure 10.1, in that on the bottom of the boxes the attacker can choose c and the address adr and request the corresponding plaintext. In our model, the attacker is allowed to request both encryption and decryption from each of the black boxes and, as we shall see, merely four encryptions and two decryptions for each black box, suffices to determine which box contains the tweaked PRINCE.

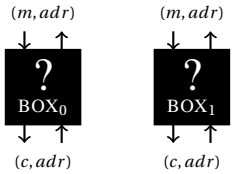


Figure 10.1: Setup for distinguisher attack on tweaked PRINCE.

In Section 6.6.1 we saw, from a differential perspective, the consequences when using the proposed address tweak with the PRINCE block cipher as suggested in Section 3.5. Specifically, as the size of the co-domain of the tweak cipher is 2^{27}, there are just this many possible differences between the outputs of the tweak encryption. When encrypting two equal plaintexts at different addresses in memory, we saw that this leads to the intermediate encryption value difference, after half of the PRINCE cipher, having a certain repetitive form:

$$\alpha = (t \parallel t \parallel t_{[10]}).$$

The attack we present uses this observation, plus the idea that we can find plaintext pairs for which the encryption will cancel out the repetitive difference added in the middle.

10.2 The Attack

In this section, we present the distinguisher attack for the tweaked PRINCE cipher. First we describe the attack in words, and try to give the reasoning behind, and then we give as pseudo-code the description of the attack as Algorithm 6.

To begin with, the attacker requests the encryption for a plaintext x located at two distinct addresses a_0 and a_1. For convenience, we denote the (unknown) difference in the output for the tweak encryption for the two addresses as

$$\alpha = E(T(a_0)) \oplus E(T(a_1)).$$

When applying the remaining part of PRINCE, the attacker obtains two ciphertexts y_0 and y_1 corresponding to the encryption of x at the respective addresses. Note, that the attacker has no knowledge about the difference $y_0 \oplus y_1$.

Next, the attacker requests the decryption of y_0 and y_1 at the *same* address. For the attack, we choose a_0 w.l.o.g. During the decryption of the y_i, $i \in \{0, 1\}$, when the decryption processes reaches the point where the output of the tweak cipher is added to the intermediate value, the difference is α. As the decryption of y_0 and y_1 is working using the same address a_0, the difference after adding the tweak cipher output is still α.

Eventually, the attacker obtains, from the decryption of the y_i at address a_0, plaintexts m_0 and m_1, for which the difference $m_0 \oplus m_1$ is unknown. The observation to make is, that the attacker has obtained plaintexts that, when encrypted at the same address, have an intermediate value difference of α at the point just before the output from the tweak cipher is added. The attacker next requests the encryption of (m_0, a_0) and (m_1, a_1). We know that

$$\alpha = E(T(a_0)) \oplus E(T(a_1)),$$

and as such, when adding the output from the tweak cipher encryption of the addresses to the intermediate encryption values, the difference becomes 0. As the difference is 0 at this point, it stays so until the very end. Eventually, the attacker obtains two ciphertexts $c_0 = E_k(m_0, a_0)$ and $c_1 = E_k(m_1, a_1)$, s.t.

$$c_0 \oplus c_1 = 0.$$

The attack uses the structure of the address tweak added on top of the PRINCE cipher. The main observation is, that the attacker is able to obtain plaintexts m_0 and m_1 for which, at the point just before $E(T(a_0))$ and $E(T(a_1))$ respectively are added, the intermediate encryption value difference is α. As

$$\alpha = E(T(a_0)) \oplus E(T(a_1)),$$

the differences cancel and it becomes 0 for the rest of the encryption.

Algorithm 6 DISTINGUISHER ATTACK ON TWEAKED PRINCE

1: Pick an arbitrary $x \in \mathbb{F}_2^{64}$
2: Pick arbitrarily addresses a_0 and a_1 in $\mathbb{F}_2^{27}$
3: Request $y_0 = E(x, a_0)$ and $y_1 = E(x, a_1)$ from BOX_0
4: Request $m_0 = D(y_0, a_0)$ and $m_1 = E(y_1, a_0)$ from BOX_0
5: Request $c_0 = E(m_0, a_0)$ and $c_1 = E(m_1, a_1)$ from BOX_0
6: **if** $c_0 \oplus c_1 = 0$ **then**
7: Guess that BOX_0 contains the tweaked PRINCE
8: **else**
9: Guess that BOX_1 contains the tweaked PRINCE

From the attack, note that the property that $c_0 \oplus c_1 = 0$ *always* holds for the black box with tweaked PRINCE. The attacker can, with just twice the number of requests indicated in Algorithm 6, try *both* black boxes and verify the guess. The only case where the guess cannot be verified in first instance, is when both black boxes give $c_0 \oplus c_1 = 0$. In this case, the test must be run again using a new plaintext x and addresses a_0 and a_1. However, assuming that the black box containing the random permutation is equally likely to output each value, the probability of this happening is 2^{-64}.

To show our attack works, we present our claim as Theorem 4.

Theorem 4. *Let E_0 be the first part of* PRINCE*, until where the output from the address tweak is added, and let E_1 be the last part of prince, remaining after adding it. Let D_0 and D_1 be the corresponding decryption functions. Also, let $T_0 = E(T(a_0))$ and $T_1 = E(T(a_1))$. Let x, y_0, y_1, m_0 and m_1 be as above.*

Then,

$$E_1(E_0(m_0) \oplus T_0) = E_1(E_0(m_1) \oplus T_1).$$

Proof. We are considering the encryption of plaintext x. First note, that $m_0 = x$, as

$$y_0 = E_1(E_0(x) \oplus T_0),$$

and

$$\begin{aligned} m_0 &= D_0(D_1(y_0) \oplus T_0) \\ &= D_0(D_1(E_1(E_0(x) \oplus T_0)) \oplus T_0) \\ &= D_0(E_0(x) \oplus T_0 \oplus T_0) \\ &= D_0(E_0(x)) \\ &= x. \end{aligned}$$

In our description,

$$y_1 = E_1(E_0(m_0) \oplus T_1).$$

Computing m_1 is a matter of

$$\begin{aligned} m_1 &= D_0(D_1(y_1) \oplus T_0) \\ &= D_0(D_1[E_1(E_0(m_0) \oplus T_1)] \oplus T_0) \\ &= D_0(E_0(m_0) \oplus T_1 \oplus T_0). \end{aligned}$$

Our claim is now verified, as

$$\begin{aligned} E_1(E_0(m_0) \oplus T_0) &= E_1(E_0(m_0) \oplus T_1 \oplus T_0 \oplus T_1) \\ &= E_1(E_0[D_0(E_0(m_0) \oplus T_1 \oplus T_0)] \oplus T_1) \\ &= E_1(E_0(m_1) \oplus T_1). \end{aligned}$$

□

10.3 Implementation

In Section 3.3 we talked about a reference implementation for the PRINCE block cipher. This reference implementation was well suited for adding the address tweak implementation on top, exactly due to the way it follows the specification down to the smallest components.

An implementation was made, which uses the PRINCE reference implementation, with the address tweak encryption on top, as specified by Figure 3.6. With this, the attack described in this chapter is realized by the code in Appendix F. Our tests have verified that by taking the theory of the attack into practice, we are truly able to distinguish the tweaked PRINCE block cipher each time.

10.4 Summary

In this chapter we presented a distinguisher attack on the tweaked PRINCE, which can distinguish two black boxes using just six queries. The probability of failing the attack is negligible, but it is worth mentioning that the attack requires the attacker to be able to control the addresses, at which he requests encryption and decryption. While a distinguisher attack is not as fatal as key recovery, it is still considered a break, within a cryptanalytic frame, as discussed in Section 2.6. Furthermore, the attack described here may have implications on work by Rivest et. al. in [20], where a construction very similar to the tweaked PRINCE is introduced as a secure tweakable block cipher.

CHAPTER 11

CONCLUSION

This master's thesis summarizes the cryptanalysis conducted on a novel lightweight block cipher PRINCE, which was developed by the Crypto Group at the Technical University of Denmark, for the Dutch company NXP Semiconductors.

While the main focus of this thesis lies with the cryptanalytic study of PRINCE, Chapters 2, 4 and 5 were completely general introductions to block ciphers, differential- and linear cryptanalysis, respectively. Introducing these concepts was deemed necessary, in order to fully motivate the choices in the design of PRINCE, and the approaches to cryptanalyzing it.

The purpose of the cryptanalytic case study has been to verify and argue, for the security strength of PRINCE. As PRINCE was designed with focus on being resistant towards the two most powerful attacks on block ciphers, differential- and linear cryptanalysis, it was essential to verify that this is indeed the case.

Our approach to arguing the strength of PRINCE towards differential- and linear cryptanalysis, has been to apply various search methods for finding the differentials and linear hulls of best probability. As one of the search methods, the megabox described in Chapter 6, was restricted to considering four rounds, this was adapted to all other methods, to be able to compare results. The megabox analysis method was possible due to the construction of the linear mixing component M' of PRINCE, which operates on smaller sub-blocks of the state independently of others. The megabox description is especially convenient in showing a lower bound on the number of active S-boxes across four rounds of PRINCE, and thus an upper bound on the best characteristic probability. The randomized search heuristics introduced, RANDOMIZED LOCAL SEARCH, the METROPOLIS ALGORITHM and SIMULATED ANNEALING, are generic algorithms used in optimization problems. In our case, we modeled the problem of finding high-probability differentials in terms of graphs, and applied the search heuristics to maximize the objective value on paths of length four.

Our findings show that the best differential for any four consecutive rounds of PRINCE contains two characteristics with 16 active S-boxes each. Each of these characteristics have a probability of 2^{-32}, thus the differential probability was 2^{-31}. In the case of linear cryptanalysis, the best found linear hull for any four consecutive rounds had just a single trail, using 16 active S-boxes. These findings indicate that PRINCE exhibits poor differential- and linear hull effects, which is a highly desirable feature for the designers. Based on our results in our thorough search for four-round

differentials and linear hulls, we conclude that PRINCE is indeed resistant towards these attacks.

An interesting extension proposed for PRINCE was the address tweak, which enables the use of a plaintexts address in memory to influence the ciphertext. The address tweak for PRINCE implies a property of differences among intermediate encryption values, under certain conditions. When searching for differentials for this scenario, we found the best to have a probability of approximately 2^{-33}, which is impractical for an attack. By applying a statistical χ^2 analysis, we found that an attacker can not distinguish the tweaked PRINCE from a random permutation by this method, given access only to encryption of plaintexts.

In Chapter 10, we presented a distinguisher algorithm in a different model, where the attacker is allowed to control the address at which encryption and decryption takes place. Using just six queries to either of the black boxes in the model, the attacker can distinguish the tweaked PRINCE from a randomly chosen permutation. While this is certainly undesirable, it is not a total break, and the practicality of the assumptions required is unclear. However, the distinguisher attack presented may have implications on the results for tweakable block ciphers, introduced by Rivest et. al. in [20], where a construction very similar to the tweaked PRINCE is used.

While the area of lightweight block ciphers is fairly new, it is starting to receive more attention in the recent years, and outstanding non-proprietary ciphers are emerging. With the innovative symmetric and unrolled design of PRINCE, and its implications of low latency and overhead of decryption implementation, it is our hope that PRINCE will endorse further study of lightweight block ciphers, with metrics other than just chip area.

Appendix A

Prince Linear Layer Matrices

The following are the two linear layer matrices used in the PRINCE block cipher. Recall that M is constructed using M' and the SR operation as $M = SR \circ M'$. The M' matrix is used as the very middle component of PRINCE_{core}, and is thus is required to be an involution to fulfill the α-reflection property.

$$M' = \begin{bmatrix}
0000100010001000000000000000000000000000000000000000000000000000 \\
0100000001000100000000000000000000000000000000000000000000000000 \\
0010001000000010000000000000000000000000000000000000000000000000 \\
0001000100010000000000000000000000000000000000000000000000000000 \\
1000100010000000000000000000000000000000000000000000000000000000 \\
0000010001000100000000000000000000000000000000000000000000000000 \\
0010000000100010000000000000000000000000000000000000000000000000 \\
0001000100000001000000000000000000000000000000000000000000000000 \\
1000100000001000000000000000000000000000000000000000000000000000 \\
0100010001000000000000000000000000000000000000000000000000000000 \\
0000001000100010000000000000000000000000000000000000000000000000 \\
0001000000010001000000000000000000000000000000000000000000000000 \\
1000000010001000000000000000000000000000000000000000000000000000 \\
0100010000000100000000000000000000000000000000000000000000000000 \\
0010001000100000000000000000000000000000000000000000000000000000 \\
0000000100010001000000000000000000000000000000000000000000000000 \\
0000000000000000100010001000000000000000000000000000000000000000 \\
0000000000000000000001000100010000000000000000000000000000000000 \\
0000000000000000001000000010001000000000000000000000000000000000 \\
0000000000000000000100010000000100000000000000000000000000000000 \\
0000000000000000100010000000100000000000000000000000000000000000 \\
0000000000000000010001000100000000000000000000000000000000000000 \\
0000000000000000000000100010001000000000000000000000000000000000 \\
0000000000000000000100000001000100000000000000000000000000000000 \\
0000000000000000100000001000100000000000000000000000000000000000 \\
0000000000000000010001000000010000000000000000000000000000000000 \\
0000000000000000001000100010000000000000000000000000000000000000 \\
0000000000000000000000010001000100000000000000000000000000000000 \\
0000000000000000000010001000100000000000000000000000000000000000 \\
0000000000000000010000000100010000000000000000000000000000000000 \\
0000000000000000001000100000001000000000000000000000000000000000 \\
0000000000000000000100010001000000000000000000000000000000000000 \\
0000000000000000000000000000000010001000100000000000000000000000 \\
0000000000000000000000000000000000000100010001000000000000000000 \\
0000000000000000000000000000000000100000001000100000000000000000 \\
0000000000000000000000000000000000010001000000010000000000000000 \\
0000000000000000000000000000000010001000000010000000000000000000 \\
0000000000000000000000000000000001000100010000000000000000000000 \\
0000000000000000000000000000000000000010001000100000000000000000 \\
0000000000000000000000000000000000010000000100010000000000000000 \\
0000000000000000000000000000000010000000100010000000000000000000 \\
0000000000000000000000000000000001000100000001000000000000000000 \\
0000000000000000000000000000000000100010001000000000000000000000 \\
0000000000000000000000000000000000000001000100010000000000000000 \\
0000000000000000000000000000000000001000100010000000000000000000 \\
0000000000000000000000000000000001000000010001000000000000000000 \\
0000000000000000000000000000000000100010000000100000000000000000 \\
0000000000000000000000000000000000010001000100000000000000000000 \\
0000000000000000000000000000000000000000000000000000100010001000 \\
0000000000000000000000000000000000000000000000000100000001000100 \\
0000000000000000000000000000000000000000000000000010001000000010 \\
0000000000000000000000000000000000000000000000000001000100010000 \\
0000000000000000000000000000000000000000000000001000100010000000 \\
0000000000000000000000000000000000000000000000000000010001000100 \\
0000000000000000000000000000000000000000000000000010000000100010 \\
0000000000000000000000000000000000000000000000000001000100000001 \\
0000000000000000000000000000000000000000000000001000100000001000 \\
0000000000000000000000000000000000000000000000000100010001000000 \\
0000000000000000000000000000000000000000000000000000001000100010 \\
0000000000000000000000000000000000000000000000000001000000010001 \\
0000000000000000000000000000000000000000000000001000000010001000 \\
0000000000000000000000000000000000000000000000000100010000000100 \\
0000000000000000000000000000000000000000000000000010001000100000 \\
0000000000000000000000000000000000000000000000000000000100010001
\end{bmatrix}$$

$$
M = \begin{bmatrix}
0000100010001000000000000000000000000000000000000000000000000000 \\
0100000001000100000000000000000000000000000000000000000000000000 \\
0010001000000010000000000000000000000000000000000000000000000000 \\
0001000100010000000000000000000000000000000000000000000000000000 \\
0000000000000000100010000000100000000000000000000000000000000000 \\
0000000000000000010001000100000000000000000000000000000000000000 \\
0000000000000000000000100010001000000000000000000000000000000000 \\
0000000000000000000100000001000100000000000000000000000000000000 \\
0000000000000000000000000000000010000000100010000000000000000000 \\
0000000000000000000000000000000001000100000001000000000000000000 \\
0000000000000000000000000000000000100010001000000000000000000000 \\
0000000000000000000000000000000000000001000100010000000000000000 \\
0000000000000000000000000000000000000000000000001000000010001000 \\
0000000000000000000000000000000000000000000000000100010000000100 \\
0000000000000000000000000000000000000000000000000010001000100000 \\
0000000000000000000000000000000000000000000000000000000100010001 \\
0000000000000000100010001000000000000000000000000000000000000000 \\
0000000000000000000001000100010000000000000000000000000000000000 \\
0000000000000000001000000010001000000000000000000000000000000000 \\
0000000000000000000100010000000100000000000000000000000000000000 \\
0000000000000000000000000000000010001000000010000000000000000000 \\
0000000000000000000000000000000001000100010000000000000000000000 \\
0000000000000000000000000000000000000010001000100000000000000000 \\
0000000000000000000000000000000000010000000100010000000000000000 \\
0000000000000000000000000000000000000000000000001000100000001000 \\
0000000000000000000000000000000000000000000000000100010001000000 \\
0000000000000000000000000000000000000000000000000000001000100010 \\
0000000000000000000000000000000000000000000000000001000000010001 \\
1000000010001000000000000000000000000000000000000000000000000000 \\
0100010000000100000000000000000000000000000000000000000000000000 \\
0010001000100000000000000000000000000000000000000000000000000000 \\
0000000100010001000000000000000000000000000000000000000000000000 \\
0000000000000000000000000000000010001000100000000000000000000000 \\
0000000000000000000000000000000000000100010001000000000000000000 \\
0000000000000000000000000000000000100000001000100000000000000000 \\
0000000000000000000000000000000000010001000000010000000000000000 \\
0000000000000000000000000000000000000000000000001000100010000000 \\
0000000000000000000000000000000000000000000000000000010001000100 \\
0000000000000000000000000000000000000000000000000010000000100010 \\
0000000000000000000000000000000000000000000000000001000100000001 \\
1000100000001000000000000000000000000000000000000000000000000000 \\
0100010001000000000000000000000000000000000000000000000000000000 \\
0000001000100010000000000000000000000000000000000000000000000000 \\
0001000000010001000000000000000000000000000000000000000000000000 \\
0000000000000000000010001000100000000000000000000000000000000000 \\
0000000000000000010000000100010000000000000000000000000000000000 \\
0000000000000000001000100000001000000000000000000000000000000000 \\
0000000000000000000100010001000000000000000000000000000000000000 \\
0000000000000000000000000000000000000000000000000000100010001000 \\
0000000000000000000000000000000000000000000000000100000001000100 \\
0000000000000000000000000000000000000000000000000010001000000010 \\
0000000000000000000000000000000000000000000000000001000100010000 \\
1000100010000000000000000000000000000000000000000000000000000000 \\
0000010001000100000000000000000000000000000000000000000000000000 \\
0010000000100010000000000000000000000000000000000000000000000000 \\
0001000100000001000000000000000000000000000000000000000000000000 \\
0000000000000000100000001000100000000000000000000000000000000000 \\
0000000000000000010001000000010000000000000000000000000000000000 \\
0000000000000000001000100010000000000000000000000000000000000000 \\
0000000000000000000000010001000100000000000000000000000000000000 \\
0000000000000000000000000000000000001000100010000000000000000000 \\
0000000000000000000000000000000001000000010001000000000000000000 \\
0000000000000000000000000000000000100010000000100000000000000000 \\
0000000000000000000000000000000000010001000100000000000000000000
\end{bmatrix}
$$

APPENDIX B

SETS I_j AND G_j FOR MEGABOX ANALYSIS

Listing B.1 gives the sets I_j and Listing B.2 gives the sets G_j, $j \in \{0,1\}$. These sets represent the *good* input/output differences to the superboxes SB_j, respectively. These are for the megabox analysis where we consider differentials using the restricted candidate cases, i.e. where each S-box substitution holds with probability 2^{-2}.

```
I_j[130] = {
    0x0001,0x0003,0x3300,0x0004,0x3303,0x0404,0x0400,0x0801
    0x0088,0x0008,0x1080,0x0081,0x0080,0x3003,0x1088,0x0011,
    0x1081,0x0018,0x4004,0x4000,0x0010,0x8880,0x3000,0x1000,
    0x8080,0x8081,0x1008,0x4440,0x1001,0x8088,0x8110,0x1010,
    0x1011,0x8018,0x1110,0x7070,0x8011,0x1018,0x8010,0x0507,
    0x8101,0x8100,0x0505,0x8008,0x0707,0x8108,0x0705,0x8001,
    0x8000,0x3030,0x8180,0x0040,0x3033,0x0118,0x0044,0x5050,
    0x0300,0x7050,0x0303,0x0444,0x0110,0x0111,0x0440,0x1180,
    0x1108,0x0181,0x0108,0x0180,0x4404,0x4400,0x0188,0x3330,
    0x1101,0x1100,0x1810,0x0100,0x0101,0x1880,0x1808,0x1801,
    0x8810,0x0810,0x5070,0x4040,0x4044,0x0811,0x1800,0x0818,
    0x0030,0x0888,0x8800,0x8801,0x0033,0x0881,0x0880,0x8808,
    0x0808,0x0330,0x0333,0x0800 };
```

Listing B.1: The I_j set

```
G_j[100] = {
    0x0001,0x0003,0x3300,0x0004,0x3303,0x0404,0x0400,0x0088,
    0x0008,0x1080,0x0081,0x0080,0x3003,0x1088,0x0011,0x1081,
    0x0018,0x4004,0x4000,0x0010,0x8880,0x3000,0x1000,0x8080,
    0x8081,0x1008,0x4440,0x1001,0x8088,0x8110,0x1010,0x1011,
    0x8018,0x1110,0x7070,0x8011,0x1018,0x8010,0x0507,0x8101,
    0x8100,0x0505,0x8008,0x0707,0x8108,0x0705,0x8001,0x8000,
    0x3030,0x8180,0x0040,0x3033,0x0118,0x0044,0x5050,0x0300,
    0x7050,0x0303,0x0444,0x0110,0x0111,0x0440,0x1180,0x1108,
    0x0181,0x0108,0x0180,0x4404,0x4400,0x0188,0x3330,0x1101,
    0x1100,0x1810,0x0100,0x0101,0x1880,0x1808,0x1801,0x8810,
    0x0810,0x5070,0x4040,0x4044,0x0811,0x1800,0x0818,0x0030,
    0x0888,0x8800,0x8801,0x0033,0x0881,0x0880,0x8808,0x0808,
    0x0330,0x0333,0x0800,0x0801 };
```

Listing B.2: The G_j set

APPENDIX C

χ^2 ANALYSIS RESULTS

This appendix provides results for 20 experiments of χ^2 analysis, operating both on nibble and byte levels.

Nibble #	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6	Test 7	Test 8	Test 9	Test 10
0	18.79	10.26	15.43	14.34	12.23	14.12	17,44	11,737	14,592	12,092
1	8.29	26.90	20.29	24.90	19.28	20.20	9,17	16,924	9,009	19,977
2	13.49	14.95	13.02	15.66	12.57	13.31	19,193	14,988	10,189	12,93
3	23.02	8.59	19.01	8.64	15.21	6.73	9,778	19,044	14,938	19,075
4	11.90	22.89	18.36	17.65	29.53	24.73	16,257	18,1	7,518	12,682
5	27.59	10.31	15.06	14.75	8.75	11.79	18,585	19,23	17,502	11,835
6	16.53	21.07	16.00	10.81	10.30	10.97	21,347	6,255	12,849	13,565
7	12.14	12.36	18.34	8.36	9.52	16.24	21,122	7,534	14,14	14,263
8	11.77	17.99	9.90	25.03	22.63	9.45	16,895	9,092	15,421	24,354
9	17.13	9.16	13.41	9.71	28.42	11.67	20,626	10,403	13,5	13,731
10	10.17	8.98	6.61	10.47	14.18	19.81	14,962	9,384	9,211	22,746
11	16.56	14.78	16.18	15.23	9.04	24.26	10,757	13,472	14,25	11,879
12	18.15	16.67	9.76	16.95	10.19	10.44	10,14	7,093	15,063	20,382
13	14.23	9.07	14.96	30.70	18.24	17.08	7,777	23,189	7,711	10,547
14	10.01	17.16	16.79	9.89	9.05	16.73	10,938	18,328	6,516	17,264
15	16.62	6.58	14.91	8.90	19.43	18.09	10,246	3,833	12,389	12,359
Nibble #	Test 11	Test 12	Test 13	Test 14	Test 15	Test 16	Test 17	Test 18	Test 19	Test 20
0	21.586	15.888	15.18	3.498	22.298	9.554	13.805	8.696	23.767	11.259
1	15.285	6.901	14.754	16.703	8.519	12.432	16.496	14.225	21.534	18.81
2	14.87	20.682	27.245	19.177	11.56	16.529	21.744	17.387	15.977	16.435
3	13.521	14.453	10.686	13.077	17.138	24.838	14.808	17.305	12.033	19.925
4	13.241	11.055	22.809	13.905	15.265	19.71	10.086	21.208	10.917	12.014
5	8.726	22.415	6.952	8.821	13.849	4.173	13.203	11.307	15.636	23.18
6	8.538	25.041	15.714	20.775	20.36	12.124	18.12	22.081	8.295	9.163
7	8.809	10.15	10.722	8.89	9.682	7.626	15.408	11.299	14.078	26.434
8	15.402	14.899	21.659	20.153	10.39	20.568	26.247	15.387	26.092	13.357
9	12.887	21.046	23.915	14.156	6.206	25.289	8.115	15.967	19.109	12.945
10	9.452	19.225	7.426	24.63	11.905	20.971	19.635	11.69	9.602	16.631
11	14.312	10.777	9.32	23.185	8.787	6.367	28.343	22.635	15.324	11.783
12	17.275	6.556	8.164	18.442	22.993	22.045	14.939	12.056	16.858	9.537
13	17.018	13.494	11.573	13.964	11.14	17.295	16.39	20.889	18.62	11.56
14	20.229	16.455	16.824	16.654	9.962	12.542	18.473	10.885	13.864	18.934
15	11.609	20.296	11.49	4.704	12.051	14.944	20.149	6.592	11.72	16.474

Table C.1: Computed χ^2 values for 20 experiments using w = 4-bit sub-blocks, counting from most significant to least significant.

Byte #	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6	Test 7	Test 8	Test 9	Test 10
0	268.06	298.09	234.68	243.14	226.23	240.16	240.21	261.15	259.79	265.48
1	255.51	250.60	261.79	251.02	234.21	259.39	223.75	265.85	248.64	238.03
2	243.97	281.51	256.54	248.93	270.68	274.08	276.34	248.70	213.22	249.53
3	260.52	235.86	250.85	272.93	280.51	257.71	268.00	238.34	263.81	230.30
4	247.96	260.69	254.49	275.01	283.09	220.96	284.75	225.01	259.69	296.92
5	289.90	227.57	278.68	279.29	251.70	221.80	215.72	281.57	254.92	291.80
6	249.43	270.84	243.54	264.53	257.02	265.79	237.55	264.28	223.66	264.34
7	272.74	254.50	226.55	250.73	257.21	263.41	247.19	234.58	276.16	250.72
Byte #	Test 11	Test 12	Test 13	Test 14	Test 15	Test 16	Test 17	Test 18	Test 19	Test 20
0	295.34	263.87	253.54	232.77	251.80	237.34	240.23	252.74	280.03	300.67
1	262.78	235.28	282.98	253.82	261.30	252.28	278.11	248.28	280.19	273.89
2	280.28	260.41	235.65	268.98	261.39	263.62	243.85	247.13	258.10	274.10
3	267.92	281.91	271.75	227.87	283.93	251.29	236.12	256.59	268.00	262.08
4	229.16	214.75	264.52	276.81	253.71	273.58	218.41	246.53	261.29	231.24
5	236.04	296.59	224.64	268.96	207.54	226.68	268.66	266.97	231.94	273.16
6	249.61	244.43	271.37	254.30	262.69	269.50	294.15	239.18	207.16	258.27
7	283.09	288.22	241.96	275.97	221.96	232.56	240.88	261.87	284.18	272.08

Table C.2: Computed χ^2 values for selected experiments using w = 8-bit sub-blocks, counting from most significant to least significant.

APPENDIX D

PROOFS ON CHARACTERISTIC TREE SIZES

Lemma 4. *In the setting with the linear layer M removed, for an input difference string $\mathcal{V}$ with m non-zero nibbles, the* expected *number of possible output differences through the S-layer is* 1 *for the reduced candidate set and* 7^m *for the extended candidate set.*

Proof. Consider an input difference string $\mathcal{V} = (v_1, v_2, \ldots, v_n)$ of n nibbles with m non-zero. Let X be a random variable denoting the number of possible output differences, through the S-layer, for input difference $\mathcal{V}$ (i.e. X is the quantity about which the lemma makes a statement). Let I denote the set

$$I = \{i \mid v_i \neq 0\}$$

and let X_i, $i \in I$ be a random variable denoting the number of possible output differences for (non-zero) input nibble v_i. We consider the two cases separately.

Restricted Candidate Case In this case, we find from the difference distribution table (Table 6.1) that

$$X_i = \begin{cases} 0 & \text{with probability } 3/15 \\ 1 & \text{with probability } 9/15\,. \\ 2 & \text{with probability } 3/15 \end{cases}$$

As such, we may readily compute the expected value of X_i as

$$\begin{aligned} E[X_i] &= 0 \cdot \frac{3}{15} + 1 \cdot \frac{9}{15} + 2 \cdot \frac{3}{15} \\ &= 1. \end{aligned}$$

Note that

$$\begin{aligned} X &= X_{i_1} \cdots X_{i_m} \\ &= \prod_{i=1}^{m} X_i, \qquad i \in I. \end{aligned}$$

As the X_i are independent random variables, the expectation of X can be computed as

$$\begin{aligned} E[X] &= E\left[\prod_{i=1}^{m} X_i\right] \\ &= \prod_{i=1}^{m} E[X_i] \\ &= E[X_i]^m \\ &= 1. \end{aligned}$$

as all $E[X_i]$ are equal.

Extended Candidate Case In this case, there are naturally more candidates for each nibble, on average, and we find that

$$X_i = \begin{cases} 6 & \text{with probability } 3/15 \\ 7 & \text{with probability } 9/15 \\ 8 & \text{with probability } 3/15 \end{cases}.$$

Thus,

$$\begin{aligned} E[X_i] &= 6 \cdot \frac{3}{15} + 7 \cdot \frac{9}{15} + 8 \cdot \frac{3}{15} \\ &= 7. \end{aligned}$$

Finally, we find the sought value as

$$\begin{aligned} E[X] &= E\left[\prod_{i=1}^{m} X_i\right] \\ &= \prod_{i=1}^{m} E[X_i] \\ &= E[X_i]^m \\ &= 7^m. \end{aligned}$$

□

APPENDIX E

SIMULATED ANNEALING TEMPERATURE EXPERIMENT DATA

This appendix gives results on findings regarding temperature tests for the SIMULATED ANNEALING randomized search heuristic.

(σ, T_0)	Best characteristic	Best differential
$(0.99, 20.0)$	2^{-33}	2^{-33}
$(0.99, 40.0)$	2^{-33}	2^{-33}
$(0.99, 80.0)$	2^{-33}	2^{-33}
$(0.99, 150.0)$	2^{-32}	2^{-32}
$(0.99, 300.0)$	2^{-33}	2^{-33}
$(0.99, 600.0)$	2^{-33}	2^{-33}
$(0.90, 20.0)$	2^{-32}	2^{-32}
$(0.90, 40.0)$	2^{-32}	2^{-32}
$(0.90, 80.0)$	2^{-33}	2^{-33}
$(0.90, 150.0)$	2^{-33}	2^{-33}
$(0.90, 300.0)$	2^{-33}	2^{-33}
$(0.90, 600.0)$	2^{-34}	2^{-34}
$(0.50, 20.0)$	2^{-34}	2^{-34}
$(0.50, 40.0)$	2^{-33}	2^{-33}
$(0.50, 80.0)$	2^{-32}	2^{-32}
$(0.50, 150.0)$	2^{-32}	2^{-32}
$(0.50, 300.0)$	2^{-33}	2^{-33}
$(0.50, 600.0)$	2^{-33}	2^{-33}

Table E.1: Experimental probabilities, using SIMULATED ANNEALING, for different starting temperatures T_0 and different σ parameters. The results are found running SIMULATED ANNEALING for 10 minutes and 60.000 iterations in each heuristic run.

APPENDIX F

CODE FOR DISTINGUISHER ATTACK ON TWEAKED PRINCE

```
// set a randomly chosen tweak key
for (int i = 0; i < 3; ++i) {
tweak_Key[i] = (random_u64() & 0x7FFFFF);
}

unsigned long long alpha, x, k0, k1, a0, a1, y0, y1, m0, m1, c0, c1;

// set random PRINCE master key
k0 = random_u64();
k1 = random_u64();

// initialize values
alpha = random_u64() & 0x7FFFFF;
x = random_u64();
a0 = random_u64() & 0x7FFFFF;
a1 = a0 ^ alpha;

y0 = encrypt(x, a0, k0, k1);
y1 = encrypt(x, a1, k0, k1);

m0 = decrypt(y0, a0, k0, k1);
m1 = decrypt(y1, a0, k0, k1);

c0 = encrypt(m0, a0, k0, k1);
c1 = encrypt(m1, a1, k0, k1);

printf("%016llx\n", (c0 ^ c1));
```

Listing F.1: C++ implementation of distinguisher attack.

BIBLIOGRAPHY

[1] Gregory R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley, 2000.

[2] Abbas Ghaemi Bafghi and Babak Sadeghiyan. Finding suitable differential characteristics for block ciphers with Ant colony technique. In *Proceedings of the Ninth International Symposium on Computers and Communications 2004 Volume 2 (ISCC"04) - Volume 02*, ISCC '04, pages 418–423, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7803-8623-X.

[3] Abbas Ghaemi Bafghi, Reza Safabakhsh, and Babak Sadeghiyan. Finding the differential characteristics of block ciphers with neural networks. *Information Sciences*, 178(15):3118 – 3132, 2008.

[4] Eli Biham and Adi Shamir. *Differential Cryptanalysis of the Data Encryption Standard*. Springer-Verlag, London, UK, UK, 1993. ISBN 0-387-97930-1.

[5] A. Bogdanov, L. Knudsen, G. Leander, C. Paar, A. Poschmann, M. Robshaw, Y. Seurin, and C. Vikkelsoe. PRESENT: An Ultra-Lightweight Block Cipher. In Pascal Paillier and Ingrid Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems - CHES 2007*, volume 4727 of *Lecture Notes in Computer Science*, pages 450–466. Springer Berlin / Heidelberg, 2007. ISBN 978-3-540-74734-5.

[6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2 edition, 2001.

[7] J. Daemen and V. Rijmen. AES Proposal: Rijndael. 1999.

[8] Joan Daemen and Vincent Rijmen. The Wide Trail Design Strategy. In Bahram Honary, editor, *Cryptography and Coding*, volume 2260 of *Lecture Notes in Computer Science*, pages 222–238. Springer Berlin / Heidelberg, 2001. ISBN 978-3-540-43026-1.

[9] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science Engineering, IEEE*, 5(1):46 –55, jan-mar 1998.

[10] Christophe De Cannière, Orr Dunkelman, and Miroslav Knežević. KATAN and KTANTAN – A Family of Small and Efficient Hardware-Oriented Block Ciphers. In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware*

and Embedded Systems - CHES 2009, volume 5747 of *Lecture Notes in Computer Science*, pages 272–288. Springer Berlin / Heidelberg, 2009. ISBN 978-3-642-04137-2.

[11] DTU Crypto Group. PRINCE – A Low-latency Block Cipher for Pervasive Computing Applications. 2012.

[12] T. Eisenbarth and S. Kumar. A Survey of Lightweight-Cryptography Implementations. *Design Test of Computers, IEEE*, 24(6):522 –533, 2007.

[13] John L. Gustafson. Reevaluating Amdahl's law. *Commun. ACM*, 31(5):532–533, May 1988.

[14] Richard A. Johnson. *Miller and Freund's Probability and Statistics for Engineers*. Prentice Hall, New Jersey, 1994. ISBN 0-13-721408-1.

[15] John Kelsey, Bruce Schneier, and David Wagner. Mod *n* Cryptanalysis, with Applications against RC5P and M6. In Lars Knudsen, editor, *Fast Software Encryption*, volume 1636 of *Lecture Notes in Computer Science*, pages 139–155. Springer Berlin / Heidelberg, 1999. ISBN 978-3-540-66226-6.

[16] Lars R. Knudsen and Matthew Robshaw. *The Block Cipher Companion*. Information security and cryptography. Springer, 2011. ISBN 978-3-642-17341-7. I-XIV, 1-267 pp.

[17] Xuejia Lai. Higher Order Derivatives and Differential Cryptanalysis. 1994.

[18] G. Leander and A. Poschmann. On the classification of 4 bit s-boxes. In Claude Carlet and Berk Sunar, editors, *Arithmetic of Finite Fields*, volume 4547 of *Lecture Notes in Computer Science*, pages 159–176. Springer Berlin / Heidelberg, 2007. ISBN 978-3-540-73073-6.

[19] Gregor Leander, Christof Paar, Axel Poschmann, and Kai Schramm. New Lightweight DES Variants. In Alex Biryukov, editor, *Fast Software Encryption*, volume 4593 of *Lecture Notes in Computer Science*, pages 196–210. Springer Berlin / Heidelberg, 2007. ISBN 978-3-540-74617-1.

[20] Moses Liskov, Ronald Rivest, and David Wagner. Tweakable Block Ciphers. *Journal of Cryptology*, 24:588–613, 2011. 10.1007/s00145-010-9073-y.

[21] Mitsuru Matsui. Linear Cryptanalysis Method for DES Cipher. In Tor Helleseth, editor, *Advances in Cryptology – EUROCRYPT '93*, volume 765 of *Lecture Notes in Computer Science*, pages 386–397. Springer Berlin / Heidelberg, 1994. ISBN 978-3-540-57600-6.

[22] Mitsuru Matsui. On correlation between the order of S-boxes and the strength of DES. In Alfredo De Santis, editor, *Advances in Cryptology ? EUROCRYPT'94*, volume 950 of *Lecture Notes in Computer Science*, pages 366–375. Springer Berlin / Heidelberg, 1995. ISBN 978-3-540-60176-0. 10.1007/BFb0053451.

[23] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1996. ISBN 0849385237.

[24] Bruce Schneier. Description of a new variable-length key, 64-bit block cipher (Blowfish). In Ross Anderson, editor, *Fast Software Encryption*, volume 809 of *Lecture Notes in Computer Science*, pages 191–204. Springer Berlin / Heidelberg, 1994. ISBN 978-3-540-58108-6.

[25] Ali Selçuk. On Probability of Success in Linear and Differential Cryptanalysis. *Journal of Cryptology*, 21:131–147, 2008.

[26] Claude E. Shannon. Communication Theory of Secrecy Systems. 1949.

[27] Taizo Shirai, Kyoji Shibutani, Toru Akishita, Shiho Moriai, and Tetsu Iwata. The 128-Bit Blockcipher CLEFIA (Extended Abstract). In Alex Biryukov, editor, *Fast Software Encryption*, volume 4593 of *Lecture Notes in Computer Science*, pages 181–195. Springer Berlin / Heidelberg, 2007. ISBN 978-3-540-74617-1.

[28] Arthur Sorkin. Lucifer, A Cryptographic Algorithm. *Cryptologia*, 8(1):22–42, 1984.

[29] Serge Vaudenay. On the security of CS-Cipher. In *Fast Software Encryption (FSE?99), LNCS 1636*, pages 260–274. Springer-Verlag, 1999.