Doctoral dissertation

On Some Symmetric Lightweight Cryptographic Designs

Martin Ågren

Department of Electrical and Information Technology,
Faculty of Engineering, LTH, Lund University, 2012.

On Some Symmetric Lightweight Cryptographic Designs

Martin Ågren

LUND UNIVERSITY

Doctoral Dissertation
Information Theory
Lund, November 2012

Martin Ågren
Department of Electrical and Information Technology
Lund University
P.O. Box 118, 221 00 Lund, Sweden

Series of licentiate and doctoral dissertations
ISSN 1654-790X; No. 46
ISBN 978-91-7473-391-4

Typeset in Palatino and Helvetica using LaTeX 2_ε.
Printed in Sweden by Tryckeriet i E-huset, Lund University, Lund.

So it has come to this.

Abstract

This dissertation presents cryptanalysis of several symmetric lightweight primitives, both stream ciphers and block ciphers. Further, some aspects of authentication in combination with a keystream generator is investigated, and a new member of the Grain family of stream ciphers, Grain-128a, with built-in support for authentication is presented.

The first contribution is an investigation of how authentication can be provided at a low additional cost, assuming a synchronous stream cipher is already implemented and used for encryption.

These findings are then used when presenting the latest addition to the Grain family of stream ciphers, Grain-128a. It uses a 128-bit key and a 96-bit initialization vector to generate keystream, and to possibly also authenticate the plaintext.

Next, the stream cipher BEAN, superficially similar to Grain, but notably using a weak output function and two feedback with carry shift registers (FCSRs) rather than linear and (non-FCSR) nonlinear feedback shift registers, is cryptanalyzed. An efficient distinguisher and a state-recovery attack is given. It is shown how knowledge of the state can be used to recover the key in a straightforward way.

The remainder of this dissertation then focuses on block ciphers. First, a related-key attack on KTANTAN is presented. The attack notably uses only a few related keys, runs in less than half a minute on a current computer, and directly contradicts the designers' claims. It is discussed why this is, and what can be learned from this.

Next, PRINTcipher is subjected to linear cryptanalysis. Several weak key classes are identified and it is shown how several observations of the same statistical property can be made for each plaintext–ciphertext pair.

Finally, the invariant subspace property, first observed for certain key classes in PRINTcipher, is investigated. In particular, its connection to large linear biases is studied through an eigenvector which arises inside the cipher and leads to trail clustering in the linear hull which, under reasonable assumptions, causes a significant number of large linear biases. Simulations on several versions of PRINTcipher are compared to the theoretical findings.

Contents

Preface

This dissertation contains a general introduction into the field of symmetric cryptography and summarizes some of the research performed by the author at the Department of Electrical and Information Technology at Lund University. It builds on several publications, listed below together with a brief description of the work related to each publication. The first two publications listed below are largely similar and correspond to Chapter 5, while the subsequent publications correspond to, in sequence, one of Chapters 6–10.

- M. Ågren, M. Hell, and T. Johansson, »On hardware-oriented message authentication,« *IET Information Security*, to appear.

  The author of this dissertation performed most of the analysis and wrote most of the paper.

- M. Ågren, M. Hell, and T. Johansson, »On hardware-oriented message authentication with applications towards RFID,« in *Proceedings of the 2011 Workshop on Lightweight Security & Privacy: Devices, Protocols, and Applications, LightSec 2011*, pp. 26–33.

  The author of this dissertation performed most of the analysis and wrote most of the paper.

- M. Ågren, M. Hell, T. Johansson, and W. Meier, »Grain-128a: A new version of Grain-128 with optional authentication,« *International Journal of Wireless and Mobile Computing*, vol. 5, no. 1, pp. 48–59, 2011.

  The author of this dissertation performed most of the work and wrote most of the paper.

- M. Ågren and M. Hell, »Cryptanalysis of the stream cipher BEAN,« in *Proceedings of the 4th International Conference on Security of Information and Networks, SIN 2011*, 2011, pp. 21–28.

  The author of this dissertation performed most of the analysis and wrote most of the paper.

- M. Ågren, »Some instant- and practical-time related-key attacks on KTANTAN32/48/64,« in *Selected Areas in Cryptography—SAC 2011*, ser. Lecture Notes in Computer Science, A. Miri and S. Vaudenay, Eds., vol. 7118. Springer-Verlag, 2012, pp. 213–229.

  The author of this dissertation is the sole author of the paper.

- M. Ågren and T. Johansson, »Linear cryptanalysis of PRINTcipher — trails and samples everywhere,« in *Progress in Cryptology—INDOCRYPT 2011*, ser. Lecture Notes in Computer Science, D. J. Bernstein and S. Chatterjee, Eds., vol. 7107. Springer-Verlag, 2011, pp. 114–133.

  The author of this dissertation performed most of the analysis and wrote most of the paper.

- M. A. Abdelraheem, M. Ågren, P. Beelen, and G. Leander, »On the distribution of linear biases: Three instructive examples,« in *Advances in Cryptology—CRYPTO 2012*, ser. Lecture Notes in Computer Science, R. Safavi-Naini and R. Canetti, Eds., vol. 7417. Springer-Verlag, 2012, pp. 50–67.

  The author of this dissertation performed most of the analysis regarding the PRINTcipher example and wrote most of Section 5.

Research presented in the following publications is not included in this dissertation, although coauthored by its author.

- M. Ågren, T. Johansson, and M. Hell, »Improving the rainbow attack by reusing colours,« in *Cryptology and Network Security—CANS 2009*, ser. Lecture Notes in Computer Science J. A. Garay, A. Miyaji, and A. Otsuka, Eds., vol. 5888, Springer-Verlag, 2009, pp. 362–378.
- M. Ågren, M. Hell, T. Johansson, and C. Löndahl, »Improved message passing techniques in fast correlation attacks on stream ciphers,« in *7th International Symposium on Turbo Codes & Iterative Information Processing*, to appear.

The research work included in this dissertation was supported in part by the Swedish Foundation for Strategic Research (SSF) through its Strategic Center for High Speed Wireless Communication at Lund.

Acknowledgments

This dissertation would not have come about without the guidance of my supervisor, Thomas Johansson, who has an immense knowledge, many good ideas and the ability to delicately notify me of when my own ideas are not so good. My assistant supervisor, Martin Hell, has been particularly good at coming up with simple answers to my half-baked and overly-complicated questions. My office neighbor and good friend Paul Stankovski also deserves many thanks for a cheerful attitude at all times and for kind help in reviewing draft manuscripts.

Florian Hug has made my life easier by providing the LaTeX template used for typesetting this dissertation, but also better by being such a good friend. All other people at the department, including the Wednesday fika group and the Friday lunch gang, have certainly contributed to the nice environment in which I have spent the last four years. I also want to thank the DTU people for being so friendly to me while I visited them, and in particular Gregor Leander who somehow kept my research on the right track.

The work presented in this dissertation would have been nowhere near even being begun without my parents, Evy and Kenneth, who gave me what I—with bias, but still—consider the best upbringing possible, and who have played a large part in the fact that I have been so lucky so far. They always make me feel welcome in their home and were kind enough to let me setup a writer's cabin where the introductory part of this dissertation was written. That environment is also intimately connected to my brother Mikael, my grandmother Sonja, and my uncles Arne and Göran, who deserve much credit for always being around.

Of all the things that have happened to me in Lund/Malmö since I moved here nine years ago, what really stands out as important is getting to know my

best friends Julia and Jonas. At the time of writing this, it is not decided where I will go next, or if I will stay here, but whatever happens I am determined to keep my best friends close.

Many more—most of them unknowingly—have provided a piece of the puzzle that has been crucial in leading to this particular outcome of an immense collection of random events. It is my hope that the work presented here can somehow be beneficial to others, much like the work of others has been so valuable to me.

The Author
Kruseböke, October 2012

1 Introduction

Cryptology is a both old and young field of research. It seems plausible that only a short time after the art of writing was invented, so was that of *secret* writing. Historically, and perhaps still to the layman, cryptology is commonly connected with warfare, espionage and the like. However, with the advent of the information society and the digital revolution, cryptology is more and more relevant also in the peaceful and friendly lives of »ordinary« people.

The academic research field has grown considerably since the seminal 1976 paper by Diffie and Hellman [DH76], who simplified the issue of key management that was hampering the use of encryption between large numbers of users (e.g., all the web servers and web clients of today), and provided a starting point for the academic cryptologic research community. When what is today the National Institute of Standards and Technology (NIST) wanted to develop the Digital Encryption Standard (DES) in the seventies, the process was closed in the sense that the exact rationale behind the final design was not publicly known. During the process of selecting the next-generation Advanced Encryption Standard (AES), however, the scientific community was invited to contribute to a larger extent. Several international researchers submitted their designs and eventually Rijndael by Belgian researchers Daemen and Rijmen was selected to be used for the AES [DR02].

The field of cryptology is commonly divided into two subfields that are not completely disjoint. *Cryptography* is related to the design of cryptographic algorithms and protocols. *Cryptanalysis*, on the other hand, is everything related to breaking these constructions, or trying to understand how, why and precisely under what circumstances they are secure. A common approach in cryptology is *Kerckhoffs' principle*, which states that a cryptosystem should be secure even if everything about the system, except the key, is public knowledge. That is, the algorithm itself should not have to be secret, only its input.

A trend in our society is that not only a few, large machines are in need of cryptographic solutions, but that a larger number of increasingly smaller devices have a need for such technology. Mobile phones and banking devices are standard everyday examples, but even more common, and smaller, are the RFID tags that replace more and more bar codes for tracking shipping pallets and even individual consumer items. The general idea is that while bar codes must be read optically, i.e., using a reading device that is aimed more or less directly at a specific area on the (packaged) goods, RFID chips employ radio waves, meaning they can register goods (chips) that pass by the reader much more »approximately.« This assures faster processing, but also invites completely new attack scenarios, especially when the RFID chips are used in more creative ways, e.g., for bus passes or building access control.

A crucial point to make is that these products are rarely directly targeting a cryptographic need. More likely, they allow people, businesses and goods to communicate easier, faster or more frequent. It is therefore not obvious that the producer will prioritize cryptographic solutions. In other words, the cryptographic part only has a limited budget in terms of, e.g., development cost, unit production cost, and battery power consumption. For this reason, the standard, general-purpose solutions, e.g., AES, are often considered out of the option when developing such technology. Lately, the academic community have produced several interesting new algorithms that target this niche, often denoted *lightweight cryptography*.

1.1 NO KEYS, PUBLIC KEYS, AND SYMMETRIC KEYS

Several different classes of cryptographic *primitives* can be identified. They are useful in different scenarios and can be combined into larger systems and solutions.

A *(cryptographic) hash function* maps variable-length *preimages* to fixed-length *hash values*. A hash function takes no key as input, but notably it should be computationally hard to find a preimage with any given hash, to change any preimage without changing the hash, and to find two distinct preimages that yield the same hash value. Still, it should be computationally easy to derive the hash value for any given preimage. Very recently, NIST completed a process similar to the AES process to select the next hash standard, SHA-3. The winner is Keccak by Bertoni et al. Hash functions are useful building blocks in several cryptographic settings, but are not considered in this dissertation.

Above, the *public-key* concept that simplified key management was briefly mentioned. Intuitively, perhaps, the key should be kept secret: the sender and the receiver both share a secret key, which they have somehow agreed on. This setting is often referred to as *symmetric* cryptography.

An obvious problem of symmetric keys is that they must somehow be distributed: it might not be feasible to preshare keys with everyone you might one day be interested in communicating with. While this can be avoided by trusting a third party to do the key communication if both parties already share keys with the third party, it does introduce the third party and the issue of trust.

In public-key cryptography, two slightly different keys are used. To encrypt a plaintext, a public key is enough, but to then decrypt the ciphertext, a corresponding private key is required. Public-key constructions are typically based on some mathematical problem, such as factoring, which is assumed to be a hard problem in a computational sense. To consider the example of factoring, the private key can consist of two large prime numbers and the corresponding public key of their product. Acquiring the private key from the public is possible in theory, but all known algorithms for factoring will require huge resources (e.g., time) to complete.

It could be suspected that once public-key cryptography arrived on the international scene, all symmetric algorithms would be moot. This is not so. Public-key cryptography is inherently slow and keys are large. Therefore, it is common that a one-time, random key is used with a symmetric algorithm to encrypt the plaintext. The one-time key, which is comparably small, is then encrypted using the public-key algorithm. Thus, public-key cryptography augments symmetric cryptography rather than replaces it.

There are traditionally two classes of symmetric encryption algorithms: block ciphers and stream ciphers. Block ciphers encrypt one block of plaintext at the time, typically 32–256 bits. During decryption, a similar block-by-block approach is used.

Stream ciphers, on the other hand, look at the plaintext as a stream of bits. Typically, a stream cipher produces a pseudorandom stream of bits which it uses to mask the plaintext. During the encryption, the same pseudorandom sequence is produced and »unmasked« from the ciphertext.

In cryptanalysis, several different attack settings are common, ranging from ciphertext-only attacks through chosen-plaintext attacks to related-key attacks. Broadly speaking, with each relaxation of the attack setting, the attack becomes more powerful (e.g., fast) and less practical. Several standard techniques have been developed, such as linear distinguishers on stream ciphers and differential or linear attacks on block ciphers.

Many cryptographic primitives have been proposed over the years, and new designs are published as cryptanalysis enhances the understanding of how to design safer and more efficient primitives. New European Schemes for Signatures, Integrity and Encryption (NESSIE), a European project which aimed to recommend several algorithms, such as block ciphers and public-key encryption schemes, notably did not select any stream ciphers due to all submitted algorithms being successfully cryptanalyzed. This spurred the eSTREAM project, which decided on two portfolios of promising stream ciphers. One of these aimed at hardware attractive constructions, and consists of Grain, Trivium, and MICKEY. A fourth cipher, FCSR-H, was included in the initial portfolio, but removed due to subsequent cryptanalysis.

Another area of interest to cryptographers, besides secrecy, is authentication: verifying that a (possibly encrypted) message has not been altered and that it was indeed sent by the supposed sender.

1.2 DISSERTATION OUTLINE

This dissertation will focus on symmetric lightweight cryptography. Several cryptanalytic results and some new designs will be presented. The remainder of this dissertation is organized as follows.

In Chapter 2 the notation used in this dissertation will be introduced. Various security notations will be discussed, and some common security goals will be presented. Then, several classes of cryptographic primitives will be introduced, followed by various building blocks commonly used and some more or less specific designs, such as the nonlinear combiner and the Grain family of stream ciphers.

In Chapter 3, several useful tools will be introduced, such as various transforms and some matrix theory.

Several of those tools will then be used in Chapter 4, which covers attack settings and describes various standard attack strategies.

Chapter 5 and onwards constitute the contributions of this dissertation, and can be broadly described as focusing on authentication, stream ciphers, and block ciphers, in that order.

Chapter 5 describes a particular way of authenticating using a class of structured matrices. A previous security result in this area is revisited and tweaked. Some insights are used to argue why the security proof can be expected to be far from tight. A new class of authenticating mechanism, which might be suitable for lightweight applications, is introduced and exemplified.

Chapter 6 describes a new member of the Grain family of stream ciphers, Grain-128a. It is a new version of the stream cipher Grain-128, which has been affected by recent cryptanalysis. Several design choices of Grain-128 have been revisited. The new cipher is also equipped with an authenticating building block, the use of which is optional meaning that devices without any need for authentication can save on gates by not implementing the authentication.

Chapter 7 presents cryptanalysis of the stream cipher BEAN, which has been suggested as a lightweight stream cipher. It is shown how to distinguish BEAN keystreams from truly random sequences, and how to speed up a brute force search for the key. It is also shown how the state, and from this the key, can be recovered using a time–data trade-off; this is exemplified by a trade-off which uses short keystream sequences but impractical time.

The dissertation then focuses exclusively on block ciphers. Chapter 8 describes a related-key attack on KTANTAN, where it is shown how the key can be recovered using practical time and memory constraints, but an arguably unrealistic attack setting. The findings are compared to the designers' claims and it is suggested how similar problems can be avoided in future designs.

Chapter 9 presents an investigation of the behavior of PRINTcipher with respect to linear cryptanalysis. In particular, it is shown how a single plaintext–ciphertext pair allows the attacker to make several statistical observations and increase the attack success probability.

Chapter 10 again relates to PRINTcipher, which is known to have invariant subspaces and behave badly in terms of linear cryptanalysis. The relation between invariant subspaces and linear biases, in any block cipher, is investigated from the particular viewpoint of eigenvectors arising in the block cipher. The analysis is augmented by some experimental results on PRINTcipher and some smaller-state versions of it. Further, one of the main results, the existence of a particular eigenvector, is investigated in detail for PRINTcipher, to explain at a lower level why it arises.

2 Symmetric Cryptography

This chapter begins with an introduction to the general notation used in this dissertation. It is discussed what exactly constitutes a cryptanalytic attack, or how one might define »security.« After this, various cryptographic primitives are introduced, such as block and stream ciphers, Sboxes and binary registers, and the filter generator.

2.1 NOTATION

For a nonnegative integer j, the integers j div $2 \geq 0$ and $j \bmod 2 \in \{0, 1\}$ are uniquely defined such that $2 \cdot (j \text{ div } 2) + (j \bmod 2) = j$.

Vectors will be written as $\boldsymbol{v}$. In particular, $\boldsymbol{p}$, $\boldsymbol{c}$, and $\boldsymbol{k}$ will be used to denote plaintext, ciphertext, and key, respectively. The length of a vector, i.e., the number of elements in it, will be denoted $|\boldsymbol{v}|$. In particular, it will be assumed that $|\boldsymbol{c}| = |\boldsymbol{p}|$.[1] For any finite set S, $|S|$ is the number of elements in S.

Vectors will be distinguished using superscripts, e.g., $\boldsymbol{v}^0$ and $\boldsymbol{v}^1$. The individual elements of a vector will be indexed using subscripts from 0, as in $\boldsymbol{v} = (v_0, v_1, \ldots, v_{n-1})$. Concatenation of two vectors, not necessarily of equal length, is denoted by $||$, e.g., the concatenation of $\boldsymbol{v}^0 = (v_0^0, v_1^0)$ and $\boldsymbol{v}^1 = (v_0^1, v_1^1)$ is $\boldsymbol{v}^0 || \boldsymbol{v}^1 = (v_0^0, v_1^0, v_0^1, v_1^1)$. The concatenation $\boldsymbol{v}||(a)$, where a is a scalar, can be written as $\boldsymbol{v}||a$; similarly $a||\boldsymbol{v} = (a)||\boldsymbol{v}$.

$\mathbb{F}_2^n$ denotes the set of n-bit vectors. For two elements $\boldsymbol{v}^0, \boldsymbol{v}^1 \in \mathbb{F}_2^n$, addition is defined elementwise modulo 2, i.e., $\boldsymbol{v}^+ = \boldsymbol{v}^0 + \boldsymbol{v}^1 \in \mathbb{F}_2^n$ is defined through

[1] Especially in the context of block ciphers (cf. Section 2.5), the ciphertext might not have the exact same bitlength as the *message*, but in all aspects of encryption, this dissertation ignores the concept of *padding* altogether.

$v_i^+ = v_i^0 + v_i^1 \bmod 2$, $0 \leq i < n$. Further, $\langle \boldsymbol{v}^0, \boldsymbol{v}^1 \rangle$ denotes the canonical inner product of two binary vectors, i.e.,

$$\langle \boldsymbol{v}^0, \boldsymbol{v}^1 \rangle = \sum_{i=0}^{n-1} v_i^0 v_i^1 \bmod 2 = \left| \left\{ i : \ v_i^0 = v_i^1 = 1 \right\} \right| \bmod 2.$$

For $\boldsymbol{v} = (v_0, v_1, \ldots, v_{n-1}) \in \mathbb{F}_2^n$, define $\boldsymbol{v} \ll 1 = (v_1, v_2, \ldots, v_{n-1}, 0)$. The vector $\boldsymbol{v} = (v_0, v_1, \ldots, v_{n-1}) \in \mathbb{F}_2^n$ may be interpreted as an integer

$$(\boldsymbol{v}) = \sum_{i=0}^{n-1} v_i 2^i,$$

e.g., the vector $\boldsymbol{v} = (1, 0, 1, 0)$ (with v_0 to the left) corresponds to the integer $(\boldsymbol{v}) = 5$. If $\boldsymbol{v}$ is of length divisible by four, it can be written in hexadecimal where each group of four bits is represented by a hexadecimal digit `0`–`f`, where in hexadecimal, the leftmost bit is the most significant bit, e.g., `1d` represents $(0, 0, 0, 1, 1, 1, 1, 0, 1)$. The function $\oplus_{\boldsymbol{v}}\colon \mathbb{F}_2^n \to \mathbb{F}_2^n$ is defined for $\boldsymbol{v} \in \mathbb{F}_2^n$ as $\oplus_{\boldsymbol{v}}(\boldsymbol{x}) = \boldsymbol{x} + \boldsymbol{v}$. The n-bit vector consisting of only zeros is denoted $\mathbf{0}^n$.

For a subspace $U \subset \mathbb{F}_2^n$, i.e., $\boldsymbol{v}^0 + \boldsymbol{v}^1 \in U$ for all $\boldsymbol{v}^0, \boldsymbol{v}^1 \in U$, denote by $U^\perp$ the orthogonal subspace, $U^\perp = \{\boldsymbol{x} \in \mathbb{F}_2^n : \ \langle \boldsymbol{x}, \boldsymbol{v} \rangle = 0, \forall \boldsymbol{v} \in U\}$. It holds that $|U| \cdot |U^\perp| = 2^n$. For $U \subset \mathbb{F}_2^n$ and $\boldsymbol{x} \in \mathbb{F}_2^n$, write $U + \boldsymbol{x} = \{\boldsymbol{u} + \boldsymbol{x} \mid \boldsymbol{u} \in U\}$. For a function $F\colon \mathbb{F}_2^n \to \mathbb{F}_2^n$ and a set $V \subseteq \mathbb{F}_2^n$, write $F(V) = \{F(\boldsymbol{x}) \mid \boldsymbol{x} \in V\}$.

The *Hamming weight* of a binary vector $\boldsymbol{v}$ is the number of ones in the vector and is denoted $w_{\mathrm{H}}(\boldsymbol{v})$. The *Hamming distance* between two vectors, $\boldsymbol{v}^0$ and $\boldsymbol{v}^1$, is denoted $d_{\mathrm{H}}(\boldsymbol{v}^0, \boldsymbol{v}^1)$ and is the number of positions where $\boldsymbol{v}^0$ and $\boldsymbol{v}^1$ differ. It follows that $d_{\mathrm{H}}(\boldsymbol{v}^0, \boldsymbol{v}^1) = w_{\mathrm{H}}(\boldsymbol{v}^0 + \boldsymbol{v}^1)$.

For a nonnegative real number x, $\lceil x \rceil$ denotes the ceiling function, i.e., $\lceil x \rceil$ is the smallest integer $\lceil x \rceil \geq x$. Similarly, $\lfloor x \rfloor$ is the largest integer $\lfloor x \rfloor \leq x$.

$\mathbf{Pr}_S[\boldsymbol{s} = \boldsymbol{s}']$ denotes the probability that $\boldsymbol{s} = \boldsymbol{s}'$, where $\boldsymbol{s} \in S$ and S is a finite set of vectors. It holds that $\sum_S \mathbf{Pr}_S[\boldsymbol{s}] = 1$. Almost always, $S = \mathbb{F}_2^n$ so this will not be given explicitly. If no other distribution is specified, the uniform distribution will be assumed, i.e., $\mathbf{Pr}_S[\boldsymbol{s} = \boldsymbol{s}'] = \frac{1}{|S|}, \forall \boldsymbol{s}' \in S$.

2.2 WHAT CONSTITUTES AN ATTACK

Cryptanalysis, as already pointed out, relates to analyzing cryptographic primitives. A large part of cryptanalysis is that of looking at cryptographic primitives, or classes of cryptographic primitives, and trying to *attack* or *break* them. It is not clear exactly how to define an attack (break), but one attempt might look like the following.

»Any algorithm which uses some data originating from a cryptographic primitive or its implementation to learn something supposedly secret can be called an attack.«

Here, »something supposedly secret« is a rather vague term, which can refer to information about a key, plaintext, preimage, or state, e.g., the whole key, some bits of the key, or some information about which keys are more or less probable than others, or even impossible.

Similarly, »some data originating from ...« could be a ciphertext or a hash value, but also side-channel information such as power consumption (possibly as a function of time), the time required to perform one particular operation (e.g., the encryption of some plaintext), or similar data related to an (execution of an) implementation of the primitive.

Notably, this definition does not mention how long time the attack is allowed to run, or how much memory it is allowed to use. In the next section, three different approaches to this will be given: one which allows the algorithm infinite resources, and two which limits them.

2.3 SECURITY NOTIONS

At its core, cryptography is about constructing a primitive which is simple to understand, easy to implement, and efficient, but still secure. Similar to above, it is not obvious how to define what it means for a primitive to be »secure.« At least three viewpoints can be offered:

2.3.1 UNCONDITIONAL SECURITY

This viewpoint comes from the line of information theory and the work by Shannon in the 1940s and 1950s. A system is said to be *unconditionally secure* if an adversary cannot break it, even with access to infinite computational resources. It is important to note that whether or not a cryptosystem is unconditionally secure might depend on the attack setting. Shannon considered *perfect secrecy*, which can be defined as the property that $\mathbf{Pr}\left[\boldsymbol{p} = \boldsymbol{p}' \mid \boldsymbol{c} = \boldsymbol{c}'\right] = \mathbf{Pr}\left[\boldsymbol{p} = \boldsymbol{p}'\right]$ for all plaintexts $\boldsymbol{p}'$ and ciphertexts $\boldsymbol{c}'$, i.e., the ciphertext leaks no information about the plaintext. Shannon essentially showed that perfect secrecy can only be achieved if the number of keys is equal to or greater than the number of plaintexts, i.e., if the keys and plaintexts are bitstrings, the keys must be at least as long as the plaintexts, which tends to make such schemes impractical.

2.3.2 PROVABLE SECURITY

A primitive is said to be provably secure if breaking it, when used in some specific scenario, implies the ability to break some mathematical problem which is normally assumed to be »hard,« e.g., factoring a large number into prime factors.

2.3.3 EMPIRICAL SECURITY

With symmetric primitives, it is often more reasonable to try to assess what an attacker is able to do in practical terms. It is clear that if an attacker's aim is to find the $|\boldsymbol{k}|$-bit key $\boldsymbol{k}$, they can make $2^{|\boldsymbol{k}|}$ guesses for the key. If it is possible to somehow validate guesses, e.g., because some plaintexts and ciphertexts are available, an *exhaustive search* (brute force) can be applied, searching through all elements in the finite search space until the correct element (e.g., key) is found. If all keys are equiprobable, on average $2^{|\boldsymbol{k}|-1}$ must be tried.

A system *provides n bits of security* if the most efficient attack on it requires computational effort comparable to an exhaustive search over n bits.

It is rarely known how to *prove* a particular primitive computationally secure. Instead, it is argued that if the algorithm has »attracted attention but not been broken,« i.e., several (skilled) researchers have spent a lot of time on it and perhaps gained several insights but, crucially, not found any flaws that allow better attacks than brute force, then there is »confidence« in the algorithm. Typically, however, it can not be ruled out that future bright ideas will provide a break. Thus, attacks tend to get better and better, and the primitives weaker and weaker. Cryptographers therefore often embed a *security margin* into their ciphers, i.e., they show (or argue) that even a »reduced« version of the algorithm is secure against known attack approaches.

In the remainder of this chapter, some general classes of cryptographic primitives will be outlined, followed by various common building blocks and some design strategies for combining the building blocks into primitives. More on attacks, e.g., settings, requirements, and practicality, can be found in Chapter 4.

2.4 STREAM CIPHERS

At least two general classes of stream ciphers can be identified: self-synchronizing and synchronous; this dissertation will only treat the synchronous stream ciphers.

The idea behind a stream cipher is that the comparably small random key is expanded into a *keystream*, which is used to mask the plaintext. That is, $\boldsymbol{c} = \boldsymbol{p} + \boldsymbol{z}$, where $\boldsymbol{z}$ is the keystream.

If $\boldsymbol{z}$ was a uniformly random sequence, used only once, this scheme would provide perfect secrecy. That scheme, called the one-time pad (OTP), is in most cases completely out of the question as it requires that a key of the same size as the plaintext has been agreed on beforehand. The one-time pad has however been used in espionage, diplomatic traffic, and together with quantum key distribution.

The intuition behind a stream cipher, then, is to use a *keystream generator*

ϕ: $\mathbb{F}_2^{|\boldsymbol{k}|} \to \mathbb{F}_2^{|\boldsymbol{z}|}$ to produce a keystream, which is in some sense close to random, or random-looking. However, it won't *be* random, and the keystream-generator will only be able to produce $2^{|\boldsymbol{k}|}$ distinct keystreams where the length of the keystream is typically $|\boldsymbol{z}| \gg |\boldsymbol{k}|$.

If a key is reused with a stream cipher as above, the exact same keystream will be produced. This is not ideal, since an outside observer would be able to learn, e.g., the parity $\boldsymbol{c}^1 + \boldsymbol{c}^2 = \boldsymbol{p}^1 + \boldsymbol{z} + \boldsymbol{p}^2 + \boldsymbol{z} = \boldsymbol{p}^1 + \boldsymbol{p}^2$ of two different plaintexts. Because of this, most modern stream ciphers use a so-called initialization vector (IV). The key $\boldsymbol{k}$ is kept secret and constant, and a public IV $\boldsymbol{v}$ is used to produce distinct keystreams through the keystream generator, which is now written as ϕ: $\mathbb{F}_2^{|\boldsymbol{k}|} \times \mathbb{F}_2^{|\boldsymbol{v}|} \to \mathbb{F}_2^{|\boldsymbol{z}|}$.

IVs can be used sequentially, or agreed upon through some protocol, but the basic idea is that the security of the cipher does not depend on the secrecy of the IV. It can be communicated in the open, or selected according to some public rule—as long as the key remains secret, the stream cipher should provide secure encryption.

A stream cipher with IV normally requires an *initialization* phase where the key and the IV is loaded into the state, and the state is updated, without producing output, to mix the key and the IV. The goal is that when the keystream generator begins generating keystream, the state should be mixed so much that a change in the IV yields an entirely different keystream.

2.5 BLOCK CIPHERS

As the name suggests, a block cipher F: $\mathbb{F}_2^{|\boldsymbol{p}|} \times \mathbb{F}_2^{|\boldsymbol{k}|} \to \mathbb{F}_2^{|\boldsymbol{p}|}$ acts on a block $\boldsymbol{p}$ of plaintext of a certain size. AES acts on blocks of 128 bits, while several of the lightweight block ciphers use smaller blocksizes such as 64 [BKL$^+$07], 48 [KLPR10], or 32 [DDK09].

For a fixed key, the block cipher must be invertible, i.e., $F_{\boldsymbol{k}}$ defined through $F_{\boldsymbol{k}}(\boldsymbol{p}) = F(\boldsymbol{p}, \boldsymbol{k})$ must be a permutation.

2.5.1 BLOCK CIPHER MODES OF OPERATION

In this section, it is briefly outlined how a block cipher acting on blocks of n bits can be used to encrypt a longer plaintext $\boldsymbol{p}$. For simplicity, assume that $|\boldsymbol{p}| = j \cdot n$. A first attempt might be to partition the plaintext into blocks of size n, i.e., $\boldsymbol{p}^0||\boldsymbol{p}^1||\ldots||\boldsymbol{p}^{j-1} = \boldsymbol{p}$. By encrypting each block in turn, the ciphertext is $\boldsymbol{c} = \boldsymbol{c}^0||\boldsymbol{c}^1||\ldots||\boldsymbol{c}^{j-1}$, where $\boldsymbol{c}^i = F_{\boldsymbol{k}}(\boldsymbol{p}^i)$. This is referred to as electronic codebook (ECB) mode and has at least one fundamental flaw: two ciphertext blocks will be the same if and only if the corresponding plaintext blocks are the same, meaning there is information leakage.

A *mode of operation* which avoids this issue is cipher block chaining (CBC), where $\boldsymbol{c}^i = F_{\boldsymbol{k}}(\boldsymbol{p}^i + \boldsymbol{c}^{i-1})$ and $\boldsymbol{c}^{-1} = \boldsymbol{v}$ is an IV.

A block cipher can also be used to construct a keystream generator, through what is referred to as counter mode (CTR): the keystream $\boldsymbol{z} = \boldsymbol{z}^0||\boldsymbol{z}^1||\dots||\boldsymbol{z}^{j-1}$ is derived through $\boldsymbol{z}^i = F_{\boldsymbol{k}}(\boldsymbol{v}||\boldsymbol{b}^i)$, where $(\boldsymbol{b}^i) = i$ and $\boldsymbol{v}$ is an IV.

2.6 AUTHENTICATION

If an attacker changes a few bits in block cipher encrypted data, at least the corresponding block(s) is (are) altered in a completely unpredictable way. The attacker has no way of predicting, on a bit-level, what changes will be made to the plaintext after decryption.

With stream cipher encrypted data, however, the corresponding change in the plaintext is easy to predict: if the ciphertext $\boldsymbol{c}$ corresponds to the plaintext $\boldsymbol{p}$, then applying the difference $\boldsymbol{a}$ to produce $\boldsymbol{c}' = \boldsymbol{c} + \boldsymbol{a}$ yields the plaintext $\boldsymbol{p}' = \boldsymbol{p} + \boldsymbol{a}$ after decryption.

Thus, an attacker who is able to alter the ciphertext and knows (roughly) what the plaintext is, may be able to predictably change it. As an example, if the ciphertext is of length $|\boldsymbol{c}| = 1$, i.e., the plaintext is just a bit 1/0 corresponding to, e.g., »buy«/»sell« or »yes«/»no«, the attacker would be able to change the plaintext after decryption, although they would not know from what and to what.

In a slightly more advanced example, if the plaintext contains an amount of money, the attacker might be able to alter this amount, perhaps without knowing the exact original or new amount. The attack strategy would be to change a few bits at a position in the ciphertext which is assumed to represent the amount of money. If the encoding scheme is known, the original amount is known, and the position in the plaintext is known (e.g., due to boilerplate templates being used), this attack could be very precise.

For this reason, stream ciphers are often coupled with some sort of *authentication*. A message authentication code (MAC) [GMS74] is a family of functions which map messages to tags using keys. The sender produces a tag $\boldsymbol{t} = f(\boldsymbol{k}_{\text{MAC}}, \boldsymbol{m})$ for a message $\boldsymbol{m}$ (plaintext or ciphertext) using a key $\boldsymbol{k}_{\text{MAC}}$ and attaches it to the message. The receiver, who shares the key $\boldsymbol{k}_{\text{MAC}}$, can produce the tag for the received message and immediately decide whether the message can be regarded as authentic.

An active adversary tries to modify a transmitted message and its tag, hoping to get this accepted at the receiver side. The designer wants the probability that the attacker succeeds to be some very small value. Attacks on authentication schemes are covered more in Section 4.12.

There are many ways to provide message authentication, e.g., using certain block cipher modes of operation such as CBC-MAC [Int99], iterating a hash function as in HMAC [BCK], or using constructions based on universal hashing [WC81] [Sti92]. This dissertation focuses on the last approach, which is the usual choice when encryption is done through a stream cipher. Typical examples would be the Galois counter mode (GCM) [MV04] and the 3GPP Integrity Algorithm UIA2 [ETS09] found in third generation mobile phone standards, Universal Mobile Telecommunications System (UMTS) and Long Term Evolution (LTE).

2.6.1 UNIVERSAL HASH FUNCTIONS

Consider a family (set) $H = \{h_i\}$ of hash functions $h_i \colon A \to B$.

There are several ways of constructing MACs from universal hash functions (or equivalently authentication codes [Sim92]).

H is *ε-almost xor universal* (ε-AXU) if $\forall \boldsymbol{x}, \boldsymbol{x}' \in A, \boldsymbol{x} \neq \boldsymbol{x}', \boldsymbol{y} \in B$,

$$\left|\{h \in H : \ h(\boldsymbol{x}) + h(\boldsymbol{x}') = \boldsymbol{y}\}\right| \leq \varepsilon \cdot |H| .$$

When constructing a MAC using an ε-AXU family, one part of the key is used to select a function $h \in H$ and the output of this function is xored with a second part of the key, used as a one-time pad, chosen randomly from B.

2.7 BOOLEAN FUNCTIONS

A function $f \colon \mathbb{F}_2^n \to \mathbb{F}_2$ is called a *Boolean function*. The function f evaluated in the point $\boldsymbol{x} = (x_0, x_1, \ldots, x_{n-1})$ will be written $f(\boldsymbol{x}) = f(x_0, x_1, \ldots, x_{n-1})$. A Boolean function f is called *balanced* if $\mathbf{Pr}\left[f(\boldsymbol{x}) = 0\right] = 1/2$ on uniform input.

With $\boldsymbol{x}, \boldsymbol{v} \in \mathbb{F}_2^n$, let

$$\boldsymbol{x}^{\boldsymbol{v}} = \prod_{0 \leq j < n \colon v_j = 1} x_j,$$

i.e., $\boldsymbol{x}^{\boldsymbol{v}}$ is the product of precisely those variables x_j where $v_j = 1$. A Boolean function can be represented as a polynomial over F_2, called the *algebraic normal form* (ANF),

$$f(\boldsymbol{x}) = \sum_{\boldsymbol{v} \in \mathbb{F}_2^n} a_{\boldsymbol{v}} \boldsymbol{x}^{\boldsymbol{v}},$$

where the coefficients $a_{\boldsymbol{v}} \in \mathbb{F}_2$. The *algebraic degree*,

$$\deg(f) = \max_{\boldsymbol{v} \in \mathbb{F}_2^n \colon a_{\boldsymbol{v}} = 1} (w_{\mathrm{H}}(\boldsymbol{v})),$$

is the number of variables in the highest order term with nonzero coefficient. A Boolean function f is said to be *affine* if $\deg(f) = 1$, and if further $a_{\boldsymbol{0}} = 0$,

f is said to be *linear*. The *nonlinearity* of an n-variable Boolean function is the minimum distance from the set $\mathcal{A}(n)$ of all n-variable affine functions,

$$\mathrm{nl}(f) = \min_{g \in \mathcal{A}(n)} \left| \{ \boldsymbol{x} \in \mathbb{F}_2^n : \ f(\boldsymbol{x}) \neq g(\boldsymbol{x}) \} \right| .$$

The correlation immunity of a Boolean function is a measure of to which degree its output is correlated to a subset of its inputs. A function is said to be *mth order correlation immune* if there is no statistical dependence between $f(\boldsymbol{x})$ and any set of m or fewer variables in $\boldsymbol{x}$.

If an mth order correlation immune function is balanced, it is called m-resilient.

2.8 SBOXES

An $n \times m$ Sbox is formally a function S: $\mathbb{F}_2^n \to \mathbb{F}_2^m$. While this covers both $m = 1$, i.e., Boolean functions, and $n = m = 128$, which could represent an entire block cipher, the term Sbox is used to denote a particular building block commonly used in block ciphers, but also sometimes in stream ciphers.

Typical Sbox sizes are 8×8 as found in AES, to 4×4 as used in, e.g., PRESENT, or even 3×3 as in PRINTcipher (cf. Chapter 9). The S in Sbox is for *substitution*, and Sboxes of size $n \times n$ should be permutations in order to yield uniformly distributed output when the input is uniform. DES uses Sboxes of size 6×4. These Sboxes can clearly not be permutations, but they do maintain equiprobable outputs given uniform input.

By writing the Sbox as S: $\mathbb{F}_2^n \to \mathbb{F}_2 \times \mathbb{F}_2 \times \ldots \times \mathbb{F}_2$, i.e.,

$$\boldsymbol{y} = S(\boldsymbol{x}) = (f_0(\boldsymbol{x}), f_1(\boldsymbol{x}), \ldots, f_{m-1}(\boldsymbol{x})),$$

each bit y_i of the output can be considered as the output of a Boolean function of the input, $y_i = f_i(\boldsymbol{x})$.

Designing cryptographically strong Sboxes is about ensuring that each individual function f_i behaves well, e.g., in terms of nonlinearity, but also that the Sbox behaves well in a larger sense. Sboxes are usually chosen with good resistance against differential and linear cryptanalysis, which will be described in Sections 4.10 and 4.11.

2.9 BINARY REGISTERS

Some of the building blocks described here can be defined over larger characteristics, but only the binary versions are presented in this section.

A binary register is an n-bit state, which is updated somehow in order to create state transitions. Denoting the initial state by $\boldsymbol{s}^0$, the ith state, $\boldsymbol{s}^i$, $i > 0$, is defined as $\boldsymbol{s}^i = f(\boldsymbol{s}^{i-1})$, where $f\colon \mathbb{F}_2^n \to \mathbb{F}_2^n$ is the state update function.

As there is only a finite number of possible states, it is clear that there will be at least one *cycle*. That is, for some $i \geq 0$ and some $t > 0$, $\boldsymbol{s}^i = \boldsymbol{s}^{i+t}$ will occur. The minimal such t is called the *period*, and such a state is said to be *strictly periodic*. Then, for $i' > i$, $\boldsymbol{s}^{i'} = \boldsymbol{s}^{i'+t}$. Note that, depending on the state update function, a state $\boldsymbol{s}^i$ might have (at least) two different preimages $\boldsymbol{s}^{i-1}$ solving $\boldsymbol{s}^i = f(\boldsymbol{s}^{i-1})$, and that once a cycle has been entered, it is repeated over and over again, assuming i is allowed to increase indefinitely. A cycle and a *tail* that leads into it may be thought of as looking like the letter ρ (rho).

One particular design criteria for a binary register might be that the cycles should be long. Ideally, there would be a single cycle of length 2^n.

All of the registers described below can be defined using either Fibonacci or Galois architecture.

The update function of a Fibonacci register is defined through

$$\boldsymbol{s}^i = (s_1^{i-1}, s_2^{i-1}, \ldots, s_{n-1}^{i-1}, s_{n-1}^i)$$

and $s_{n-1}^i = f_{n-1}(\boldsymbol{s}^{i-1})$. That is, most of the state is simply shifted, one bit leaves the register, and one new bit is calculated as a function f_{n-1} of the previous state. It is preferable that s_0^{i-1} is used in the function f_{n-1}, i.e., that the bit which leaves the register influences the new bit. Further, if it is used only linearly, the state will be uniquely invertible.

Consider the sequence $\boldsymbol{u} = (u_0, u_1, \ldots, u_{n-1}, u_n, u_{n+1}, \ldots, u_{N-1})$ created by a Fibonacci register, where $\boldsymbol{u}^0 = (u_0, u_1, \ldots, u_{n-1})$ is the initial state of the register, u_n is the first bit created through the feedback, u_{n+1} the second, etc. In this way, it becomes natural to consider the N-bit sequence $\boldsymbol{u}$ created from the n-bit initial state. The states will be denoted by $\boldsymbol{u}^i = (u_i, u_{i+1}, \ldots, u_{i+n-1})$. At time i, the bit u_i may be thought of as the *output* of the register.

2.9.1 LINEAR FEEDBACK SHIFT REGISTERS (LFSRS)

The linear feedback shift register (LFSR) is a useful primitive as it is straightforward to analyze from the point of cycles and periods.

LFSRS IN FIBONACCI MODE

An LFSR in Fibonacci mode is defined through

$$u_{i+n} = \sum_{j=0}^{n-1} d_j u_{i+j}, \quad d_j \in \mathbb{F}_2.$$

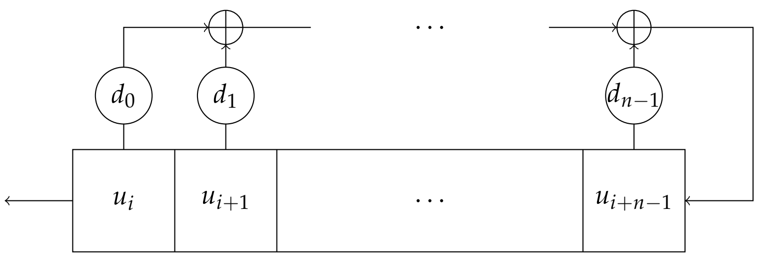


Figure 2.1: An LFSR in Fibonacci architecture.

That is, some particular bits of the state at time i are combined linearly to produce u_{i+n}, see Figure 2.1. It is clear that the all-zero state is a fixed point in the state transition.

Which bits from the state to use in the feedback is determined by the constants d_j. By defining $d_n = 1$ and the *feedback polynomial*

$$g(x) = \sum_{j=0}^{n} d_j x^{n-j},$$

it is possible to relate properties of the sequence $\boldsymbol{u}$ and the polynomial $g(x)$.

First, say that a feedback polynomial of degree n is *reducible* (*irreducible*) if it can (not) be written as a product $g(x) = g_0(x)g_1(x)$ of two polynomials g_i with coefficients in $\mathbb{F}_2$ and degrees $1 \leq \deg(g_i) < n$. Further, say that an irreducible feedback polynomial of degree n is *primitive* if the smallest positive integer j such that $g(x)$ divides $x^j - 1$ is $j = 2^n - 1$.

Proposition 2.1 The nonzero states of an n-bit Fibonacci LFSR will constitute a cycle of length $2^n - 1$ if and only if the feedback polynomial is of degree n and primitive.

For large n, the risk of hitting the all-zero state when the LFSR is initialized at random can be considered practically zero. Since explicit primitive polynomials are known for very large n, it is practically possible to create LFSRs with expected period

$$\frac{1}{2^n} \cdot 1 + \frac{2^n - 1}{2^n} \cdot (2^n - 1) \approx 2^n.$$

Assume that a primitive feedback polynomial is used. With $N = 2^n - 1$, the N-bit sequence $\boldsymbol{u}$ will contain 2^{n-1} ones and $2^{n-1} - 1$ zeros. Further, all windows of size n of the sequence (taking them cyclically, there are $2^n - 1$ such windows) will be pairwise different; all possible n-bit strings except for the

Table 2.1: The number $\mathrm{L}_2(n)$ of irreducible polynomials of degree n for some small values of n.

n	4	5	6	7	8	9	10
$\mathrm{L}_2(n)$	3	6	9	18	30	56	99

all-zero string will appear precisely once. This appears close to optimal from the perspective of random-like behavior. However, the register is inherently linear. It is defined by a linear recursion, and every bit in the sequence $\boldsymbol{u}$ can be written as a sum of at most n bits of the initial state. Given any n bits of the output stream, the initial state can be reconstructed. Thus, an LFSR by itself is not enough to define, e.g., a stream cipher.

The number of irreducible polynomials of degree n, $\mathrm{L}_2(n)$, is related to the so-called Möbius transform, and is given by

$$\mathrm{L}_2(n) = \sum_{d\colon d|n} \mu\left(\frac{n}{d}\right) 2^d,$$

where $d|n$ means that d divides n, and the Möbius function $\mu(m)$ is given by

$$\mu(m) = \mu\left(p_0^{a_0} p_1^{a_1} \cdots p_{j-1}^{a_{j-1}}\right) = \begin{cases} 1, & j = 0, \\ 0, & \exists i: \ a_i > 1, \\ (-1)^j, & a_i = 1, \quad \forall i, \end{cases}$$

where p_i are the j distinct prime numbers factoring m [Sim70]. Some example values of $\mathrm{L}_2(n)$ are given in Table 2.1.

LFSRS IN GALOIS MODE

An LFSR in Galois mode is defined through

$$\boldsymbol{s}^i = (s_1^{i-1}, s_2^{i-1}, \ldots, s_{n-1}^{i-1}, 0) + s_0^{i-1}(d'_0, d'_1, \ldots, d'_{n-2}, d'_{n-1}),$$

and is related to a finite field (Galois field) as follows.

A finite field containing 2^n elements is denoted $\mathbb{F}_{2^n}$ and is defined by an irreducible polynomial $g(x)$ of degree n.[2] The elements of the field are all polynomials with coefficients in $\mathbb{F}_2$ and degree $< n$. Addition and multiplication of two elements is defined modulo $g(x)$. With every vector $\boldsymbol{v} \in \mathbb{F}_2^n$ one can associate an element $v(x) \in \mathbb{F}_{2^n}$ in a natural way:

$$\boldsymbol{v} = (v_0, v_1, \ldots, v_{n-1}) \Longleftrightarrow v(x) = v_{n-1}x^0 + v_{n-2}x^1 + \ldots + v_0 x^{n-1}.$$

[2]Strictly speaking, there is only one finite field containing 2^n elements, up to isomorphism.

The multiplication $w(x)$ of the two elements x and $v(x)$ in the finite field is

$$w(x) = x \cdot \left(\sum_{i=0}^{n-1} v_{n-1-i} x^i \right) \bmod g(x) = \underbrace{\sum_{i=0}^{n-1} v_{n-1-i} x^{i+1}}_{w''(x)} \bmod g(x).$$

If $v_0 = 0$, the reduction modulo $g(x)$ is trivial and $\boldsymbol{w} = \boldsymbol{v} \ll 1$.

If, on the other hand, $v_0 = 1$, the reduction is slightly more involved. Write $g(x) = x^n + g'(x)$, $\deg(g') < n$, and $w''(x) = x^n + w'(x)$, $\deg(w') < n$. Calculating $w''(x) \bmod g(x)$ can be done by finding polynomials $q(x)$ and $r(x)$, $\deg(r) < n$, such that $w''(x) = q(x) \cdot g(x) + r(x)$. Observe that with $q(x) = 1$,

$$r(x) = w''(x) + g(x) = w'(x) + g'(x),$$

and $\deg(r) < n$. That is, $\boldsymbol{w} = (\boldsymbol{v} \ll 1) + \boldsymbol{g}'$.

All in all,

$$w(x) = x \cdot v(x)$$

can be calculated through

$$\boldsymbol{w} = (\boldsymbol{v} \ll 1) + v_0 \cdot \boldsymbol{g}'.$$

Note in particular how this compares to the definition of a Galois LFSR above with feedback polynomial

$$g(x) = \sum_{j=0}^{n} d'_j x^{n-j},$$

where $d'_n = 1$. Further, this shows how to calculate $u(x) \cdot v(x)$ for any elements $u(x), v(x) \in \mathbb{F}_{2^n}$, see Figure 2.2. The upper register is cleared, the middle register is filled with $v(x)$, and the lower register is filled with $u(x)$. The upper register is updated with the contents of the middle one precisely when the output of the lower register is 1. Ignoring the scaling in the figure, the feedback polynomial of the LFSR might be, e.g., $g(x) = 1 + x + x^2 + x^7 + x^{128}$.

2.9.2 NONLINEAR FEEDBACK SHIFT REGISTERS (NFSRS)

In a nonlinear feedback shift register (NFSR) in Fibonacci mode, the feedback function $u_{i+n} = f(\boldsymbol{u}^i)$ is a nonlinear Boolean function of the state. Unlike the theory regarding feedback polynomials for LFSRs, there is no general understanding of how NFSRs behave in terms of (e.g.,) cycles. That is, for random functions f nothing can be said about how many cycles have a particular period.

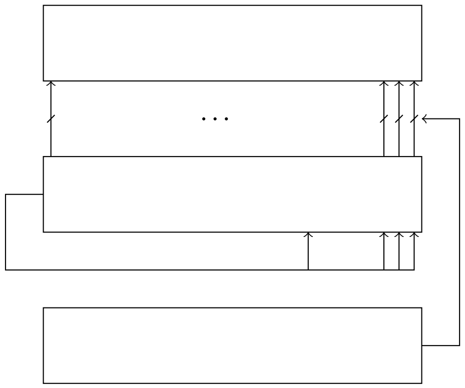


Figure 2.2: A rough hardware implementation of multiplication between two elements in a finite field using a Galois LFSR.

Thus, if an NFSR constitutes the state of, e.g., a stream cipher, there is some probability of the state (and thus, e.g., the keystream) exhibiting short cycles. Even for comparably small n, e.g., $n = 80$, investigating the period distribution by brute force is out of the question.

2.9.3 FEEDBACK WITH CARRY SHIFT REGISTERS (FCSRS)

A feedback with carry shift register (FCSR) is a particular kind of NFSR, where the theory allows precise derivation of the period. An FCSR is defined by an odd number q and computes the 2-adic expansion of the rational number h/q, where h represents the initial state. The 2-adic expansion of h/q, very briefly, is a sequence $\boldsymbol{u}$ such that

$$h/q = \sum_{j=0}^{\infty} u_j 2^j,$$

where the power series does not converge in the usual sense.

FCSRs were proposed by Klapper and Goresky [KG94] and their cryptographic properties were thoroughly examined and determined by the same authors in [KG97]. The aim of this section is to give some understanding of FCSRs, but the treatment will be brief as only small parts of the theory presented here will be used in this dissertation. A general Fibonacci FCSR is given in Figure 2.3.

The state consists of two parts: one main register

$$\boldsymbol{u}^i = (u_{i+0}, u_{i+1}, \ldots, u_{i+n-1})$$

and one integer memory m_i, initialized as $m_0 = 0$. The feedback rule is given by $\boldsymbol{d} = (d_0, d_1, \ldots, d_{n-1}) \in \mathbb{F}_2^n$.

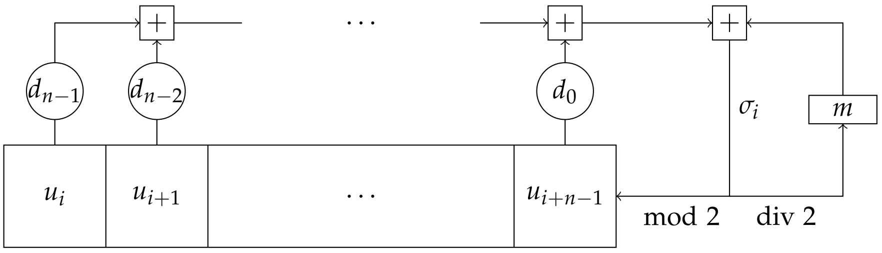


Figure 2.3: An FCSR in Fibonacci architecture.

In each update of the FCSR, the sum

$$\sigma_i = m_{i-1} + \sum_{j=0}^{n-1} u_{i+j} d_{n-j-1}$$

is computed and the state is updated through

$$\begin{aligned} m_i &= \sigma_i \text{ div } 2, && (2.1) \\ u_{i+n} &= \sigma_i \bmod 2. && (2.2) \end{aligned}$$

The FCSR automaton is completely determined by $q = 1 - 2(\boldsymbol{d}) < 0$, the *connection integer*. The size of the main register is given by $n = \lfloor \log(|q| + 1) \rfloor$, i.e., the bit length of d. The state of the FCSR is associated with the integer h_i through

$$h_i = m_i 2^n - \sum_{k=0}^{n-1} \sum_{j=0}^{k} d_{j-1} u_{i+k-j} 2^k,$$

where $d_{-1} = -1$. The output of the FCSR is then the 2-adic expansion of h_i/q and it can be shown [KG97] that the output is strictly periodic if and only if $0 \leq h_i \leq |q|$. If the state at time i corresponds to the integer h_i, then the state h_{i+1} at time $i+1$ corresponds to the integer

$$h_{i+1} = 2^{-1} h_i \bmod q.$$

Thus the ith output of the FCSR is given by $(2^{-i} h_0 \bmod q) \bmod 2$ where h_0 corresponds to the initial state. Now, if q is odd, $0 < h_i < |q|$, and q and h_i are coprime, then the period of the output sequence equals $\text{ord}_q(2)$. Thus, the optimal choice of q is a negative prime with 2 being a primitive root. The FCSR automaton will then produce a maximum length sequence.

The following result [ABM08] will be useful in this dissertation.

Proposition 2.2 Using any FCSR of length n and starting in any state, it takes at most $n+4$ FCSR updates to reach a strictly periodic state.

Let $\mathcal{T}_y$ denote the nonempty set of register taps used when computing σ_i, i.e., $j \in \mathcal{T}_y$ if and only if $d_j = 1$. Note that $|\mathcal{T}_y| = w_{\mathrm{H}}(\boldsymbol{d})$. It was shown in [KG97], that if the memory is initialized with any nonnegative memory m, the memory will decrease exponentially until it reaches

$$0 \leq m \leq |\mathcal{T}_y|,$$

where it will then stay. A slightly stronger formulation is possible if the memory is initialized using, e.g., $m = 0$ as in this dissertation:

Proposition 2.3 If $0 \leq m_i < |\mathcal{T}_y|$, then $0 \leq m_{i+1} < |\mathcal{T}_y|$.

Proof. The smallest possible value m_{i+1} is created with $m_i = 0$ and all taps contributing a 0, yielding $m_{i+1} = 0 \operatorname{div} 2 = 0$. On the other extreme, the largest possible value m_{i+1} is created with $m_i = |\mathcal{T}_y| - 1$ and all taps contributing a 1, yielding $m_{i+1} = (|\mathcal{T}_y| - 1 + |\mathcal{T}_y|) \operatorname{div} 2 < |\mathcal{T}_y|$. ■

Due to this, the carry can be realized using $\lceil \log |\mathcal{T}_y| \rceil$ bits. Thus, an FCSR of length n implicitly has a state which consists of in total $n + \lceil \log |\mathcal{T}_y| \rceil$ bits.

2.10 THE COMBINER

The (nonlinear) combiner is a general approach to constructing a keystream generator from a set of LFSRs. In its purest form, it is mostly of historic interest; Braeken and Lano [BL05] have shown that it is very difficult to design a safe and efficient combiner (cf. Section 4.6).

The general idea is to use m LFSRs of lengths n_j, initialized by the secret key of size $|\boldsymbol{k}| = \sum_{j=0}^{m-1} n_j$, and combine their outputs through a Boolean function $f\colon \mathbb{F}_2^n \to \mathbb{F}_2$ which uses as input one bit from each register,

$$z_i = f(u_i^0, u_i^1, \ldots, u_i^{m-1}).$$

The lengths n_j of the LFSRs should preferably be selected coprime, i.e.,

$$\gcd(n_j, n_{j'}) = 1$$

for all j, j' such that $j \neq j'$.

2.11 THE FILTER GENERATOR

The filter generator is essentially a combiner with $m = 1$, i.e., a single LFSR is used and the sequence $\boldsymbol{u}$ is »filtered« through a Boolean function f. Braeken and Lano [BL05] have shown that it appears to be possible to construct secure and efficient filter generators (cf. Section 4.6).

The filter generator models in a sense the simplest possible keystream generator and has been thoroughly analyzed in the context of fast correlation attacks (cf. Section 4.8).

2.12 THE GRAIN FAMILY OF STREAM CIPHERS

In this section, the Grain family of stream ciphers is briefly introduced. The purpose of this is two-fold: it is given as an example of synchronous stream ciphers, but also because some familiarity with the Grain family will be useful in Chapters 5 and 6.

Grain is notable for its extremely small hardware representation. During the initial phase of the eSTREAM project, the original version, Grain v0, was strengthened after some observations by Berbain et al. [BGM06]. The final version is known as Grain v1 [HJM06].

Like the other portfolio ciphers, Grain v1 is modern in the sense that it allows for public IVs, yet they only use 80-bit keys. Recognizing the emerging need for 128-bit keys, Hell et al. [HJMM06] proposed Grain-128 supporting 128-bit keys and 96-bit IVs. The design is akin to that of 80-bit Grain, but notably, the nonlinear parts of the cipher have smaller degrees than their counterparts in Grain v1.

The general idea of Grain with key size $|\boldsymbol{k}|$ is to use an NFSR and an LFSR, each of size $|\boldsymbol{k}|$, and an output function which uses state material from both registers. The LFSR feeds into the NFSR. The NFSR is loaded with the key, and the LFSR is loaded with the IV, padded with a constant. After key-and-IV-loading, but before keystream generation, the state is initialized in $2 \cdot |\boldsymbol{k}|$ state updates where the suppressed output is fed back to the LFSR. After initialization, keystream generation begins.

Implementations of any keystream generator matching the above description can be made faster by implementing the state update and output function several times in parallel. One particular feature of the Grain ciphers is that newly produced bits in the registers are not used for several clockings. This means that no recursion is necessary to implement the parallel functions.

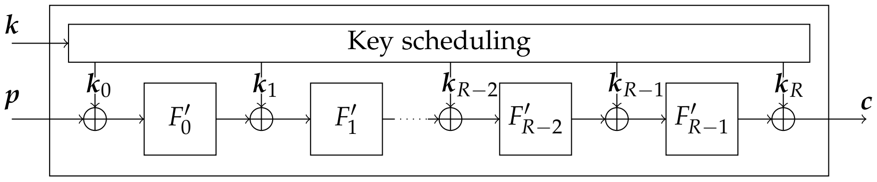


Figure 2.4: A key-alternating block cipher.

2.13 KEY-ALTERNATING BLOCK CIPHERS

A typical approach to designing a block cipher is to load the plaintext into a state, which is then updated several times using a round function. To facilitate implementation, the round function is typically either the same in all rounds, or is only altered to some small extent, e.g., by the use of round constants. Between each round function application, key material is added to the state. The idea is that repeated application of the relatively simple round function will mix the key and plaintext material to produce a ciphertext which is in some sense a very complicated function of the inputs.

This is called a *key-alternating block cipher*, see Figure 2.4. AES is a key-alternating block cipher, as are all the block ciphers which will be studied in this dissertation. While the design of round functions is fairly straightforward, the design of key schedules, i.e., the possibly nonlinear rules that derive the round keys from the master key $\boldsymbol{k}$, is more ad hoc with no clear consensus on what is »enough« and what is »not enough« [AÅBL12], e.g., in terms of nonlinearity. Indeed, all results on block ciphers in this dissertation relate to properties of the key schedules.

This dissertation will denote by R the number of rounds in the block cipher, while r will be used to index the rounds. In several designs, either $\boldsymbol{k}_0$ or $\boldsymbol{k}_R$ is zero. It is then natural to combine the round function F'_r and the round key $\boldsymbol{k}_{r+1}$ or $\boldsymbol{k}_r$ into F_r. With $0 \le r_1 \le r_2 \le R$, define $F_{r_1,r_2}(\boldsymbol{s},\boldsymbol{k})$ as the partial encryption that applies rounds $r_1, r_1+1, \dots, r_2-1$ to the state $\boldsymbol{s}$ using key $\boldsymbol{k}$. Similarly, $F^{-1}_{r_1,r_2}(\boldsymbol{s},\boldsymbol{k})$ applies the decryption of rounds $r_2-1, \dots, r_1+1, r_1$ to the state $\boldsymbol{s}$ using key $\boldsymbol{k}$. The full block cipher can then be decomposed as, e.g.,

$$\boldsymbol{c} = F(\boldsymbol{p},\boldsymbol{k}) = F_{0,R}(\boldsymbol{p},\boldsymbol{k}) = F_{r,R}(F_{0,r}(\boldsymbol{p},\boldsymbol{k}),\boldsymbol{k})$$

for any $r \in \{0, 1, \dots, R\}$.

One type of key-alternating block cipher is the *substitution–permutation network* (SPN), where the round function consists of a nonlinear state update using Sboxes and a linear state update. The simplest type of linear state update is a bit permutation as in, e.g., PRESENT [BKL+07], while AES uses a linear layer where each input bit affects several output bits.

3 Tools for Cryptanalysis

Some useful tools are briefly introduced in this chapter, namely various transforms of Boolean and vectorial functions, matrix theory, coding theory, and hypothesis testing, all of which will be used in the sequel.

3.1 TRANSFORMS

The Fourier transform, and its inverse transform, is a common tool in physics and engineering. It deals with time-continuous systems and provides a mapping between the time and frequency domains.

It is possible to define discrete counterparts that turn out to be very useful in several cryptographic settings, and two such transforms will be used in this dissertation. They will be referred to as the discrete Fourier transform and the Walsh transform, respectively. The discrete Fourier transform will be referred to simply as the Fourier transform as no confusion with the time-continuous version should occur in this dissertation as a result of this slight naming simplification. A compact and content-rich introduction to these transforms by Carlet can be found in [Car10a] [Car10b]. Several names appear in the literature for these transforms; this dissertation uses the same naming as Carlet, but another notation.

3.1.1 THE (DISCRETE) FOURIER TRANSFORM

The Fourier transform of the Boolean function $f(\boldsymbol{x}) = f(x_0, x_1, \ldots, x_{n-1})$ is a real-valued function over $\mathbb{F}_2^n$ which is defined as

$$\mathcal{F}(f)(\boldsymbol{\omega}) = \sum_{\boldsymbol{x}\in\mathbb{F}_2^n} f(\boldsymbol{x})(-1)^{\langle \boldsymbol{\omega},\boldsymbol{x}\rangle}, \quad \boldsymbol{\omega} \in \mathbb{F}_2^n. \tag{3.1}$$

If there is a nonzero bit in $\boldsymbol{x} \in \mathbb{F}_2^n$, then two $\boldsymbol{\omega} \in \mathbb{F}_2^n$ which differ in precisely this bit will give distinct values of $\langle \boldsymbol{\omega}, \boldsymbol{x} \rangle$. This leads to the following, which will be very useful in this dissertation.

Proposition 3.1 $\langle \boldsymbol{\omega}, \boldsymbol{x} \rangle$ is a balanced function of $\boldsymbol{\omega}$ for $\boldsymbol{x} \neq \mathbf{0}$, and constantly 0 for $\boldsymbol{x} = \mathbf{0}$. Consequently,

$$\sum_{\boldsymbol{\omega} \in \mathbb{F}_2^n} (-1)^{\langle \boldsymbol{\omega}, \boldsymbol{x} \rangle} = \begin{cases} 2^n, & \boldsymbol{x} = \mathbf{0}, \\ 0, & \boldsymbol{x} \neq \mathbf{0}. \end{cases}$$

Proposition 3.2 The Boolean function f can be recovered from its Fourier transform $\mathcal{F}(f)$ through

$$f(\boldsymbol{x}) = 2^{-n} \sum_{\boldsymbol{\omega} \in \mathbb{F}_2^n} \mathcal{F}(f)(\boldsymbol{\omega})(-1)^{\langle \boldsymbol{\omega}, \boldsymbol{x} \rangle}.$$

Proof.

$$\begin{aligned} \sum_{\boldsymbol{\omega} \in \mathbb{F}_2^n} \mathcal{F}(f)(\boldsymbol{\omega})(-1)^{\langle \boldsymbol{\omega}, \boldsymbol{x} \rangle} &= \sum_{\boldsymbol{\omega} \in \mathbb{F}_2^n} \left(\sum_{\boldsymbol{y} \in \mathbb{F}_2^n} f(\boldsymbol{y})(-1)^{\langle \boldsymbol{\omega}, \boldsymbol{y} \rangle} \right) (-1)^{\langle \boldsymbol{\omega}, \boldsymbol{x} \rangle} \\ &= \sum_{\boldsymbol{\omega} \in \mathbb{F}_2^n} \sum_{\boldsymbol{y} \in \mathbb{F}_2^n} f(\boldsymbol{y})(-1)^{\langle \boldsymbol{\omega}, \boldsymbol{y}+\boldsymbol{x} \rangle} \\ &= \sum_{\boldsymbol{y} \in \mathbb{F}_2^n} f(\boldsymbol{y}) \sum_{\boldsymbol{\omega} \in \mathbb{F}_2^n} (-1)^{\langle \boldsymbol{\omega}, \boldsymbol{y}+\boldsymbol{x} \rangle} \\ &= 2^n f(\boldsymbol{x}), \end{aligned}$$

where the last step follows from Proposition 3.1. ■

A naive algorithm for calculating $\mathcal{F}(f)$ would require time 2^{2n}: for each $\boldsymbol{\omega}$, sum over all $\boldsymbol{x}$. However, with $\boldsymbol{\omega}^{n-1} = (\omega_0, \omega_1, \ldots, \omega_{n-2})$ and $\omega_{n-1} \in \mathbb{F}_2$,

$$\begin{aligned} \mathcal{F}(f)(\boldsymbol{\omega}) &= \mathcal{F}(f)(\boldsymbol{\omega}^{n-1} || \omega_{n-1}) \\ &= \sum_{\boldsymbol{x} \in \mathbb{F}_2^{n-1}} \left(f(\boldsymbol{x}||0)(-1)^{\langle \boldsymbol{\omega}^{n-1}, \boldsymbol{x} \rangle} + f(\boldsymbol{x}||1)(-1)^{\langle \boldsymbol{\omega}^{n-1}, \boldsymbol{x} \rangle}(-1)^{\omega_{n-1}} \right) \\ &= \sum_{\boldsymbol{x} \in \mathbb{F}_2^{n-1}} f(\boldsymbol{x}||0)(-1)^{\langle \boldsymbol{\omega}^{n-1}, \boldsymbol{x} \rangle} + (-1)^{\omega_{n-1}} \sum_{\boldsymbol{x} \in \mathbb{F}_2^{n-1}} f(\boldsymbol{x}||1)(-1)^{\langle \boldsymbol{\omega}^{n-1}, \boldsymbol{x} \rangle}. \end{aligned}$$

That is, the two Fourier coefficients $\mathcal{F}(f)(\boldsymbol{\omega}^{n-1}||0)$ and $\mathcal{F}(f)(\boldsymbol{\omega}^{n-1}||1)$ are related and large parts of the calculations can be shared. From a generalized version of this observation, a recursive »butterfly« algorithm can be derived, which calculates $\mathcal{F}(f)$ using time $\mathcal{O}(n2^n)$ and memory $\mathcal{O}(n2^n)$.

Finally, define $\mathrm{L}_1(f) = 2^{-n} \sum_{\boldsymbol{\omega}} |\mathcal{F}(f)(\boldsymbol{\omega})|$. The following holds [KM93].

Proposition 3.3 With $f(\boldsymbol{x}) = f_0(\boldsymbol{x}) f_1(\boldsymbol{x})$, $\mathrm{L}_1(f) \leq \mathrm{L}_1(f_0)\, \mathrm{L}_1(f_1)$.

3.1.2 THE WALSH TRANSFORM

The Walsh transform of the Boolean function $f(x) = f(x_0, x_1, \ldots, x_{n-1})$ is defined as

$$\widehat{f}(\omega) = \sum_{x \in \mathbb{F}_2^n} (-1)^{f(x)+\langle \omega, x\rangle}, \quad \omega \in \mathbb{F}_2^n. \tag{3.2}$$

Given the similarity with the Fourier transform (the Walsh transform is indeed nothing else than the »Fourier transform« applied to the *sign function* $(-1)^{f(x)}$ rather than $f(x)$ itself), the following can be readily derived (cf. Proposition 3.2):

Proposition 3.4 The Boolean function f can be recovered from its Walsh transform $\widehat{f}$ through

$$(-1)^{f(x)} = 2^{-n} \sum_{\omega \in \mathbb{F}_2^n} \widehat{f}(\omega)(-1)^{\langle \omega, x\rangle}.$$

Also, it should not be surprising that a similar butterfly algorithm exists for relatively fast calculation of the Walsh transform.

The following will be useful in Chapter 10 [Car10a, page 275].

Proposition 3.5 (the Poisson summation formula) Consider a Boolean function $f\colon \mathbb{F}_2^n \to \mathbb{F}_2$ and a subspace U of $\mathbb{F}_2^n$. For every $x \in \mathbb{F}_2^n$,

$$\sum_{\alpha \in U^\perp} (-1)^{\langle \alpha, x\rangle} \widehat{f}(\alpha) = \left|U^\perp\right| \sum_{y \in U+x} (-1)^{f(y)}.$$

3.1.3 THE VECTORIAL WALSH TRANSFORM

Consider the function $F\colon \mathbb{F}_2^n \to \mathbb{F}_2^m$, and for $\beta \in \mathbb{F}_2^m$ the component function $\langle \beta, F\rangle$, where $\langle \mathbf{0}, F\rangle = 0$ is treated as a valid component function. The vectorial Walsh transform $\widehat{F}$ of the function F is the mapping which maps an ordered pair $(\alpha, \beta) \in \mathbb{F}_2^n \times \mathbb{F}_2^m$ to the value at α of the Walsh transform of $\langle \beta, F\rangle$, i.e.,

$$\widehat{F}(\alpha, \beta) = \sum_{x \in \mathbb{F}_2^n} (-1)^{\langle \beta, F(x)\rangle + \langle \alpha, x\rangle}.$$

The following will be useful (cf. Proposition 3.4):

Proposition 3.6 The vectorial function F can be recovered from its Walsh transform $\widehat{F}$ through

$$(-1)^{\langle \beta, F(x)\rangle} = 2^{-n} \sum_{\alpha \in \mathbb{F}_2^n} \widehat{F}(\alpha, \beta)(-1)^{\langle \alpha, x\rangle}.$$

3.2 MATRIX THEORY

Consider an $|I| \times |J|$ matrix $M = (m_{i,j})_{i \in I, j \in J}$ containing $|I| \times |J|$ elements $m_{i,j}$ in $|I|$ rows and $|J|$ columns. Denote by M^{T} the complex-conjugated transpose of M, and by I the identity matrix of some square size which should be clear from the context. The $m \times n$ matrix product $M = M_1 M_0$ of two matrices M_1 and M_0 of sizes $m \times k$ and $k \times n$, respectively, is defined through

$$m_{i,j} = \sum_{l=0}^{k-1} m^1_{i,l} m^0_{l,j}.$$

In the sequel, indices may be given using vectors rather than integers. This should be interpreted as $m_{\boldsymbol{\alpha},\boldsymbol{\beta}} = m_{(\boldsymbol{\alpha}),(\boldsymbol{\beta})}$.

A square matrix M is said to be *normal* if $M^{\mathrm{T}}M = MM^{\mathrm{T}}$ and *unitary* if $M^{\mathrm{T}}M = MM^{\mathrm{T}} = \mathrm{I}$. If a matrix is unitary and all elements are real-valued, it is said to be *orthogonal*. The *rank* of a matrix is the number of linearly independent rows.

3.2.1 TOEPLITZ MATRICES

A *Toeplitz matrix* is a matrix where each descending diagonal from right to left is constant. That is, a Toeplitz matrix M is of the form

$$M = \begin{pmatrix} m_0 & m_1 & \cdots & m_{|J|-1} \\ m_1 & m_2 & \cdots & m_{|J|} \\ m_2 & m_3 & \cdots & m_{|J|+1} \\ \vdots & \vdots & \ddots & \vdots \\ m_{|I|-1} & m_{|I|} & \cdots & m_{|J|+|I|-2} \end{pmatrix}.$$

and is determined by the $|I| + |J| - 1$ elements of the top row and right column.

Toeplitz matrices are usually defined using constant *left-to-right* diagonals, but the definition presented here will be easier to work with in Chapter 5.

3.2.2 EIGENVECTORS AND SIMILAR MATRICES

A nonzero vector $\boldsymbol{v}$, with $|\boldsymbol{v}| = |I|$, is a (left) eigenvector of M with eigenvalue λ if $\boldsymbol{v}M = \lambda\boldsymbol{v}$. Similarly, a nonzero vector $\boldsymbol{v}^{\mathrm{T}}$, with $|\boldsymbol{v}| = |J|$, is a (right) eigenvector of M with eigenvalue λ if $M\boldsymbol{v}^{\mathrm{T}} = \lambda\boldsymbol{v}^{\mathrm{T}}$. A vector is said to be *normalized* if $\boldsymbol{v}\boldsymbol{v}^{\mathrm{T}} = 1$. Clearly, all eigenvectors can be normalized if wanted.

Consider from now on quadratic matrices M. To each $m \times m$ matrix M, one can associate a polynomial $\det(M)$ of degree m, known as the *determinant* of M. Further, the *characteristic polynomial* of M is the polynomial $\det(M - \lambda\mathrm{I})$, and its roots are the m (not necessarily distinct) eigenvalues λ_i. The *algebraic multiplicity* of an eigenvalue is the multiplicity of the corresponding root of the characteristic polynomial. The number of linearly independent eigenvectors with a particular eigenvalue is referred to as the *geometric multiplicity* of the eigenvalue. These multiplicities are not necessarily the same, but the algebraic multiplicity is never less than the geometric multiplicity.

Two matrices M and L are *similar* if $M = P^{-1}LP$ for some invertible matrix P. Several different similarities are common. As an example, every quadratic matrix M is similar to a *Jordan matrix*, i.e., a matrix with the pattern

$$J = \begin{pmatrix} J_0 & 0 & \dots & 0 \\ 0 & J_1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & J_{s-1} \end{pmatrix},$$

where $J_0, J_1, \dots, J_{s-1}$ are *Jordan blocks*,

$$J_i = \begin{pmatrix} \lambda & 1 & 0 & \dots & 0 \\ 0 & \lambda & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \dots & \lambda & 1 \\ 0 & 0 & \dots & 0 & \lambda \end{pmatrix},$$

where the size of J_i is related to the algebraic and geometric multiplicity of the eigenvalues of the matrix M and in particular, if the algebraic multiplicity of an eigenvalue λ is 1, then the corresponding $J_i = (\lambda)$ is of size 1×1.

All normal matrices are *diagonalizable*, i.e., they are similar to a *diagonal* matrix, and can be written as $M = U^{\mathrm{T}}DU$, where

$$U = \begin{pmatrix} \boldsymbol{v}^0 \\ \boldsymbol{v}^1 \\ \vdots \\ \boldsymbol{v}^{m-1} \end{pmatrix}$$

is a unitary matrix with its rows taken as the normalized left eigenvectors of M, and D is a diagonal matrix containing the corresponding eigenvalues, i.e.,

$$D = \begin{pmatrix} \lambda_0 & 0 & \dots & 0 \\ 0 & \lambda_1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \lambda_{m-1} \end{pmatrix}.$$

It follows that all eigenvalues of a unitary matrix lie on the unit circle, $|\lambda_i| = 1$. Further, one can bound the eigenvalues of a submatrix of a unitary matrix.[1]

Proposition 3.7 Consider any unitary matrix $M = (m_{i,j})_{i\in I, j\in J}$. For any subsets $I' \subseteq I$, $J' \subseteq J$, define the submatrix $A = (m_{i,j})_{i\in I', j\in J'}$. Then the eigenvalues λ_i of A lie on the unit disk, i.e., they satisfy $|\lambda_i| \leq 1$.

3.2.3 ASYMPTOTIC BEHAVIOR OF MATRIX POWERS

The following result by Oldenburger [Old40] relates the eigenvectors and eigenvalues of M and the asymptotic behavior of M^R.[2]

Proposition 3.8 Consider an $m \times m$ matrix M with left eigenvectors $\boldsymbol{v}^i$, $0 \leq i < m$ and corresponding eigenvalues λ_i, i.e.,

$$\boldsymbol{v}^i M = \lambda_i \boldsymbol{v}^i, \quad 0 \leq i < m.$$

Then the sequence M^R converges if and only if $|\lambda_i| < 1$ for all i, except possibly $\lambda_i = 1$ for one or more i, but then the algebraic and geometric multiplicities of the eigenvalue 1 must be the same.

A proof given by Friedberg and Insel [FI92] might be more accessible than that in the original paper by Oldenburger. By making some further assumptions, the limit can be derived explicitly and the proof becomes rather straightforward.

Proposition 3.9 Consider the same setting as in Proposition 3.8. Further assume that the matrix M is unitary. Assume that all eigenvectors are normalized and that all eigenvalues λ_i fulfill either $|\lambda_i| < 1$ or $\lambda_i = 1$. Then

$$M^R \to \sum_{i\colon \lambda_i = 1} \boldsymbol{v}^{i\mathrm{T}} \boldsymbol{v}^i, \quad R \to \infty.$$

Proof. Since M is unitary, it is diagonalizable, $M = U^{\mathrm{T}} D U$, where the rows of U are the eigenvectors $\boldsymbol{v}^i$, and $UU^{\mathrm{T}} = \mathrm{I}$. It follows that

$$M^R = U^{\mathrm{T}} D^R U = \sum_i \boldsymbol{v}^{i\mathrm{T}} \lambda_i^R \boldsymbol{v}^i.$$

Finally, $\lambda_i^R = 1$ for i such that $\lambda_i = 1$ and $\lambda_i^R \to 0$, $R \to \infty$, otherwise. ■

[1] The proposition can be shown using the induced Euclidean norm; vector and matrix norms are not considered in this dissertation.

[2] Which particular norm is used to define the asymptotics is not of interest as only matrices of finite sizes are considered, in which case all matrix norms are equivalent.

For diagonalizable (but not necessarily unitary) matrices M, the above technique, together with *unitary equivalence*, not covered here, can be used to show that M^R does indeed converge. For matrices that are not diagonalizable, something more involved has to be done, e.g., looking at unitary equivalence with upper triangular matrices [FI92] or the Jordan form [Old40].

The work presented in this dissertation makes use of a particular formulation, which is possible if the limit M^R exists by Proposition 3.8 and the eigenvalue 1 has algebraic multiplicity 1. This follows from, e.g., the Jordan form of the matrix M, similar to Proposition 3.9.

Proposition 3.10 Consider the same setting as in Proposition 3.8. Assume that all eigenvectors are normalized and that $\lambda_0 = 1$ and $|\lambda_i| < 1$, $0 < i < m$. Then

$$M^R \to \boldsymbol{v}^{0\mathrm{T}}\boldsymbol{v}^0, \quad R \to \infty.$$

3.3 A BRIEF INTRODUCTION TO CODING

Error control coding, or channel coding, is commonly used when information should reliably be transferred over a noisy channel, and has several connections to cryptography, including Shannon, who developed several fundamental results regarding what is theoretically (not) achievable in coding.

Assume that the goal is to transmit a k-bit message $\boldsymbol{a}$ over a noisy channel, e.g., a binary symmetric channel (BSC) which flips each transmitted bit with probability $p_{\text{BSC}} \neq 0$. This is accomplished by adding redundancy: the k-bit message $\boldsymbol{a}$ is mapped to an n-bit codeword $\boldsymbol{x} = \phi(\boldsymbol{a}) \in \mathcal{B} \subset \mathbb{F}_2^n$, $|\mathcal{B}| = 2^k$. The channel may introduce noise, i.e., the receiver observes $\boldsymbol{y}$ which is possibly different from $\boldsymbol{x}$, but by exploiting the added redundancy, a »best guess« codeword $\hat{\boldsymbol{y}}$ can be recovered, suggesting an information sequence $\hat{\boldsymbol{x}}$, where hopefully $\hat{\boldsymbol{x}} = \boldsymbol{x}$.

Shannon defined the *rate* $\frac{k}{n}$ and the *channel capacity* C, which for the BSC would depend on p_{BSC}, and showed that for fixed $\frac{k}{n} < C$, arbitrarily small bit error rates are possible if $n \to \infty$. Conversely, for $\frac{k}{n} > C$, arbitrarily small bit error rates are not possible.

3.3.1 RANDOM CODING

Shannon used random coding arguments to show the result outlined above. That is, they considered randomly chosen mappings ϕ. For such a random code it is not necessarily possible to implement a practical encoder and decoder. Various codes have been proposed and some, in particular turbo codes and low-density parity-check (LDPC) codes, seem both practical and able to operate with very low bit error rates at rates very close to the channel capacity.

3.3.2 LINEAR CODES

A linear code has the property that every sum of codewords is a codeword. It can be specified through its $k \times n$ *generator matrix* G, so that the information sequence $\boldsymbol{a} \in \mathbb{F}_2^k$ is mapped to the codeword $\boldsymbol{x} \in \mathbb{F}_2^n$ through

$$\boldsymbol{x} = \boldsymbol{a}G.$$

The possibility of error correction is related to the *minimum distance* of the code,

$$d_{\min} = \min_{\substack{\boldsymbol{a},\boldsymbol{a}' \in \mathbb{F}_2^k, \\ \boldsymbol{a} \neq \boldsymbol{a}'}} d_{\mathrm{H}}\left(\boldsymbol{a}G, \boldsymbol{a}'G\right),$$

since if $d_{\min}$ is smaller, fewer errors are needed to make an erroneous decoding possible. Due to the linearity,

$$d_{\min} = \min_{\substack{\boldsymbol{a} \in \mathbb{F}_2^k, \\ \boldsymbol{a} \neq \boldsymbol{0}}} w_{\mathrm{H}}\left(\boldsymbol{a}G\right).$$

That is, the minimum distance is precisely the weight of the nonzero codeword with the lowest weight. Computing this characteristic of the code is generally difficult. The problem of finding the minimum distance $d_{\min}$ of a (binary) linear code is NP-hard and the corresponding decision problem is NP-complete [Var97]. Some algorithms for finding minimum-weight codewords are Stern's algorithm [Ste89]—with implementation aspects covered by, e.g., Canteaut and Chabaud [CC98] and Bernstein et al. [BLP08]—and one algorithm by May et al. [MMT11] and Johansson and Löndahl [JL11], independently.

3.4 HYPOTHESIS TESTING

Example 3.1 Consider the independent and identically distributed (iid) random binary variables X_i, $0 \leq i < n$ drawn according to either the probability distribution P_0 for which

$$\mathbf{Pr}_{P_0}\left[X = 0\right] = \frac{1}{2} + \varepsilon,$$
$$\mathbf{Pr}_{P_0}\left[X = 1\right] = \frac{1}{2} - \varepsilon,$$

$\varepsilon \neq 0$, or from the probability distribution P_1 for which

$$\mathbf{Pr}_{P_1}\left[X = 0\right] = \frac{1}{2},$$
$$\mathbf{Pr}_{P_1}\left[X = 1\right] = \frac{1}{2}.$$

Assume that n corresponding samples are available, i.e., $x_0, x_1, \ldots, x_{n-1}$ have been observed. For large enough n, it should be possible to distinguish between the two distributions, i.e., given samples $x_0, x_1, \ldots, x_{n-1}$, it should be possible to tell, with small probability of error, which of the two distributions P_0 and P_1 the samples belong to, i.e., give $\hat{b} \in \{0,1\}$ such that $\mathbf{Pr}\left[b = \hat{b}\right]$ is significantly larger than one half.

It is straightforward to compute $s = \sum_{i=0}^{n-1} x_i$, where $0 \leq s \leq n$. Now, if the distribution used is P_0, one expects $s = n\left(\frac{1}{2} - \varepsilon\right)$, while if the distribution is P_1, one expects $s = \frac{n}{2}$. One might define a threshold $\theta = n\left(\frac{1}{2} - \frac{\varepsilon}{2}\right)$ and (for $\varepsilon > 0$) construct a decision rule such as »when $s \leq \theta$, say $\hat{b} = 0$.« When making a decision in this way, two types of errors are possible: false positives ($b = 1$, $\hat{b} = 0$) and false negatives ($b = 0$, $\hat{b} = 1$). □

The above is a relatively simple example of hypothesis testing, as it is binary both in the sense that the random variables are binary, and in that it is known which two distributions are possible.

In the remainder of this section, this problem will be treated more formally. In particular it will be derived, approximately, how large n must be in order to guarantee small error probabilities. First, consider the setting with two known distributions, P_0 and P_1.

Lemma 3.11 (Neyman-Pearson) Let $X_0, X_1, \ldots, X_{n-1}$ be drawn iid according to the probability distribution P_b, $b \in \{0,1\}$. Consider the decision problem corresponding to the hypotheses $P_b = P_0$ vs. $P_b = P_1$. For $T \geq 0$ define a region

$$\mathcal{A}_n(T) = \left\{ \frac{\mathbf{Pr}_{P_0}\left[x_0, x_1, \ldots, x_{n-1}\right]}{\mathbf{Pr}_{P_1}\left[x_0, x_1, \ldots, x_{n-1}\right]} > T \right\}.$$

Let $\alpha = \mathbf{Pr}_{P_0}\left[\mathcal{A}_n^c(T)\right]$ and $\beta = \mathbf{Pr}_{P_1}\left[\mathcal{A}_n(T)\right]$ be the error probabilities corresponding to the decision region $\mathcal{A}_n(T)$, where $\mathcal{A}_n^c(T)$ is the complement of $\mathcal{A}_n(T)$. Let $\mathcal{B}_n$ be any other decision region with associated error probabilities α' and β'. If $\alpha' \leq \alpha$, then $\beta' \geq \beta$.

Now define the *relative entropy* between two probability distributions over the same domain (also known as Kullback-Leibler distance or information divergence) as

$$D\left(\mathbf{Pr}_{P_0}[x] \,||\, \mathbf{Pr}_{P_1}[x]\right) = \sum_{x \in P_0} \mathbf{Pr}_{P_0}[x] \log \frac{\mathbf{Pr}_{P_0}[x]}{\mathbf{Pr}_{P_1}[x]},$$

and write $D\left(P_0 || P_1\right) = D\left(\mathbf{Pr}_{P_0}[x] \,||\, \mathbf{Pr}_{P_1}[x]\right)$ for convenience.

Assume that a set of iid data has been observed from the distribution P_b with $b \in \{0,1\}$ and consider the *null hypothesis* H_0 and the *alternate hypothesis* H_1, where the hypothesis $H_{\hat{b}}$ is the hypothesis that $b = \hat{b}$. A decision rule determines a $\hat{b}$ for each x. Define $\alpha = \mathbf{Pr}\left[b = 0, \hat{b} = 1\right]$ (false negative probability) and $\beta = \mathbf{Pr}\left[b = 1, \hat{b} = 0\right]$ (false positive probability). Unfortunately, α and β can not be computed exactly, so the performance of the test is not known in general, but some asymptotic expressions for the error probabilities are available.

First, write $\frac{1}{2} + \varepsilon = \frac{1}{2}(1 + 2\varepsilon)$, and use the Taylor approximation

$$\ln(1+x) \approx x - \frac{x^2}{2}$$

to write

$$\ln\left(\frac{1}{2} + \varepsilon\right) = \ln(1+2\varepsilon) - \ln 2 \approx 2\varepsilon - 2\varepsilon^2 - \ln 2.$$

In this way, approximate

$$\begin{aligned} D\left(P_0 || P_1\right) &= \left(\frac{1}{2} + \varepsilon\right) \log \frac{\frac{1}{2} + \varepsilon}{\frac{1}{2}} + \left(\frac{1}{2} - \varepsilon\right) \log \frac{\frac{1}{2} - \varepsilon}{\frac{1}{2}} \\ &= \frac{1}{\ln 2}\left(\left(\frac{1}{2} + \varepsilon\right) \ln(1+2\varepsilon) + \left(\frac{1}{2} - \varepsilon\right) \ln(1-2\varepsilon)\right) \\ &\approx 1 + \frac{1}{2\ln 2}\left((1+2\varepsilon)(2\varepsilon - 2\varepsilon^2) + (1-2\varepsilon)(-2\varepsilon - 2\varepsilon^2)\right) \\ &= \frac{4\varepsilon^2}{2\ln 2} = \frac{2\varepsilon^2}{\ln 2}. \end{aligned} \tag{3.3}$$

3.4.1 KEEPING α AND β SIMILAR

Define the probability of error $p_e = \frac{1}{2}(\alpha + \beta)$, assuming that it is equally probable that the samples come from P_0 or P_1. Define $S = \sum_{i=0}^{n-1} X_i$ and assume that n is large. Then the central limit theorem suggests that S is normally distributed and the expected value and standard deviation of S are

$$\mu_b = \begin{cases} n\left(\frac{1}{2} - \varepsilon\right), & b = 0, \\ \frac{n}{2}, & b = 1, \end{cases}$$

and

$$\sigma_b = \begin{cases} \frac{1}{2}\sqrt{n\left(1 - 4\varepsilon^2\right)}, & b = 0, \\ \frac{\sqrt{n}}{2}, & b = 1, \end{cases}$$

respectively.

Assume that the decision is made based on a threshold

$$\theta = \frac{n}{2} - \frac{\varepsilon}{2} = \frac{1}{2}(n - \varepsilon)$$

(cf. Example 3.1). The probability function of the standard normal distribution, with expected value μ and standard deviation σ, is

$$\Phi_{\sigma}^{\mu}(t) = \frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^{t} e^{-\frac{1}{2}\left(\frac{u-\mu}{\sigma}\right)^2} \mathrm{d}u,$$

so $\alpha = 1 - \Phi_{\sigma_0}^{\mu_0}(\theta)$ and $\beta = \Phi_{\sigma_1}^{\mu_1}(\theta)$, i.e.,

$$\alpha = \frac{1}{\sigma_0\sqrt{2\pi}} \int_{\frac{1}{2}(1-\varepsilon)}^{+\infty} e^{-\frac{1}{2}\left(\frac{u-\mu_0}{\sigma_0}\right)^2} \mathrm{d}u$$

and

$$\beta = \frac{1}{\sigma_1\sqrt{2\pi}} \int_{-\infty}^{\frac{1}{2}(1-\varepsilon)} e^{-\frac{1}{2}\left(\frac{u-\mu_1}{\sigma_1}\right)^2} \mathrm{d}u.$$

Assume that ε is small so that the distributions are close. With

$$n = \frac{\ln 2 \cdot d}{4 \cdot \varepsilon^2} \approx \frac{d}{2D\left(P_0||P_1\right)}$$

(cf. Equation 3.3), it follows that $p_{\mathrm{e}} \approx \Phi_1^0\left(-\frac{\sqrt{d}}{2}\right)$ by Theorem 6 in [BJV04]. As an example, with $d = 1$, $p_{\mathrm{e}} \approx .35$.

3.4.2 FIXING α

The relative entropy $D\left(P_0||P_1\right)$ from above is related to the asymptotic behavior of α and β through what is commonly referred to as Stein's lemma. That is, by fixing α, β decreases with increasing n and

$$\frac{\log \beta}{n} \to -D\left(P_0||P_1\right), \quad n \to \infty.$$

From this,

$$\beta \approx 2^{-nD(P_0||P_1)},$$

for large enough n.

Now study a slightly more general situation than above: assume that 2^L different sequences have been observed—$2^L - 1$ correspond to P_1 and 1 corresponds to P_0. The aim is to identify which sequence corresponds to P_0. Fixing α, one can expect the number of false positives to be approximately

$$\left(2^L - 1\right) 2^{-nD(P_0||P_1)},$$

and setting this to 1 yields

$$n \approx \frac{L}{D\left(P_0 || P_1\right)}.$$

By Equation 3.3,

$$n \approx \frac{L \ln 2}{2\varepsilon^2},$$

and in particular when $L = 1$,

$$n \approx \frac{1}{\varepsilon^2},$$

which is a common rule of thumb in cryptanalysis: when trying to distinguish between two distributions as above, one (asymptotically) needs approximately ε^{-2} samples.

4 Techniques for Cryptanalysis

In this chapter, several techniques are presented which can be used in cryptanalysis. These range from general to specific, including, e.g., analysis of Boolean functions and truncated related-key differential cryptanalysis of block ciphers.

4.1 ATTACK SETTINGS

The aim of this section is to go through some attack settings, i.e., what information the attacker is assumed to have access to.

4.1.1 KNOWN-CIPHERTEXT ATTACKS

One example of an attack on, e.g., a block cipher would be the following: given a ciphertext $\boldsymbol{c}$, the attack finds the corresponding plaintext $\boldsymbol{p}$ and key $\boldsymbol{k}$. This would compromise the particular message targeted, but also all past and future communication that uses the same key.

Since $2^{|\boldsymbol{p}|+|\boldsymbol{k}|} > 2^{|\boldsymbol{c}|}$, there are several key–plaintext pairs that yield the given ciphertext (assuming uniform distributions). Thus, the attacker will need to look at more than one ciphertext to uniquely identify the key and plaintext.

A ciphertext-only attack would have to use some information about the source, e.g., that the plaintexts consist of eight-bit blocks representing 7-bit ASCII codes so that some bits are known to be 0.

4.1.2 KNOWN-PLAINTEXT ATTACKS

By giving the attacker access to more and more information, their task is in some sense made easier, but also less realistic. A scenario that is still very realistic is the known-plaintext attack: given a plaintext and the corresponding ciphertext, the attacker learns the key. If $|\boldsymbol{k}| > |\boldsymbol{c}|$, several keys map the plaintext to the ciphertext, and the attack will need to observe more than one plaintext–ciphertext pair to uniquely identify the key.

4.1.3 CHOSEN-PLAINTEXT ATTACKS

Slightly more involved is an attack that not only *knows* the plaintexts, but *chooses* them. The chosen-plaintext attack makes some suitable choice of plaintexts (typically a set of plaintexts with some particular relation between them) and (somehow) acquires the encryption of them.

Even more involved is the *adaptive* chosen-plaintext attack, where some plaintexts are chosen depending on the ciphertexts obtained for the previously chosen plaintexts.

With each such modification, the attack scenario gets less and less practical, but typically, the attack requirements (time, data, memory) get more practical or at least less impractical. At any rate, such attacks, although perhaps entirely academical in their nature, provide some knowledge about the cryptographic primitive.

For stream ciphers, known-plaintext attacks translate to known-keystream attacks. Chosen-plaintext attacks do not apply, although chosen-IV attacks would be a close relative.

4.1.4 CHOSEN-CIPHERTEXT ATTACKS

In the chosen-ciphertext attack, the attack is able to request both encryptions and encryptions. Typically, after requesting ciphertexts corresponding to some chosen plaintexts, the attack proceeds by similarly requesting plaintexts corresponding to some chosen ciphertexts.

4.1.5 RELATED-KEY ATTACKS

Another kind of attack is the related-key attack [Knu93] [Bih94], where the attacker is able to (somehow) access not only the encryption $\boldsymbol{c}$ of $\boldsymbol{p}$ under the key $\boldsymbol{k}$, but also the encryption $\boldsymbol{c}'$ of $\boldsymbol{p}'$ under the key $\boldsymbol{k}'$. The plaintexts $\boldsymbol{p}$, $\boldsymbol{p}'$ might be the same, but the keys $\boldsymbol{k}$, $\boldsymbol{k}'$ are not. They are instead assumed to be *related* through some function f, i.e., $\boldsymbol{k}' = f(\boldsymbol{k})$. It can be argued that the attack is more realistic or plausible if the relation between the keys is in a sense »simple.« As an example, $\boldsymbol{k}' = \boldsymbol{k} + \boldsymbol{\Delta}$ for a constant $\boldsymbol{\Delta}$ might be more realistic than $\boldsymbol{k}' = f(\boldsymbol{k})$ for some heavily involved and nonlinear function f.

In particular, it might be possible to construct an efficient related-key, related-IV attack on, e.g., Grain by defining f as $f(\boldsymbol{k}, \boldsymbol{v}) = \phi_i^{-1}(\phi_i(\boldsymbol{k}, \boldsymbol{v}) + \Delta)$, where $\phi_i(\boldsymbol{k}, \boldsymbol{v})$ represents the initialization. However, the relation f depends on the secret key and the chances of ever knowingly acquiring data that fits the setting of the attack would be marginal. Also, it is not instantly clear what *knowledge* such an attack would provide about the cipher. (A related tongue-in-cheek paper [Rij10] by Rijmen is recommended.)

Whether a related-key attack can be carried out in practice could depend on how the primitive is used: if the key is burnt into the device, they seem theoretical; if some protocol surrounds the primitive, it might be possible to, perhaps probabilistically, apply the wanted relation f to data that is passed to the primitive.

In the end, related-key attacks are most useful as theoretical tools that improve the knowledge of the cipher. The recent cryptanalysis of AES [BKR11], although marginal, can be traced to entirely impractical related-subkey attacks which helped build an understanding of helpful ingredients in the attack.

4.1.6 DISTINGUISHING ATTACKS

Distinguishing attacks in essence try to identify some nonrandom property in the cryptographic primitive. They are commonly applied to stream ciphers, but may also be used with block ciphers run in counter mode or even in what is called known-key cryptanalysis [KR07]. A distinguisher is a decision rule, which takes a sequence of bits or blocks as input and decides whether it is most likely to have been generated by the cipher or if it looks random. Thus, it will output either $\mathbb{CIPHER}$ or $\mathbb{RANDOM}$. Exactly how the distinguisher (algorithm) is constructed will depend on the cryptographic primitive. If the distinguisher is able to produce the correct output more than half of the time it is considered successful, i.e., as a cryptanalytic result.

Whether the distinguisher is also practically useful is of course another thing. Distinguishers sometimes require huge amounts of data and/or time. If the required amount of time is more than $2^{|k|}$, it would be faster to brute force the key: if a key is found, the distinguisher outputs $\mathbb{CIPHER}$; otherwise $\mathbb{RANDOM}$. The NESSIE project considered all distinguishing attacks faster than brute force as valid attacks, prohibiting use of the stream cipher. This raised some concern [RH02] and the eSTREAM project took a more relaxed view, not ruling out primitives only due to distinguishers with huge time requirements [HJB08].

If a distinguisher can be found for a cipher, or a building block of the cipher, it can possibly be used to derive a key-recovery attack on the cipher. Englund and Johansson [EJ05] give a distinguisher for one of the two registers in the stream cipher LILI-128 [SDGM01] and guess the content of the other register. Using the distinguisher, one can distinguish between correct guesses and wrong guesses. Similar work [HJ07] [EHJ07] has been performed on the Pomaranch stream cipher [JHK05].

Another example of distinguishers relates to a scenario where a large data object (an image, a video, etc.) is transferred on the stream cipher-encrypted channel. If the attacker can observe the ciphertext $\boldsymbol{c}$ and knows *a priori* that one of two possible plaintexts $\boldsymbol{p}_1, \boldsymbol{p}_2$ is transferred, they can determine the two possible keystreams, $z_1 = \boldsymbol{c} + \boldsymbol{p}_1$ and $z_2 = \boldsymbol{c} + \boldsymbol{p}_2$. Using the distinguisher they are able to determine which looks more like the »typical« keystream. They then know, probabilistically, which of the two plaintexts was transferred.

4.2 GENERIC ATTACKS

All attacks need to be pitted against the corresponding generic attack to determine whether they are attacks at all and to what extend they »beat« the generic attack. For example, the generic known-plaintext attack is the brute force attack: the key is found in time at most $2^{|\boldsymbol{k}|}$, and expected $2^{|\boldsymbol{k}|-1}$, by exhaustively trying all keys. Here, the time has to be understood as $2^{|\boldsymbol{k}|}$ calls to the cryptographic primitive. Also, it should be noted that with $|\boldsymbol{k}| > |\boldsymbol{c}|$, a unique key can not be identified by a single trial encryption, so the time requirement will be $2^{|\boldsymbol{k}|} + 2^{|\boldsymbol{k}|-|\boldsymbol{c}|} + 2^{|\boldsymbol{k}|-2\cdot|\boldsymbol{c}|} + \ldots + 2^{|\boldsymbol{k}|-j\cdot|\boldsymbol{c}|}$, for some modest integer j. For all practical purposes, this can be considered as $2^{|\boldsymbol{k}|}$.

Any attack which manages to »beat« $2^{|\boldsymbol{k}|}$ is indeed an attack. If, on the other hand, an »attack« performs, e.g., $2^{\frac{2}{3}|\boldsymbol{k}|}$ steps, where each step can be roughly compared to $2^{\frac{2}{3}|\boldsymbol{k}|}$ calls to the cryptographic primitive, it is not an attack. It should be noted that accurately comparing the steps of the attack to »calls to the cryptographic primitive« for a fair comparison is not always straightforward.

4.3 ATTACK REQUIREMENTS

At least three requirements characterize an attack:

- time requirement,
- memory requirement, and
- data requirement.

It might also be useful to discuss run-time memory requirement, where an attacker is not able to use long-term storage for precomputed values, but is able to use short-term memory *during* the attack. Another example of an interesting attack characteristics is the precomputation time, which relates to the one-time computation required to, e.g., construct a look-up table or derive some special structures regarding a primitive.

As an example, the brute force attack requires time $2^{|k|}$, and small or negligible memory and data. The table-lookup attack, on the other hand, requires negligible time to find the key in a precomputed table of size $2^{|k|}$. The precomputation time is $2^{|k|}$. Time–memory trade-offs [Hel80] [Oec03] [BBS06] can be used to balance between these two extremes.

Data requirements describe how much data the attacker needs to be able to perform the attack. The data requirement for an attack on a stream cipher might be, e.g., the length required of one keystream, or the combined lengths of several different keystreams for different IVs and/or keys. If an attack uses some statistical property found in plaintext–ciphertext pairs, or in keystreams, the amount of data required depends on the »strength« of the exploited statistics (cf. Section 3.4).

4.4 ANALYSIS OF BOOLEAN FUNCTIONS

The Walsh transform, defined in Subsection 3.1.2, can be used to describe many properties of a Boolean function, as defined in Section 2.7.

Note that $\widehat{f}(\mathbf{0}) = \sum_{x\in\mathbb{F}_2^n}(-1)^{f(x)}$, so f is balanced if and only if $\widehat{f}(\mathbf{0}) = 0$.

Now, write

$$\widehat{f}(\boldsymbol{\omega}) = |\{x \in \mathbb{F}_2^n : f(x) = \langle x, \boldsymbol{\omega}\rangle\}| - |\{x \in \mathbb{F}_2^n : f(x) \neq \langle x, \boldsymbol{\omega}\rangle\}| \tag{4.1}$$

and

$$\mathbf{Pr}\left[f(x) = \langle \boldsymbol{\omega}, x\rangle\right] = \frac{\left|\{x \in \mathbb{F}_2^n : f(x) = \langle x, \boldsymbol{\omega}\rangle\}\right|}{2^n}.$$

From this follows that

$$\mathbf{Pr}\left[f(x) = \langle \boldsymbol{\omega}, x\rangle\right] = \frac{1}{2} + \frac{\widehat{f}(\boldsymbol{\omega})}{2^{n+1}}.$$

That is,

$$\mathbf{Pr}\left[f(x) = \langle \boldsymbol{\omega}, x\rangle\right] = \frac{1}{2} + \varepsilon_f(\boldsymbol{\omega}),$$

where

$$\varepsilon_f(\boldsymbol{\omega}) = \frac{\widehat{f}(\boldsymbol{\omega})}{2^{n+1}}$$

is referred to as the *bias* of the *linear approximation*

$$f(x) = \langle \boldsymbol{\omega}, x\rangle.$$

Recall that the nonlinearity $\mathrm{nl}(f)$ of f is a measure of to what extent it looks like an affine function. It can be seen that

$$\mathrm{nl}(f) = 2^{n-1} - \frac{1}{2}\max_{\boldsymbol{\omega}\in\mathbb{F}_2^n}\left|\widehat{f}(\boldsymbol{\omega})\right|.$$

Assume that $\max_{\omega \in \mathbb{F}_2^n} \left|\widehat{f}(\boldsymbol{\omega})\right|$ is achieved for $\boldsymbol{\omega} = \boldsymbol{\omega}_{\max}$. Then $\langle \boldsymbol{\omega}_{\max}, \boldsymbol{x} \rangle$ is a *best linear approximation* of f and produces output that is correlated to the output of the function f as

$$\mathbf{Pr}\left[f(\boldsymbol{x}) = \langle \boldsymbol{\omega}_{\max}, \boldsymbol{x} \rangle\right] = \frac{1}{2} + \varepsilon_f(\boldsymbol{\omega}_{\max}), \qquad \varepsilon_f(\boldsymbol{\omega}_{\max}) = \frac{1}{2} \pm \frac{\mathrm{nl}(f)}{2^n}.$$

In cryptanalysis, it is common that nonlinear building blocks, e.g., Boolean functions, are approximated by linear blocks and some added noise. The nonlinearity $\mathrm{nl}(f)$ is then a measure of how good the best approximation is. However, linear approximations other than the best could be more favorable to consider due to the cipher's internal structure. Typically, approximations with as few terms as possible might allow an attack that is much more successful than one that uses the best approximation.

Recall that if the least number of terms in a linear approximation with nonzero bias is $m+1$, then f is mth order correlation immune. Thus, a Boolean function is mth order correlation immune if and only if $\widehat{f}(\boldsymbol{\omega}) = 0$, for all $1 \leq w_{\mathrm{H}}(\boldsymbol{\omega}) \leq m$.

Finally, define the *correlation* $c_f(\boldsymbol{\omega}) = 2\varepsilon_f(\boldsymbol{\omega})$ of a linear approximation $f(\boldsymbol{x}) = \langle \boldsymbol{\omega}, \boldsymbol{x} \rangle$ as this will be useful in the sequel.

Example 4.1 Let $f(x_0, x_1, x_2) = 1 + x_0 + x_1 + x_2 + x_0x_1$. Then

$$(f(0,0,0), f(1,0,0), \ldots, f(1,1,1)) = (1,0,0,0,0,1,1,1).$$

It is straightforward to derive

$$\widehat{f}(\mathbf{0}) = (-1)^1 + (-1)^0 + (-1)^0 + (-1)^0 + (-1)^0 + (-1)^1 + (-1)^1 + (-1)^1 = 0,$$

so f is balanced. Similarly,

$$\begin{aligned}\widehat{f}(1,0,0) &= (-1)^{1+0} + (-1)^{0+1} + (-1)^{0+0} + (-1)^{0+1} \\ &\quad + (-1)^{0+0} + (-1)^{1+1} + (-1)^{1+0} + (-1)^{1+1} = 0,\end{aligned}$$

so $f(\boldsymbol{x}) = \langle (1,0,0), \boldsymbol{x} \rangle$ is a linear approximation with bias 0. However,

$$\begin{aligned}\widehat{f}(0,0,1) &= (-1)^{1+0} + (-1)^{0+0} + (-1)^{0+0} + (-1)^{0+0} \\ &\quad + (-1)^{0+1} + (-1)^{1+1} + (-1)^{1+1} + (-1)^{1+1} = 4,\end{aligned}$$

so $f(\boldsymbol{x}) = \langle (0,0,1), \boldsymbol{x} \rangle$ is a linear approximation with bias $\frac{\widehat{f}(0,0,1)}{2^4} = 2^{-2}$. The complete Walsh spectrum is

$$(\widehat{f}(0,0,0), \widehat{f}(1,0,0), \ldots, \widehat{f}(1,1,1)) = (0,0,0,0,4,-4,-4,-4), \tag{4.2}$$

so $\mathrm{nl}(f) = 2^2 - \frac{4}{2} = 2$ and there are four best linear approximations with $\left|\varepsilon_f(\boldsymbol{\omega})\right| = 2^{-2}$ and four unbiased ones. □

4.5 ANALYSIS OF VECTORIAL FUNCTIONS

The above analysis and notation can be extended to functions $F\colon \mathbb{F}_2^n \to \mathbb{F}_2^m$, to study *linear approximations*

$$\langle \boldsymbol{\beta}, F(\boldsymbol{x})\rangle = \langle \boldsymbol{\alpha}, \boldsymbol{x}\rangle,$$

where $\boldsymbol{\alpha} \in \mathbb{F}_2^n$ is referred to as the *input mask* and $\boldsymbol{\beta} \in \mathbb{F}_2^m$ as the *output mask*. Similar to above, the *bias* is defined through

$$\mathbf{Pr}\left[\langle \boldsymbol{\beta}, F(\boldsymbol{x})\rangle = \langle \boldsymbol{\alpha}, \boldsymbol{x}\rangle\right] = \frac{1}{2} + \varepsilon_F(\boldsymbol{\alpha}, \boldsymbol{\beta}),$$

and the *correlation* $c_F(\boldsymbol{\alpha}, \boldsymbol{\beta}) = 2\varepsilon_F(\boldsymbol{\alpha}, \boldsymbol{\beta})$. Then

$$\varepsilon_F(\boldsymbol{\alpha}, \boldsymbol{\beta}) = \frac{c_F(\boldsymbol{\alpha}, \boldsymbol{\beta})}{2} = \frac{\widehat{F}(\boldsymbol{\alpha}, \boldsymbol{\beta})}{2^{n+1}}.$$

From Proposition 3.1 follows that $\widehat{F}(\mathbf{0}, \mathbf{0}) = 2^n$, while $\widehat{F}(\boldsymbol{\alpha}, \mathbf{0}) = 0$ for $\boldsymbol{\alpha} \neq \mathbf{0}$. Further, if F is a permutation, $\widehat{F}(\mathbf{0}, \boldsymbol{\beta}) = 0$ for $\boldsymbol{\beta} \neq \mathbf{0}$.

Define the $2^m \times 2^n$ *correlation matrix* $C_F = (c_F(\boldsymbol{\alpha}, \boldsymbol{\beta}))_{\boldsymbol{\alpha}\in\mathbb{F}_2^n, \boldsymbol{\beta}\in\mathbb{F}_2^m}$ originally introduced by Daemen et al. [DGV95]. This should be understood as the matrix where the element at column $(\boldsymbol{\alpha})$ and row $(\boldsymbol{\beta})$ is $c_F(\boldsymbol{\alpha}, \boldsymbol{\beta})$.

Example 4.2 Consider the function $F\colon 2^3 \to 2^3$ such that

$$\begin{aligned} F(0,0,0) &= (1,0,0), & F(0,0,1) &= (1,1,1),\\ F(1,0,0) &= (0,1,0), & F(1,0,1) &= (1,1,0),\\ F(0,1,0) &= (0,0,0), & F(0,1,1) &= (0,0,1),\\ F(1,1,0) &= (1,0,1), & F(1,1,1) &= (0,1,1). \end{aligned}$$

All the nontrivial $\langle \boldsymbol{\alpha}, \boldsymbol{x}\rangle$ and $\langle \boldsymbol{\beta}, F(\boldsymbol{x})\rangle$ are given in Table 4.1, from which the correlation matrix can be derived: for each choice of $\boldsymbol{\alpha}, \boldsymbol{\beta}$, compare the two corresponding columns. The value at each position in the matrix (not considering the scaling factor) is the number of agreeing bits in the columns minus the number of disagreeing bits (cf. Equation 4.1). The correlation matrix is

$$2^{-3}\begin{array}{c} \\ {\scriptstyle 0}\\ {\scriptstyle 1}\\ {\scriptstyle 2}\\ {\scriptstyle 3}\\ {\scriptstyle 4}\\ {\scriptstyle 5}\\ {\scriptstyle 6}\\ {\scriptstyle 7} \end{array}\!\begin{array}{c} \begin{array}{cccccccc} {\scriptstyle 0} & {\scriptstyle 1} & {\scriptstyle 2} & {\scriptstyle 3} & {\scriptstyle 4} & {\scriptstyle 5} & {\scriptstyle 6} & {\scriptstyle 7} \end{array}\\ \begin{pmatrix} 8 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\ 0 & 0 & -4 & -4 & 0 & 0 & 4 & -4\\ 0 & 4 & -4 & 0 & 4 & 0 & 0 & 4\\ 0 & 4 & 0 & -4 & -4 & 0 & -4 & 0\\ 0 & 0 & 4 & -4 & 4 & 4 & 0 & 0\\ 0 & 0 & 0 & 0 & 4 & -4 & -4 & -4\\ 0 & 4 & 0 & 4 & 0 & 4 & 0 & -4\\ 0 & -4 & -4 & 0 & 0 & 4 & -4 & 0 \end{pmatrix} \end{array}, \tag{4.3}$$

	$(\boldsymbol{\alpha})$								$(\boldsymbol{\beta})$						
$\boldsymbol{x}$	1	2	3	4	5	6	7	$F(\boldsymbol{x})$	1	2	3	4	5	6	7
(0,0,0)	0	0	0	0	0	0	0	(1,0,0)	1	0	1	0	1	0	1
(1,0,0)	1	0	1	0	1	0	1	(0,1,0)	0	1	1	0	0	1	1
(0,1,0)	0	1	1	0	0	1	1	(0,0,0)	0	0	0	0	0	0	0
(1,1,0)	1	1	0	0	1	1	0	(1,0,1)	1	0	1	1	0	1	0
(0,0,1)	0	0	0	1	1	1	1	(1,1,1)	1	1	0	1	0	0	1
(1,0,1)	1	0	1	1	0	1	0	(1,1,0)	1	1	0	0	1	1	0
(0,1,1)	0	1	1	1	1	0	0	(0,0,1)	0	0	0	1	1	1	1
(1,1,1)	1	1	0	1	0	0	1	(0,1,1)	0	1	1	1	1	0	0

Table 4.1: All values of $\langle \boldsymbol{\alpha}, \boldsymbol{x} \rangle$ and $\langle \boldsymbol{\beta}, F(\boldsymbol{x}) \rangle$ for $\boldsymbol{\alpha}, \boldsymbol{\beta} \neq \mathbf{0}$, where F is the function in Example 4.3.

where column indices $(\boldsymbol{\alpha})$ and row indices $(\boldsymbol{\beta})$ have been written out.

Observe that for the component function $f'(\boldsymbol{x}) = \langle (1,0,1), F(\boldsymbol{x}) \rangle$,

$$(f'(0,0,0), f'(1,0,0), \ldots, f'(1,1,1)) = (1,0,0,0,0,1,1,1),$$

so f' is precisely the Boolean function f from Example 4.1. Thus, the correlation matrix row corresponding to $(\boldsymbol{\beta}) = 5$ is the Walsh transform of the function f, up to scaling (cf. Equation 4.2). □

Proposition 4.1 Let the correlation matrix C of size $2^n \times 2^n$ correspond to the invertible function $F\colon \mathbb{F}_2^n \to \mathbb{F}_2^n$. Then the following statements hold.

1. C^{T} is the correlation matrix of the function F^{-1}.
2. The identity matrix I of size $2^n \times 2^n$ is the correlation matrix of the identity function id: $\mathbb{F}_2^n \to \mathbb{F}_2^n$, $\mathrm{id}(\boldsymbol{x}) = \boldsymbol{x}$.
3. $C^{\mathrm{T}}C = CC^{\mathrm{T}} = \mathrm{I}$, i.e., C is an orthogonal matrix.
4. All eigenvalues λ_i of C lie on the unit circle, $|\lambda_i| = 1$.

Proposition 4.2 Consider $F_0\colon \mathbb{F}_2^n \to \mathbb{F}_2^k$, $F_1\colon \mathbb{F}_2^k \to \mathbb{F}_2^m$, the composite function

$$F = F_1 \circ F_0,$$

i.e., $F(\boldsymbol{x}) = F_1(F_0(\boldsymbol{x}))$, and the corresponding correlation matrices C_0, C_1, and C. Then

$$C = C_1 C_0.$$

Proof. It is clear that the two matrices C and C_1C_0 are of the same dimension $2^m \times 2^n$. The element at column $(\boldsymbol{\alpha})$ and row $(\boldsymbol{\beta})$ in C_1C_0 is

$$\begin{aligned}
c_{10}(\boldsymbol{\alpha},\boldsymbol{\beta}) &= \sum_{\boldsymbol{j}\in\mathbb{F}_2^k} c_{F_1}(\boldsymbol{j},\boldsymbol{\beta})c_{F_0}(\boldsymbol{\alpha},\boldsymbol{j}) \\
&= \sum_{\boldsymbol{j}\in\mathbb{F}_2^k} \left(2^{-k}\sum_{\boldsymbol{y}\in\mathbb{F}_2^k}(-1)^{\langle\boldsymbol{\beta},F_1(\boldsymbol{y})\rangle+\langle\boldsymbol{j},\boldsymbol{y}\rangle}\right)\left(2^{-n}\sum_{\boldsymbol{x}\in\mathbb{F}_2^n}(-1)^{\langle\boldsymbol{j},F_0(\boldsymbol{x})\rangle+\langle\boldsymbol{\alpha},\boldsymbol{x}\rangle}\right) \\
&= 2^{-k-n}\sum_{\boldsymbol{x},\boldsymbol{y}}(-1)^{\langle\boldsymbol{\beta},F_1(\boldsymbol{y})\rangle+\langle\boldsymbol{\alpha},\boldsymbol{x}\rangle}\sum_{\boldsymbol{j}}(-1)^{\langle\boldsymbol{j},\boldsymbol{y}+F_0(\boldsymbol{x})\rangle}.
\end{aligned}$$

By Proposition 3.1,

$$\begin{aligned}
c_{10}(\boldsymbol{\alpha},\boldsymbol{\beta}) &= 2^{-k-n}\sum_{\boldsymbol{x}\in\mathbb{F}_2^n}(-1)^{\langle\boldsymbol{\beta},F_1(F_0(\boldsymbol{x}))\rangle+\langle\boldsymbol{\alpha},\boldsymbol{x}\rangle}2^k \\
&= 2^{-n}\sum_{\boldsymbol{x}\in\mathbb{F}_2^n}(-1)^{\langle\boldsymbol{\beta},F(\boldsymbol{x})\rangle+\langle\boldsymbol{\alpha},\boldsymbol{x}\rangle},
\end{aligned}$$

which is precisely the correlation $c_F(\boldsymbol{\alpha},\boldsymbol{\beta})$. ■

The following follows from Proposition 3.1.

Proposition 4.3 For $\oplus_{\boldsymbol{v}}$, i.e., $\oplus_{\boldsymbol{v}}(\boldsymbol{x}) = \boldsymbol{x}+\boldsymbol{v}$, the correlation matrix $C_{\oplus_{\boldsymbol{v}}}$ is a diagonal matrix where the diagonal elements are $c_{\oplus_{\boldsymbol{v}}}(\boldsymbol{\alpha},\boldsymbol{\alpha}) = (-1)^{\boldsymbol{v}(\boldsymbol{\alpha})}$.

The following follows from

$$c_F(\boldsymbol{\alpha},\boldsymbol{\beta}) = \mathbf{Pr}\left[\langle\boldsymbol{\alpha},\boldsymbol{x}\rangle+\langle\boldsymbol{\beta},F(\boldsymbol{x})\rangle = 0\right] - \mathbf{Pr}\left[\langle\boldsymbol{\alpha},\boldsymbol{x}\rangle+\langle\boldsymbol{\beta},F(\boldsymbol{x})\rangle = 1\right].$$

Proposition 4.4 Assume that $\boldsymbol{\alpha} = \boldsymbol{\alpha}^0+\boldsymbol{\alpha}^1$, and that $\boldsymbol{\beta} = \boldsymbol{\beta}^0+\boldsymbol{\beta}^1$. Consider a vectorial function F such that $\langle\boldsymbol{\alpha}^0,\boldsymbol{x}\rangle+\langle\boldsymbol{\beta}^0,F(\boldsymbol{x})\rangle$ and $\langle\boldsymbol{\alpha}^1,\boldsymbol{x}\rangle+\langle\boldsymbol{\beta}^1,F(\boldsymbol{x})\rangle$ are statistically independent. Then

$$c_F(\boldsymbol{\alpha},\boldsymbol{\beta}) = c_F(\boldsymbol{\alpha}^0,\boldsymbol{\beta}^0)\cdot c_F(\boldsymbol{\alpha}^1,\boldsymbol{\beta}^1).$$

4.6 LINEAR DISTINGUISHING ATTACKS

A typical distinguisher on a stream cipher is constructed by finding some bias in the keystream. In the simplest case, some biased linear sum of keystream bits is identified [HJ11]. If

$$\mathbf{Pr}\left[\sum_i d_i z_i = 0\right] = \frac{1}{2}+\varepsilon,$$

and the bias is time-invariant, so that one can ensure $d_0 = 1$ and write

$$\mathbf{Pr}\left[\sum_i d_i z_{t+i} = 0\right] = \frac{1}{2} + \varepsilon, \quad t \geq 0,$$

the distinguisher requires about ε^{-2} samples to succeed, assuming independence (cf. Section 3.4). Since each sample is constructed from a window of length $m = \max_{i:\, d_i=1} i + 1$, the keystream must be of length at least $\varepsilon^{-2} + m$ for the distinguishing attack to succeed.

Braeken and Lano [BL05] considered distinguishing attacks on combiners and filter generators (cf. Sections 2.10 and 2.11). They found that combiners with a state of 256 bits need to use Boolean functions taking more than 36 bits of input and be optimally chosen. Such functions appear to be very hard to find. For filter generators, the number of input bits is at least 30, but they note that this assumes that the attacker is able to access very long keystreams. By limiting the keystreams to 2^{40} bits, they find that secure filter generators can be constructed using Boolean functions on 14 bits.

4.7 CORRELATION ATTACKS

Siegenthaler showed how the combiner could be attacked in the ciphertext-only setting; in this section, a known-plaintext, i.e., known-keystream, setting is assumed.

The attack focuses on the output function f: if some *correlation* can be identified between one of the input bits u_i^j and the output z_i, the key can be found faster than brute force, which requires that the combiner is run at most $\prod_{j=1}^{m} 2^{n_j}$ times.

As the key is precisely the initial state of the combiner, which in turn is precisely the initial state of all the LFSRs, the idea is to obtain LFSR initial states individually. By Kerckhoffs' principle, only the initial states are unknown to the attacker, i.e., the structure of the combiner is known, including the configurations of the LFSRs and the output function f.

Assume that there is a correlation with the jth LFSR, i.e.,

$$\mathbf{Pr}\left[z_i = u_i^j\right] = \frac{1}{2} + \varepsilon, \quad \varepsilon \neq 0,$$

and that the attacker has observed n bits of keystream, $\boldsymbol{z}$. The attacker searches through the space of initial states for the jth LFSR, i.e., tries all $\boldsymbol{v}^0 \in \mathbb{F}_2^{n_j}$. For each initial state, the sequence $\boldsymbol{v}$ is produced simply by clocking the LFSR. The attacker can easily check in how many positions the keystream agrees with each such sequence, i.e., they can calculate the numbers $N_{\boldsymbol{v}^0} = n - d_{\mathrm{H}}(\boldsymbol{z}, \boldsymbol{v})$.

For a wrong guess, the attacker expects $N_{v^0} \approx \frac{n}{2}$, but for the correct guess, they expect $N_{v^0} \approx \frac{n}{2} + \varepsilon n$.

Once the initial state of the jth LFSR has been obtained, the attacker can try to recover other LFSR initial states in the same way. Thus, if some correlation can be found for each individual LFSR, the complete key will be recovered in at most $\sum_{j=1}^{m} 2^{n_j}$ LFSR sequence generations.

Once some of the inputs to f are known, it might be possible to obtain better correlations for the other registers, e.g., when the jth LFSR has been recovered, the attacker knows when each of the two functions

$$f_0(\boldsymbol{x}) = f(x_0, \ldots, x_{j-1}, 0, x_{j+1}, \ldots, x_{m-1})$$

and

$$f_1(\boldsymbol{x}) = f(x_0, \ldots, x_{j-1}, 1, x_{j+1}, \ldots, x_{m-1})$$

is being used to produce the keystream, and might be able to derive stronger correlations for these two functions than for f. As a special case of this, it might also happen that the knowledge of $\boldsymbol{u}_j^0$ reduces the search space for $\boldsymbol{u}_{j'}^0$, $j' \neq j$.

It was the correlation attack that led Siegenthaler to define the notion of correlation immunity (cf. page 14). From it follows that a correlation attack on a combiner with an mth order correlation immune output function will have to consider (at least) $m + 1$ LFSRs simultaneously.

4.8 FAST CORRELATION ATTACKS

The attack cost of the correlation attack is exponential in the size of the smallest LFSR with a correlation. Thus, one straightforward countermeasure is to select large LFSR sizes n_i.

For implementation purposes, it is tempting to design such a large LFSR so that it uses a small number t of taps. Meier and Staffelbach considered keystream-correlated LFSRs with large L and small t, and showed how they could recover the (initial) state of the LFSR. Their attack is called the *fast correlation attack* and the underlying idea is to exploit the parity check equations that can be produced from the characteristic polynomial of the LFSR.

4.9 MEET-IN-THE-MIDDLE ATTACKS

The meet-in-the-middle attack [DH77] was first described by Diffie and Hellman. Assume that an R-round block cipher $F_{0,R}$ can be split as $F_{0,R} = F_{r,R} \circ F_{0,r}$ so that $F_{0,r}$ depends only on j key bits and $F_{r,R}$ depends only on $|\boldsymbol{k}| - j$ key bits, i.e., no key bit is used in both $F_{0,r}$ and $F_{r,R}$. Then an exhaustive search, using the plaintext–ciphertext pair $\boldsymbol{p}, \boldsymbol{c}$, can be performed by computing the 2^j possible values of $F_{0,r}(\boldsymbol{p})$ and the $2^{|\boldsymbol{k}|-j}$ possible values of $F_{r,R}^{-1}(\boldsymbol{c})$, and finding collisions between these two lists. This attack requires, e.g., time about $\max(2^j, 2^{|\boldsymbol{k}|-j})$ and memory about $\min(2^j, 2^{|\boldsymbol{k}|-j})$.

Diffie and Hellman originally considered this attack in the context of DES, which uses keys of 56 bits. A naive approach to doubling the key size to the much more secure 112 bits would be to apply two DES encryptions in turn, using a key $\boldsymbol{k} = \boldsymbol{k}^0 || \boldsymbol{k}^1$. The meet-in-the-middle reasoning above suggested that such a scheme would not provide 112 bits of security. Due to this, Triple DES [NIS12], also known as 3DES, has been recommended by NIST for either a 168-bit key $\boldsymbol{k} = \boldsymbol{k}^0 || \boldsymbol{k}^1 || \boldsymbol{k}^2$, or a 112-bit key where $\boldsymbol{k}^0 = \boldsymbol{k}^2$ to prevent the meet-in-the-middle attack.

4.10 (TRUNCATED) DIFFERENTIAL CRYPTANALYSIS

Differential cryptanalysis was publicly introduced by Biham and Shamir [BS93] in 1990. The idea is to study how a difference in the state propagates through the state during the encryption.

Assume that the attacker has found a differential on $R-1$ rounds of a key-alternating block cipher which always holds, i.e., for some $\Delta \boldsymbol{p}$, the state $R-1$ rounds into encryption, $\boldsymbol{s} = F_{0,R-1}(\boldsymbol{p})$, exhibits a difference $\Delta \boldsymbol{s}$:

$$\boldsymbol{p}^0 + \boldsymbol{p}^1 = \Delta \boldsymbol{p} \quad \Rightarrow \quad F_{0,R-1}(\boldsymbol{p}^0) + F_{0,R-1}(\boldsymbol{p}^1) = \Delta \boldsymbol{s}$$

for all $\boldsymbol{p}^0$ and all keys. Assume that the attacker has access to $\boldsymbol{c}^0$ and $\boldsymbol{c}^1$ corresponding to known (or chosen) plaintexts $\boldsymbol{p}^0$ and $\boldsymbol{p}^1$ such that $\boldsymbol{p}^0 + \boldsymbol{p}^1 = \Delta \boldsymbol{p}$. The attacker guesses the last round key $\boldsymbol{k}^R$ and performs partial decryptions of the two ciphertexts $\boldsymbol{c}^0$ and $\boldsymbol{c}^1$: $\boldsymbol{s}^0 = F_{R-1,R}^{-1}(\boldsymbol{c}^0)$, $\boldsymbol{s}^1 = F_{R-1,R}^{-1}(\boldsymbol{c}^1)$. If $\boldsymbol{s}^0 + \boldsymbol{s}^1 = \Delta \boldsymbol{s}$, the key guess is viable, but if $\boldsymbol{s}^0 + \boldsymbol{s}^1 \neq \Delta \boldsymbol{s}$, the guess is wrong. In this way, the attacker can reduce the number of possible round keys from $2^{|\boldsymbol{p}|}$ to some number m, where hopefully $m \ll 2^{|\boldsymbol{p}|}$.

The above is heavily simplified, but enough for this dissertation: most differentials used in cryptanalysis do not hold with probability 1, but with probability $\frac{1}{2} + \varepsilon$ for some $-\frac{1}{2} < \varepsilon < \frac{1}{2}$, or even probability 0, in which case the attack strategy is slightly different [Knu98]. Further, one might hope to guess less than the entire last round key, or parts of several of the last round keys to attack a differential on $R-j$ rounds, where $j > 1$.

4.10.1 TRUNCATED DIFFERENTIALS

Knudsen [Knu95] extended the technique to truncated differentials, where one does not study the entire state, but only, e.g., a few bits or only care about whether some larger word is zero or nonzero.

4.10.2 TRUNCATED RELATED-KEY DIFFERENTIALS

In this dissertation, truncated differentials will be considered in the related-key setting. In [BR10], a differential is denoted by $(\Delta\boldsymbol{p}, \Delta\boldsymbol{k}) \to \Delta\boldsymbol{s}$, where a difference in the plaintext and key gives a difference in the state some number of rounds into the encryption. This dissertation adopts and extends this notation. To denote truncated differentials, i.e., differentials where one only studies the differences in certain bit positions, a mask and a value will be used and written in hexadecimal using [mask : value]. As an example, [00010a00:00010800] denotes a differential in a 32-bit state concerning three bits: in two bits, there is bit-inequality, while in one bit, there is bit-equality. For the other bits, the difference is not known or not interesting. In pseudo-C code, such a mask-value pair could be used to identify a match by

`if ( (`$\boldsymbol{s}^0$`^`$\boldsymbol{s}^1$`)&mask) == 0 ) { ... }`.

In this dissertation, $\Delta\boldsymbol{k}$ always involves only a single bit, so this bit will be mentioned specifically, e.g., as in $(\mathbf{0}, k_{32}) \to$ [08075080 : 00000080], where the two related keys differ precisely in k_{32}, and there is no difference in the plaintext. In fact, in this dissertation, only $\Delta\boldsymbol{p} = \mathbf{0}$ is used. However, by always writing $(\Delta\boldsymbol{p}, \Delta\boldsymbol{k})$, it will be very clear that related-key differentials are used to give related-key attacks.

4.11 LINEAR CRYPTANALYSIS

Linear cryptanalysis originally appeared around the same time as differential cryptanalysis, and also primarily applies to block ciphers. The first linear cryptanalysis was by Matsui [Mat94b] on DES (see also the work by Tardy-Corfdir and Gilbert [TCG92]). Similar to above, the attacker guesses some round key material in the first and/or last rounds of the cipher and tries to observe some statistical behavior »inside« the cipher. In this case, the attacker uses linear approximations as introduced in Section 4.5, i.e.,

$$\langle \boldsymbol{\beta}, F(\boldsymbol{x}) \rangle = \langle \boldsymbol{\alpha}, \boldsymbol{x} \rangle,$$

for some $\boldsymbol{\alpha}, \boldsymbol{\beta} \in \mathbb{F}_2^{|\boldsymbol{p}|}$.

Consider for simplicity the full block cipher F, i.e., keep the first and/or last few rounds removed earlier. The idea is to use a relatively large bias $\varepsilon_F(\boldsymbol{\alpha}, \boldsymbol{\beta})$ so that it can be observed using about $\varepsilon_F^{-2}(\boldsymbol{\alpha}, \boldsymbol{\beta})$ samples (cf. Section 3.4), where one sample might refer to one plaintext–ciphertext pair, and it is assumed that all plaintext–ciphertext pairs are statistically independent.

Split F as $F = F_{R-1} \circ F_{R-2} \circ \ldots \circ F_0$ and assume that $c_{F_r}(\boldsymbol{\alpha}^r, \boldsymbol{\beta}^r)$ have been identified as the correlations for some input and output masks $\boldsymbol{\alpha}^r$, $\boldsymbol{\beta}^r$ for the functions F_r, $0 \leq r < R$. Further, assume that $\boldsymbol{\alpha} = \boldsymbol{\alpha}^0$, $\boldsymbol{\beta} = \boldsymbol{\beta}^{R-1}$, and

$\boldsymbol{\alpha}^{r+1} = \boldsymbol{\beta}^r$, $0 \leq r < R-1$, so that the input and output masks are chained from $\boldsymbol{\alpha}$ through $\boldsymbol{\beta}$. Then, assuming independence,

$$c_F(\boldsymbol{\alpha}, \boldsymbol{\beta}) = \prod_{r=0}^{R-1} c_{F_r}(\boldsymbol{\alpha}^r, \boldsymbol{\beta}^r) \tag{4.4}$$

through what is commonly referred to as the piling-up lemma.

It is clear that independence does not hold in general, but since it might be straightforward to derive all possible correlations $c_{F_r}(\boldsymbol{\alpha}^r, \boldsymbol{\beta}^r)$ using the vectorial Walsh transform on the nonlinear building blocks of the round functions, e.g., a relatively small Sbox, while it is less practical to investigate the full cipher at once, this simple analysis suggests one way of finding linear characteristics with large biases. For each round, derive a set of best linear approximations. Then chain them together and hope that the true correlation $c_F(\boldsymbol{\alpha}, \boldsymbol{\beta})$ does not differ too much from the value suggested by the piling-up lemma.

In the sequel, it is useful to distinguish between a linear approximation (or *linear characteristic*) and a *linear trail*. The basic concept of a linear trail has already been introduced above: it represents one particular way of reaching $\langle \boldsymbol{\beta}, F(\boldsymbol{x}) \rangle$ from $\langle \boldsymbol{\alpha}, \boldsymbol{x} \rangle$. More formally, a trail $\boldsymbol{\theta}$ from $\boldsymbol{\alpha}$ to $\boldsymbol{\beta}$ for an R-round block cipher F is a vector

$$\boldsymbol{\theta} = \boldsymbol{\theta}^0 || \boldsymbol{\theta}^1 || \ldots || \boldsymbol{\theta}^{R-1} || \boldsymbol{\theta}^R,$$

collecting $\boldsymbol{\theta}^0 = \boldsymbol{\alpha}$, $\boldsymbol{\theta}^R = \boldsymbol{\beta}$ and $R-1$ intermediate state masks. It is no restriction to assume that all $\boldsymbol{\theta}^r \neq \mathbf{0}$. By defining the *correlation* of a trail as

$$C_{\boldsymbol{\theta}} = \prod_{r=0}^{R-1} c_{F_r}(\boldsymbol{\theta}^r, \boldsymbol{\theta}^{r+1}) \tag{4.5}$$

(cf. Equation 4.4), it can be seen, e.g., using correlation matrices, that

$$c_F(\boldsymbol{\alpha}, \boldsymbol{\beta}) = \sum_{\boldsymbol{\theta}\colon \boldsymbol{\theta}^0 = \boldsymbol{\alpha}, \boldsymbol{\theta}^R = \boldsymbol{\beta}} C_{\boldsymbol{\theta}}. \tag{4.6}$$

In order to consider key-alternating block ciphers as introduced in Section 2.13, i.e.,

$$F = \oplus_{\boldsymbol{k}^R} \circ F'_{R-1} \circ \oplus_{\boldsymbol{k}^{R-1}} \circ F'_{R-2} \circ \ldots \circ \oplus_{\boldsymbol{k}^2} \circ F'_1 \circ \oplus_{\boldsymbol{k}^1} \circ F'_0 \circ \oplus_{\boldsymbol{k}^0},$$

cf. Figure 4.1, it is helpful to consider the expanded key

$$E(\boldsymbol{k}) = \boldsymbol{k}^0 || \boldsymbol{k}^1 || \ldots || \boldsymbol{k}^{R-1} || \boldsymbol{k}^R.$$

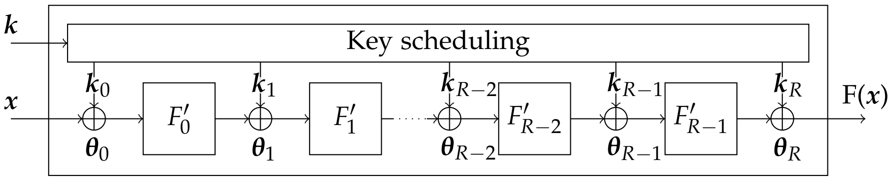


Figure 4.1: A key-alternating block cipher with a trail, cf. Figure 2.4.

As a trail through the cipher F is followed, key bits are collected, each time keeping or flipping the sign of the correlation depending on the key (cf. Proposition 4.3). That is,

$$C_{\boldsymbol{\theta}} = (-1)^{\langle \boldsymbol{\theta}, E(\boldsymbol{k}) \rangle} \prod_r c_{F'_r}(\boldsymbol{\theta}^r, \boldsymbol{\theta}^{r+1}) = (-1)^{\langle \boldsymbol{\theta}, E(\boldsymbol{k}) \rangle} C^{\mathbf{0}}_{\boldsymbol{\theta}},$$

where

$$C^{\mathbf{0}}_{\boldsymbol{\theta}} = \prod_r c_{F'_r}(\boldsymbol{\theta}^r, \boldsymbol{\theta}^{r+1})$$

is the correlation of the trail $\boldsymbol{\theta}$ when the xor key is the all-zero key (cf. Equation 4.5). Combining this with Equation 4.6 yields

$$c_F(\boldsymbol{\alpha}, \boldsymbol{\beta}) = \sum_{\boldsymbol{\theta}\colon \boldsymbol{\theta}^0 = \boldsymbol{\alpha}, \boldsymbol{\theta}^R = \boldsymbol{\beta}} (-1)^{\langle \boldsymbol{\theta}, E(\boldsymbol{k}) \rangle} C^{\mathbf{0}}_{\boldsymbol{\theta}}, \tag{4.7}$$

which is commonly referred to as the *linear hull equation*, and states that the correlation of a linear approximation is the sum of the correlations of the linear trails, where each term in the sum gets its sign depending on the key. That is, the exact value of the correlation depends on the particular key used, suggesting that while the correlation of a linear approximation might be small (large) on average, for some weak (strong) keys the key-dependent sum might add up to something relatively large (small). Linear hulls have been studied since [Nyb95]. See [Lea11] for a more detailed discussion.

The wide trail design strategy [DR01] is one particular way of trying to achieve resistance against linear cryptanalysis in an SPN: the linear layer is chosen so that all trails involve a »large« number $w_{\mathrm{H}}(\boldsymbol{\theta})$ of bits of the progressing state. In particular, if the linear layer is a bit permutation, the cipher *does not* follow the wide trail design strategy.

Finally, note that cryptanalysis based on zero-bias approximations has been proposed [BR11], but that this dissertation exclusively considers large correlations as good for attackers and bad for designers.

4.12 INSERTION AND SUBSTITUTION ATTACKS ON AUTHENTICATION SCHEMES

Recall the setting outlined in Section 2.6 and consider a message $\boldsymbol{m}$ and a tag $\boldsymbol{t} = f(\boldsymbol{k}_{\text{MAC}}, \boldsymbol{m})$ produced from the message $\boldsymbol{m}$ and the key $\boldsymbol{k}_{\text{MAC}}$.

Traditionally, two attack success probabilities are of interest: that of insertion (impersonation) attacks and that of substitution attacks. It has been shown that the substitution attack is always the more powerful attack [JKS94].

In the *insertion attack*, the attacker inserts data on the channel, i.e., submits $(\boldsymbol{m}, \boldsymbol{t})$ to the receiver. The message–tag pair will be accepted with probability at most

$$P_{\text{I}} = \max_{(\boldsymbol{m},\boldsymbol{t})} \mathbf{Pr}\left[(\boldsymbol{m}, \boldsymbol{t}) \text{ is valid}\right],$$

i.e.,

$$P_{\text{I}} = \max_{(\boldsymbol{m},\boldsymbol{t})} \mathbf{Pr}\left[f(\boldsymbol{k}_{\text{MAC}}, \boldsymbol{m}) = \boldsymbol{t}\right].$$

In a *substitution attack*, the attacker observes a valid message–tag pair $(\boldsymbol{m}, \boldsymbol{t})$ on the channel and replaces it with $(\boldsymbol{m}', \boldsymbol{t}')$, $\boldsymbol{m}' \neq \boldsymbol{m}$. This will succeed with probability at most

$$P_{\text{S}} = \max_{\substack{(\boldsymbol{m},\boldsymbol{t}),(\boldsymbol{m}',\boldsymbol{t}'),\\ \boldsymbol{m}\neq\boldsymbol{m}'}} \mathbf{Pr}\left[(\boldsymbol{m}', \boldsymbol{t}') \text{ is valid} \mid (\boldsymbol{m}, \boldsymbol{t}) \text{ is valid}\right],$$

i.e.,

$$P_{\text{S}} = \max_{\substack{(\boldsymbol{m},\boldsymbol{t}),(\boldsymbol{m}',\boldsymbol{t}'),\\ \boldsymbol{m}\neq\boldsymbol{m}'}} \mathbf{Pr}\left[f(\boldsymbol{k}_{\text{MAC}}, \boldsymbol{m}') = \boldsymbol{t}' \mid f(\boldsymbol{k}_{\text{MAC}}, \boldsymbol{m}) = \boldsymbol{t})\right].$$

Note that this definition is information-theoretical: it deals with a computationally unbounded attacker who can derive the full set of keys that map $\boldsymbol{m}$ to $\boldsymbol{t}$ and then choose a pair $(\boldsymbol{m}', \boldsymbol{t}')$ that matches the largest fraction of those keys.

Ideally, $P_{\text{I}} = P_{\text{S}} = 2^{-|\boldsymbol{t}|}$, so that an attacker's best strategy is to guess tags uniformly at random.

5 MACs Revisited and a New Lightweight Family of Universal Hash Functions

A new type of MAC construction is proposed, based on the framework of universal hash functions, Toeplitz matrices and ε-biased sample spaces. Some new theoretical results in this area are derived. In particular, some results by Krawczyk [Kra95] are corrected and extended. The new construction seems interesting to implement in hardware, compared to previous universal hash function based constructions. This class also seems promising with respect to security. The aforementioned results by Krawczyk gives a bound for the security of the construction and we demonstrate how this bound can be expected to be far from tight.

The hardware-attractive new constructions come at the price of not being able to prove the exact substitution probability. Instead, the expected probability is examined both through theoretical methods as well as through simulation. The arguments point at a sufficiently low substitution probability for the proposed constructions.

This chapter is organized as follows. In Section 5.1 we give some basic definitions. Section 5.2 then derives the theoretical background for the new constructions, generalizing some previous results related to the use of Toeplitz matrices for authentication. Section 5.3 proves some lemmas used in Section 5.2. In Section 5.4, we setup the particular problem we study, namely authentication used with a keystream generator, and in Section 5.5, we go through existing constructions and see how they fit the setting. In Section 5.6 we then give a new class of constructions and discuss its properties. Section 5.7 studies a specific instance, while Section 5.8 concludes the chapter.

5.1 UNIVERSAL HASH FUNCTION PRELIMINARIES

Authentication using an AXU hash function is done as follows. Split the key as $\boldsymbol{k}_{\text{MAC}} = \boldsymbol{k}_{\text{A}}||\boldsymbol{k}_{\text{B}}$, where $\boldsymbol{k}_{\text{B}}$ is of the same size as the tag and is referred to as the OTP. Assume that $H = \{h_i\}$ is ε-AXU. Then if the tag is generated as $\boldsymbol{t} = h_{\boldsymbol{k}_{\text{A}}}(\boldsymbol{m}) + \boldsymbol{k}_{\text{B}}$, we have $P_{\text{S}} \leq \varepsilon$. For the next message, we assume that $\boldsymbol{k}_{\text{MAC}}$ has received a new value.

Let S be a distribution on binary sequences $\boldsymbol{s} = (s_0, s_1, \ldots, s_{n-1})$ of length n. Then, *S passes the linear test $\boldsymbol{\alpha} \in \mathbb{F}_2^n$ with bias ε* if

$$\left| \frac{|\{\boldsymbol{s} \in S : \langle \boldsymbol{\alpha}, \boldsymbol{s} \rangle = 0\}|}{|S|} - \frac{1}{2} \right| \leq \varepsilon.$$

S is an *ε-biased distribution* if it passes all linear tests $\boldsymbol{\alpha} \neq \mathbf{0}$ with bias ε.

We consider S as a set of n-bit vectors, each one taken with the same probability $1/|S|$. To recapitulate the above, $\boldsymbol{s} = (s_0, s_1, \ldots, s_{n-1}) \in S$ is a random variable and so is $\langle \boldsymbol{\alpha}, \boldsymbol{s} \rangle \in \mathbb{F}_2$ for any $\boldsymbol{\alpha} \neq \mathbf{0}$. If all such binary random variables are deviating at most ε from the uniform binary distribution, we refer to S as an ε-biased distribution. Clearly, selecting S as all 2^n binary vectors would give us a 0-biased distribution. The idea is to make S smaller but still keep ε small.

5.2 THE GENERALIZED TOEPLITZ CONSTRUCTION—A BASIS FOR NEW CONSTRUCTIONS

Assume that $\boldsymbol{k} = (k_0, k_1, \ldots, k_{w+L-2})$ is a sequence of key bits chosen uniformly random from $\mathbb{F}_2^{w+L-1}$ and define from it the Toeplitz matrix

$$T_{\boldsymbol{k}} = \begin{pmatrix} k_0 & k_1 & \ldots & k_{w-1} \\ k_1 & k_2 & \ldots & k_w \\ k_2 & k_3 & \ldots & k_{w+1} \\ \vdots & \vdots & \ddots & \vdots \\ k_{L-1} & k_L & \ldots & k_{w+L-2} \end{pmatrix}.$$

Let $\boldsymbol{t} = (t_0, \ldots, t_{w-1})$ be a bitvector of length w and $\boldsymbol{m} = (m_0, \ldots, m_{L-1})$ a bitvector of length L. The tag is calculated as

$$\boldsymbol{t} = \boldsymbol{m} T_{\boldsymbol{k}}.$$

An algorithmic interpretation is that we initialize $\boldsymbol{t} \leftarrow \mathbf{0}$. Introduce a window of size w to form

$$\boldsymbol{K}_0 = (k_0, \ldots, k_{w-1}), \boldsymbol{K}_1 = (k_1, \ldots, k_w), \ldots, \boldsymbol{K}_{L-1} = (k_{L-1}, \ldots, k_{w+L-2}).$$

For each bit m_i, if it is zero we do nothing and if it is one we update $\boldsymbol{t} \leftarrow \boldsymbol{t} \oplus \boldsymbol{K}_i$. Note how we can use a register $\boldsymbol{K}$ to maintain a state which we shift and accumulate into the tag $\boldsymbol{t}$. It is a simple step to show that $P_{\mathrm{S}} = 2^{-w}$.

The Toeplitz matrix above uses $w + L - 1$ key bits $k_0, k_1, \ldots, k_{w+L-2}$, chosen uniformly at random, which is much too costly. The next step is to remove the requirement for genuine randomness and instead use some bitstream generator to decrease the key size.

Use $\boldsymbol{k}_{\mathrm{A}}$ to select a sequence $\boldsymbol{s} \in S$ from the ε-biased distribution, and associate with it the Toeplitz matrix

$$T_{\boldsymbol{s}} = \begin{pmatrix} s_0 & s_1 & \cdots & s_{w-1} \\ s_1 & s_2 & \cdots & s_w \\ s_2 & s_3 & \cdots & s_{w+1} \\ \vdots & \vdots & \ddots & \vdots \\ s_{L-1} & s_L & \cdots & s_{w+L-2} \end{pmatrix}.$$

Then use the xor value $\boldsymbol{k}_{\mathrm{B}}$ and $h_{\boldsymbol{k}_{\mathrm{A}}}(\boldsymbol{m}) = h_{\boldsymbol{s}}(\boldsymbol{m}) = \boldsymbol{m}T_{\boldsymbol{s}}$ to calculate the tag as

$$\boldsymbol{t} = h_{\boldsymbol{k}_{\mathrm{A}}}(\boldsymbol{m}) + \boldsymbol{k}_{\mathrm{B}} = h_{\boldsymbol{s}}(\boldsymbol{m}) + \boldsymbol{k}_{\mathrm{B}} = \boldsymbol{m}T_{\boldsymbol{s}} + \boldsymbol{k}_{\mathrm{B}}.$$

The following theorem [Kra95], inspired by [AGHP90] [NN93] [Kra94], relates the security of the construction to the nonrandomness of the sequence.

Theorem 5.1 A family of hash functions defined by a Toeplitz construction as above, where the bits $s_0, s_1, \ldots, s_{w+L-2}$ determine the Toeplitz matrix, is $(2^{-w} + 2\varepsilon)$-almost xor universal over equal-length strings if the distribution of the bit sequences $\boldsymbol{s}$ is ε-biased.

The theorem can actually be improved by changing the coefficient in front of ε from 2 to $2(1 - 2^{-w})$, but for large w this change is negligible. See Subsection 5.2.1 for a discussion on the proof of this theorem.

One might want to extend the security to messages of variable length (cf. Theorem 2 in [Sar08]):

Theorem 5.2 Let $A = \mathbb{F}_2^L$, $A' = \mathbb{F}_2^0 \cup \mathbb{F}_2^1 \cup \mathbb{F}_2^2 \cup \ldots \cup \mathbb{F}_2^{L-1}$, and $B = \mathbb{F}_2^w$. Assume that $H = \{h_i \mid i \in I\}$ is a set of hash functions h_i: $A \to B$. Define $H' = \{h'_i \mid i \in I\}$ by h'_i: $A' \to B$, $h'_i(\boldsymbol{m}) = h_i(\boldsymbol{m}||1||\boldsymbol{0}^{L-|\boldsymbol{m}|-1})$. If H is the $(2^{-w} + 2\varepsilon)$-AXU construction in Theorem 5.1, then H' is $(2^{-w} + 2\varepsilon)$-AXU.

The crucial property of the functions $h_i \in H$ is that appending zeros to the message does not change the tag. This property also allows h'_i to be calculated without invoking h_i on the zero-padded message.

5.2.1 PROVING THEOREM 5.1

For some $\boldsymbol{a}$ of length $n - w + 1$, let the $w \times n$ matrix A be constructed as

$$A = \begin{pmatrix} \boldsymbol{\alpha}^0 \\ \boldsymbol{\alpha}^1 \\ \vdots \\ \boldsymbol{\alpha}^{w-1} \end{pmatrix},$$

with

$$\boldsymbol{\alpha}^i = \mathbf{0}^{w-i-1}\boldsymbol{a}\mathbf{0}^i,$$

i.e., the rows of A are zero-padded shifted versions of $\boldsymbol{a}$. This matrix has full rank w if $\boldsymbol{a} \neq \mathbf{0}$, which will be assumed from now on. Define

$$\mathcal{R} = \{\gamma A \mid \gamma \in \mathbb{F}_2^w\},$$

which is the set of all 2^w linear combinations of the rows of A,

$$\mathcal{R}_{>0} = \{\gamma A \mid \gamma \in \mathbb{F}_2^w, \gamma \neq \mathbf{0}\} = \mathcal{R}\backslash\{\mathbf{0}\},$$

which is the set of all $2^w - 1$ nontrivial linear combinations of the rows of A, and

$$\mathcal{R}_{>1} = \{\gamma A \mid \gamma \in \mathbb{F}_2^w, w_{\mathrm{H}}(\gamma) > 1\},$$

which is the set of all $2^w - w - 1$ linear combinations of the rows of A that use at least two rows.

The proof of Theorem 5.1, as carried out in [Kra95], starts with noting that the success probability of replacing an $(n - w + 1)$-bit message and w-bit tag $(\boldsymbol{m}, \boldsymbol{t})$ with $(\boldsymbol{m} + \boldsymbol{a}, \boldsymbol{t} + \boldsymbol{b})$ is precisely the probability that $\boldsymbol{a}T_{\boldsymbol{s}} = \boldsymbol{b}$, with $\boldsymbol{s} \in S$ taken from the ε-biased distribution. This probability, further, is precisely the probability that $A\boldsymbol{s}^{\mathrm{T}} = \boldsymbol{b}^{\mathrm{T}}$. From above, A has full rank w since $\boldsymbol{a} \neq \mathbf{0}$.

The proof uses Fourier analysis and an indicator function

$$f_{\boldsymbol{\alpha},\beta}(\boldsymbol{s}) = \begin{cases} 1, & \langle \boldsymbol{\alpha}, \boldsymbol{s} \rangle = \beta, \\ 0, & \langle \boldsymbol{\alpha}, \boldsymbol{s} \rangle \neq \beta, \end{cases}$$

i.e.,

$$f_{\boldsymbol{\alpha},\beta}(\boldsymbol{s}) = \langle \boldsymbol{\alpha}, \boldsymbol{s} \rangle + \beta + 1.$$

A lemma then claims that $\mathrm{L}_1(f_{\boldsymbol{\alpha},\beta}) = 1$ for all vectors $\boldsymbol{\alpha}$ and bits β. The proof of this result is not included in [Kra95], but is stated to be quite straightforward. There are two different common ways of defining the Fourier transform of a Boolean function—this dissertation refers to them as the Fourier transform and the Walsh transform—and there is some confusion in [Kra95]: it can be seen that the result is not correct. It can however be corrected by requiring $(\boldsymbol{\alpha}, \beta) \neq (\mathbf{0}, 1)$, which is not a problem in this setting, where $\boldsymbol{\alpha} \neq \mathbf{0}$ follows from $\boldsymbol{a} \neq \mathbf{0}$:

Lemma 5.3 $\mathrm{L}_1(f_{\boldsymbol{\alpha},\beta}) = 1$ for all $(\boldsymbol{\alpha}, \beta) \neq (\mathbf{0}, 1)$.

More interestingly, we can strengthen Lemma 12 in [Kra95] from an inequality to an equality. That is, we can study the matrix–vector version of the indicator function,

$$f_{A,\boldsymbol{b}}(\boldsymbol{s}) = \begin{cases} 1, & A\boldsymbol{s}^{\mathrm{T}} = \boldsymbol{b}^{\mathrm{T}}, \\ 0, & A\boldsymbol{s}^{\mathrm{T}} \neq \boldsymbol{b}^{\mathrm{T}}, \end{cases}$$

and find the following.

Lemma 5.4 $\mathrm{L}_1(f_{A,\boldsymbol{b}}) = 1$ for all A of full rank.

The proof uses some valuable information on the Fourier coefficients, and since we will use these insights later, we formalize them.

Lemma 5.5 For $\boldsymbol{\omega} \in \mathcal{R}$, write $\boldsymbol{\omega} = \gamma A$ for some (unique) $\gamma \in \mathbb{F}_2^w$. Then

$$\mathcal{F}(f_{A,\boldsymbol{b}})(\boldsymbol{\omega}) = (-1)^{\langle \gamma, \boldsymbol{b} \rangle} 2^{n-w}.$$

For $\boldsymbol{\omega} \notin \mathcal{R}$, $\mathcal{F}(f_{A,\boldsymbol{b}})(\boldsymbol{\omega}) = 0$.

That is, $\mathcal{F}(f_{A,\boldsymbol{b}})(\boldsymbol{\omega}) = \pm 2^{n-w}$ for the 2^w values of $\boldsymbol{\omega} \in \mathcal{R}$ that are linear combinations of rows in the full-rank matrix A and zero for all other $\boldsymbol{\omega}$. The sign is positive precisely when the corresponding linear combination of bits in $\boldsymbol{b}$ is zero.

We prove these lemmas in Section 5.3. Theorem 5.1 now follows by the arguments in [Kra95], but we continue towards a modified, and arguably more detailed, proof here for completeness and some further insights.

First, let $\mu(\boldsymbol{s})$ be the distribution of sequences of length n and assume that there are 2^l distinct sequences, each occurring with probability 2^{-l}. Define $\varepsilon_{\boldsymbol{\omega}}$ as the bias ε for the linear test $\boldsymbol{\omega}$, but with the sign kept. That is,

$$|\{\boldsymbol{s} \in S : \langle \boldsymbol{\omega}, \boldsymbol{s} \rangle = 0\}| = 2^{l-1} + \varepsilon_{\boldsymbol{\omega}} 2^l.$$

We get

$$\begin{aligned} \mathcal{F}(\mu)(\boldsymbol{\omega}) &= \sum_{\boldsymbol{s} \in \mathbb{F}_2^n} \mu(\boldsymbol{s})(-1)^{\langle \boldsymbol{\omega}, \boldsymbol{s} \rangle} = \sum_{\boldsymbol{s} \in S} 2^{-l}(-1)^{\langle \boldsymbol{\omega}, \boldsymbol{s} \rangle} \\ &= 2^{-l}\left[(2^{l-1} + \varepsilon_{\boldsymbol{\omega}} 2^l)(+1) + (2^{l-1} - \varepsilon_{\boldsymbol{\omega}} 2^l)(-1)\right] = 2\varepsilon_{\boldsymbol{\omega}} \end{aligned}$$

for all $\boldsymbol{\omega} \neq \mathbf{0}$ (see also [KM93, pages 14–15]).

The following can be found in the proofs of Theorem 5 in [Kra95] and Lemma 4.5 in [KM93]:

Lemma 5.6

$$\left|\mathbf{Pr}_S\left[A\boldsymbol{s}^{\mathrm{T}} = \boldsymbol{b}^{\mathrm{T}}\right] - 2^{-w}\right| = 2^{-n}\left|\sum_{\boldsymbol{\omega}\neq\mathbf{0}} \mathcal{F}(\mu)(\boldsymbol{\omega})\mathcal{F}(f_{A,\boldsymbol{b}})(\boldsymbol{\omega})\right|$$

Proof (Theorem 5.1). By Lemma 5.5, $\mathcal{F}(f_{A,\boldsymbol{b}})(\boldsymbol{\omega}) = \pm 2^{n-w}$ for precisely $2^w - 1$ values of $\boldsymbol{\omega} \neq \mathbf{0}$ and 0 for all other $\boldsymbol{\omega} \neq \mathbf{0}$. Since $|\varepsilon_{\boldsymbol{\omega}}| \leq \varepsilon$ for all $\boldsymbol{\omega} \neq \mathbf{0}$, we get

$$\begin{aligned}\left|\mathbf{Pr}_S\left[A\boldsymbol{s}^{\mathrm{T}} = \boldsymbol{b}^{\mathrm{T}}\right] - 2^{-w}\right| &= 2^{-n}\left|\sum_{\boldsymbol{\omega}\neq\mathbf{0}} 2\varepsilon_{\boldsymbol{\omega}}\mathcal{F}(f_{A,\boldsymbol{b}})(\boldsymbol{\omega})\right| = 2\cdot 2^{-n}\left|\sum_{\boldsymbol{\omega}\in\mathcal{R}_{>0}} \varepsilon_{\boldsymbol{\omega}}2^{n-w}\right| \\ &= 2\cdot 2^{-w}\left|\sum_{\boldsymbol{\omega}\in\mathcal{R}_{>0}} \varepsilon_{\boldsymbol{\omega}}\right| \leq 2\cdot 2^{-w}(2^w - 1)\varepsilon < 2\varepsilon.\end{aligned}$$

■

5.2.2 INTERPRETING THE ABOVE TO FIND AN ATTACK

The above might suggest how to find a possible attack: For each $\boldsymbol{\omega} \neq \mathbf{0}$, find $\varepsilon_{\boldsymbol{\omega}}$. Then choose $\mathcal{F}(f)$ nonzero for the 2^w values of $\boldsymbol{\omega}$ that have the largest absolute values for $\varepsilon_{\boldsymbol{\omega}}$. By also matching signs, we can get all products $\varepsilon_{\boldsymbol{\omega}}\mathcal{F}(f)(\boldsymbol{\omega}) \geq 0$ (or $\leq$) and arrive at

$$\left|\mathbf{Pr}_S\left[A\boldsymbol{s}^{\mathrm{T}} = \boldsymbol{b}^{\mathrm{T}}\right] - 2^{-w}\right| = 2\left|\sum_{\boldsymbol{\omega}\neq\mathbf{0}} \varepsilon_{\boldsymbol{\omega}}\mathcal{F}(f)(\boldsymbol{\omega})\right| = 2\sum_{\boldsymbol{\omega}\neq\mathbf{0}} |\varepsilon_{\boldsymbol{\omega}}|\, 2^{-w}.$$

Having selected $\mathcal{F}(f)$, we can invert and acquire f. But this is probably not a realizable attack. There are $\binom{2^n-1}{2^w}2^{w-1}$ such transforms but only

$$(2^{n-w} - 1)2^w < 2^n$$

valid functions $f_{A,\boldsymbol{b}}$. Thus, it is highly unlikely that we have found a transform $\mathcal{F}(f)$ that can be inverted into $f_{A,\boldsymbol{b}}$ on our form, giving $\boldsymbol{a}, \boldsymbol{b}$. It would be interesting to see if it would be possible to devise an (efficient) algorithm for choosing large values of $\mathcal{F}(f)$ with matching signs and end up with the coefficients of a valid function.

A more practical insight gained from the above is that a good attack strategy might be to replace $(\boldsymbol{m}, \boldsymbol{t})$ with $(\boldsymbol{m} + \boldsymbol{a}, \boldsymbol{t} + \boldsymbol{b})$, where $\boldsymbol{a}$ is chosen as the linear test with the largest bias, i.e., $|\varepsilon_{\boldsymbol{a}}| = \max_{\boldsymbol{\omega}}(|\varepsilon_{\boldsymbol{\omega}}|)$. This will succeed with probability

$$\mathbf{Pr}_S\left[A\boldsymbol{s}^{\mathrm{T}} = \boldsymbol{b}^{\mathrm{T}}\right] \leq 2^{-w} + 2\cdot 2^{-w}\left|\sum_{\boldsymbol{\omega}\in\mathcal{R}_{>0}} \varepsilon_{\boldsymbol{\omega}}\right|.$$

By noting that the linear tests $\boldsymbol{a}$ and $\boldsymbol{a}||0$ have the same biases (the tests are formally applied to vectors of different lengths), it can be seen that $\varepsilon_{\boldsymbol{a}} = \varepsilon_{\boldsymbol{\alpha}^{w-1}}$.

Thus, the sum over $\boldsymbol{\omega}$ will contain $\varepsilon_{\boldsymbol{\alpha}^{w-1}} = \varepsilon_{\boldsymbol{a}}$, which can be expected to give a »large« contribution. Further, by assuming that $\varepsilon_{\boldsymbol{\alpha}^i} = \varepsilon_{\boldsymbol{\alpha}^{w-1}}$, $i = 0, \dots, w-2$, one arrives at

$$\begin{aligned}\mathbf{Pr}_S\left[A\boldsymbol{s}^{\mathrm{T}} = \boldsymbol{b}^{\mathrm{T}}\right] &\le 2^{-w} + 2 \cdot 2^{-w} \left| \sum_{\boldsymbol{\omega} \in \mathcal{R}_{>0}} \varepsilon_{\boldsymbol{\omega}} \right| \\ &\le 2^{-w} + 2 \cdot 2^{-w} \cdot w \cdot \varepsilon_{\boldsymbol{a}} + \left| \sum_{\boldsymbol{\omega} \in \mathcal{R}_{>1}} \varepsilon_{\boldsymbol{\omega}} \right|. \end{aligned} \tag{5.1}$$

In Section 5.7, we will study this probability closer for a specific design.

5.3 PROVING LEMMAS 5.3–5.5

Proof (Lemma 5.3). Assume $\boldsymbol{\alpha} = \mathbf{0}$ and $\beta = 0$. Then $f_{\boldsymbol{\alpha},\beta}(\boldsymbol{\omega}) = 1$ for all $\boldsymbol{\omega}$, so

$$\mathcal{F}(f_{\boldsymbol{\alpha},\beta})(\boldsymbol{\omega}) = \sum_{\boldsymbol{x} \in \mathbb{F}_2^n} f_{\boldsymbol{\alpha},\beta}(\boldsymbol{x})(-1)^{\langle \boldsymbol{\omega}, \boldsymbol{x} \rangle} = \sum_{\boldsymbol{x} \in \mathbb{F}_2^n} (-1)^{\langle \boldsymbol{\omega}, \boldsymbol{x} \rangle}.$$

By Proposition 3.1,

$$\mathcal{F}(f_{\boldsymbol{\alpha},\beta})(\boldsymbol{\omega}) = \begin{cases} 2^n, & \boldsymbol{x} = \mathbf{0}, \\ 0, & \boldsymbol{x} \neq \mathbf{0}, \end{cases}$$

so $\mathrm{L}_1(f_{\boldsymbol{\alpha},\beta}) = 2^{-n} \sum_{\boldsymbol{\omega}} \left|\mathcal{F}(f_{\boldsymbol{\alpha},\beta})(\boldsymbol{\omega})\right| = 1$.

For the remainder of the proof, assume that $\boldsymbol{\alpha} \neq \mathbf{0}$. We have

$$\mathcal{F}(f_{\boldsymbol{\alpha},\beta})(\boldsymbol{\omega}) = \sum_{\boldsymbol{x} \in \mathbb{F}_2^n} f_{\boldsymbol{\alpha},\beta}(\boldsymbol{x})(-1)^{\langle \boldsymbol{\omega}, \boldsymbol{x} \rangle} = \sum_{\boldsymbol{x}\colon f_{\boldsymbol{\alpha},\beta}(\boldsymbol{x})=1} (-1)^{\langle \boldsymbol{\omega}, \boldsymbol{x} \rangle}.$$

By Proposition 3.1, $\langle \boldsymbol{\alpha}, \boldsymbol{x} \rangle$ is a balanced function, so $f_{\boldsymbol{\alpha},\beta}(\boldsymbol{x})$ is a balanced function. That is, there are precisely 2^{n-1} different $\boldsymbol{x}$ for which $f_{\boldsymbol{\alpha},\beta}(\boldsymbol{x}) = 1$. We see that

$$\mathcal{F}(f_{\boldsymbol{\alpha},\beta})(\mathbf{0}) = \sum_{\boldsymbol{x}\colon f_{\boldsymbol{\alpha},\beta}(\boldsymbol{x})=1} (-1)^{\langle \mathbf{0}, \boldsymbol{x} \rangle} = 2^{n-1},$$

and similarly

$$\mathcal{F}(f_{\boldsymbol{\alpha},\beta})(\boldsymbol{\alpha}) = \sum_{\boldsymbol{x}\colon f_{\boldsymbol{\alpha},\beta}(\boldsymbol{x})=1} (-1)^{\langle \boldsymbol{\alpha}, \boldsymbol{x} \rangle} = \sum_{\boldsymbol{x}\colon f_{\boldsymbol{\alpha},\beta}(\boldsymbol{x})=1} (-1)^{\beta} = 2^{n-1}(-1)^{\beta}.$$

Assume that $\boldsymbol{\omega} \neq \mathbf{0}, \boldsymbol{\alpha}$. It remains to show that $\mathcal{F}(f_{\boldsymbol{\alpha},\beta})(\boldsymbol{\omega}) = 0$. It will then follow that $\mathrm{L}_1(f_{\boldsymbol{\alpha},\beta}) = 2^{-n} \sum_{\boldsymbol{\omega}} \left|\mathcal{F}(f_{\boldsymbol{\alpha},\beta})(\boldsymbol{\omega})\right| = 1$. Note that due to $\boldsymbol{\omega} \neq \mathbf{0}$,

$\langle \boldsymbol{\omega}, \boldsymbol{x} \rangle$ and $\langle \boldsymbol{\alpha}, \boldsymbol{x} \rangle$ are balanced, and that since $\boldsymbol{\omega} \neq \boldsymbol{\alpha}$, they are independent. Thus,

$$\begin{aligned}\mathcal{F}(f_{\boldsymbol{\alpha},\beta})(\boldsymbol{\omega}) &= \sum_{\boldsymbol{x}\in\mathbb{F}_2^n} f_{\boldsymbol{\alpha},\beta}(\boldsymbol{x})(-1)^{\langle \boldsymbol{\omega},\boldsymbol{x}\rangle} \\ &= 2^n \left(\mathbf{Pr}\left[\langle \boldsymbol{\alpha}, \boldsymbol{x}\rangle = \beta, \langle \boldsymbol{\omega}, \boldsymbol{x}\rangle = 0\right]\right. \\ &\quad \left.-\mathbf{Pr}\left[\langle \boldsymbol{\alpha}, \boldsymbol{x}\rangle = \beta, \langle \boldsymbol{\omega}, \boldsymbol{x}\rangle = 1\right]\right) \\ &= 2^n \left(\mathbf{Pr}\left[\langle \boldsymbol{\alpha}, \boldsymbol{x}\rangle = \beta\right] \mathbf{Pr}\left[\langle \boldsymbol{\omega}, \boldsymbol{x}\rangle = 0\right]\right. \\ &\quad \left.-\mathbf{Pr}\left[\langle \boldsymbol{\alpha}, \boldsymbol{x}\rangle = \beta\right] \mathbf{Pr}\left[\langle \boldsymbol{\omega}, \boldsymbol{x}\rangle = 1\right]\right) \\ &= 2^n \left(\frac{1}{2}\cdot\frac{1}{2} - \frac{1}{2}\cdot\frac{1}{2}\right) = 0.\end{aligned}$$

■

Lemma 5.4 follows directly from Lemma 5.5, so we focus on the latter. Let us find $\mathcal{F}(f_{A,\boldsymbol{b}})(\boldsymbol{\omega})$ for all $\boldsymbol{\omega}$. Similar to the proof of Lemma 5.3, we should consider precisely those vectors $\boldsymbol{x} \in S$ for which $f_{A,\boldsymbol{b}}(\boldsymbol{x}) = 1$. For these $\boldsymbol{x}$ it holds that $\langle \boldsymbol{\alpha}^i, \boldsymbol{x}\rangle = b_i$, $i = 0, 1, \ldots, w-1$.

Let $\boldsymbol{\omega} = \boldsymbol{\gamma} A = \sum \gamma_i \boldsymbol{\alpha}^i$ be a linear combination of the rows in A. Then

$$\begin{aligned}\mathcal{F}(f_{A,\boldsymbol{b}})(\boldsymbol{\omega}) &= \sum_{\boldsymbol{x}:\, f_{A,\boldsymbol{b}}(\boldsymbol{x})=1} (-1)^{\langle \boldsymbol{\omega},\boldsymbol{x}\rangle} = \sum_{\boldsymbol{x}:\, f_{A,\boldsymbol{b}}(\boldsymbol{x})=1} (-1)^{\langle \sum \gamma_i \boldsymbol{\alpha}^i,\boldsymbol{x}\rangle} \\ &= \sum_{\boldsymbol{x}:\, f_{A,\boldsymbol{b}}(\boldsymbol{x})=1} (-1)^{\sum \gamma_i \langle \boldsymbol{\alpha}^i,\boldsymbol{x}\rangle} = \sum_{\boldsymbol{x}:\, f_{A,\boldsymbol{b}}(\boldsymbol{x})=1} (-1)^{\sum \gamma_i b_i} \\ &= \sum_{\boldsymbol{x}:\, f_{A,\boldsymbol{b}}(\boldsymbol{x})=1} (-1)^{\langle \boldsymbol{\gamma},\boldsymbol{b}\rangle} = \left|\{\boldsymbol{x} :\ f_{A,\boldsymbol{b}}(\boldsymbol{x}) = 1\}\right| (-1)^{\langle \boldsymbol{\gamma},\boldsymbol{b}\rangle}.\end{aligned}$$

Further,

$$\begin{aligned}\left|\{\boldsymbol{x} :\ f_{A,\boldsymbol{b}}(\boldsymbol{x}) = 1\}\right| &= 2^n \mathbf{Pr}\left[\langle \boldsymbol{\alpha}^i, \boldsymbol{x}\rangle = \beta_i, \forall i\right] \\ &= 2^n \prod_i \mathbf{Pr}\left[\langle \boldsymbol{\alpha}^i, \boldsymbol{x}\rangle = \beta_i\right] = 2^n (2^{-1})^w,\end{aligned}$$

due to independence, so $\mathcal{F}(f_{A,\boldsymbol{b}})(\boldsymbol{\omega}) = 2^{n-w}(-1)^{\langle \boldsymbol{\gamma},\boldsymbol{b}\rangle}$.

A is of full rank w so all linear combinations $\boldsymbol{\gamma} A$ will be distinct. Thus, we have 2^w values of $\boldsymbol{\omega}$ for which $\mathcal{F}(f_{A,\boldsymbol{b}})(\boldsymbol{\omega}) = \pm 2^{n-w}$. If all other $\mathcal{F}(f_{A,\boldsymbol{b}})(\boldsymbol{\omega})$ are zero, we are done. Note that we can already claim that $\mathrm{L}_1(f_{A,\boldsymbol{b}}) \geq 1$ and that it is enough to show that $\mathrm{L}_1(f_{A,\boldsymbol{b}}) = 1$. By using the fact that

$$f_{A,\boldsymbol{b}}(\boldsymbol{x}) = \prod f_{\boldsymbol{\alpha}^i, b_i}(\boldsymbol{x}),$$

together with Proposition 3.3 and Lemma 5.3, we get

$$1 \leq \mathrm{L}_1(f_{A,\boldsymbol{b}}) = \mathrm{L}_1(\prod f_{\alpha^i,b_i}) \leq \prod \mathrm{L}_1(f_{\alpha^i,b_i}) = 1.$$

Thus, we have »sandwiched« $\mathrm{L}_1(f_{A,\boldsymbol{b}})$, and must have $\mathrm{L}_1(f_{A,\boldsymbol{b}}) = 1$. ■

5.4 USING AUTHENTICATION TOGETHER WITH A STREAM CIPHER

We are interested in a mechanism combining encryption and authentication in some packet-based communication system. We assume that the encryption is performed using a modern stream cipher, using a secret key $\boldsymbol{k}$ and a public initial value (IV). The stream cipher would either be a dedicated one, or a suitable mode of operation for a block cipher. The key $\boldsymbol{k}_{\mathrm{MAC}}$ used to produce the MAC is derived from $\boldsymbol{k}$ by using keystream before encryption starts. In the analysis below, $\boldsymbol{k}_{\mathrm{MAC}}$ is assumed to be chosen uniformly at random.

To appreciate why all of $\boldsymbol{k}_{\mathrm{MAC}}$ is extracted before encryption, consider a first approach that, after initializing the stream cipher, extracts whatever randomness the MAC construction needs to get started ($\boldsymbol{k}_{\mathrm{A}}$ above). It encrypts the message as usual and feeds the data into the authentication box. Finally, once the entire message has been processed, it extracts some additional randomness from the keystream to finalize the authentication tag ($\boldsymbol{k}_{\mathrm{B}}$ above). However, using bits from the »end« of the keystream in this manner is not a very good idea if the message can have variable size: with a slightly shorter message, the key material previously used for encryption, would now be used for authentication. The security implications would be disastrous—it is crucial that the sender and receiver use the same one-time pad. This has been noted independently in [FGRV10].

Our solution is to extract all the key bits in $\boldsymbol{k}_{\mathrm{MAC}}$ needed for the authentication before encrypting. This either means that we need to store some of these key bits somewhere until we need them, or we use a construction that solves this in another way, e.g., putting them in a processing state. The message is processed bit by bit and does not need to be stored. Summarizing this, we present an overview of this approach in Figure 5.1. In the remainder of this chapter, we will focus on the dashed rectangle in Figure 5.1 which inputs a key and a message and outputs a tag.

5.5 AN OVERVIEW OF PREVIOUS CONSTRUCTIONS

In this section, we will go through previous constructions of universal hash function-based MACs. We will ignore the process of deriving $\boldsymbol{k}_{\mathrm{MAC}}$. We consider the hardware cost for the MAC generation part, as marked in Figure 5.1. For some constructions, we have sketched a hardware implementation and

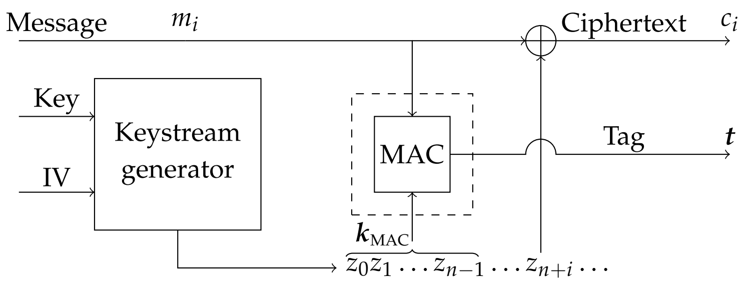


Figure 5.1: The setting we consider in this chapter. The first n output bits from a keystream generator, initialized with a key and IV, form $\boldsymbol{k}_{\text{MAC}}$ which is used to initialize the authentication mechanism. The rest of the keystream bits are used to encrypt the message bits m_i, $i = 0, 1, \dots$. The end result is a sequence of ciphertext bits c_i and the authentication tag $\boldsymbol{t}$.

Table 5.1: The gate counts used for different functions.

Function	Gate count
Flip flop	8
NAND2	1
XOR2	2.5

will mention the gate count obtained. The gate counts required for the ANDs, XORs, flip flops, etc. can be given as estimates at best. The exact cost of any implementation will depend on many parameters, such as the exact type of hardware used, the latest-and-greatest optimizations and tricks, and so on. Nonetheless, an estimate using some established measurements is highly useful in quickly assessing the feasibility of an algorithm.

We have used hardware cost figures found in several other papers, e.g., [HJM06]; they are found in Table 5.1. It should be noted that this hardware analysis is »simple:« it essentially counts the number of w-bit registers used and the number of gates used to process the contents of those registers.

Several different MAC algorithms have been considered; the MAC generation in GCM, the UIA2 algorithm in UMTS, the LH and UH constructions by Sarkar, Krawczyk's CRC construction, and the LFSR-based Toeplitz construction.

The last three constructions are all ε-AXU. Notably, they only update what will be the output using xor. Thus, the OTP can be added at the beginning just as well as at the end. We note that this allows us to save one register.

5.5.1 GCM: GMAC

In GMAC [MV04], the tag size is bounded as $w \geq 128$, although we will consider smaller w below. We have $\boldsymbol{k}_{\text{MAC}} = \boldsymbol{k}_{\text{A}}||\boldsymbol{k}_{\text{B}}$ as above. Ignoring what the specification calls »additional authenticated data«, the L-bit message $\boldsymbol{m}$ is split into w-bit blocks $\boldsymbol{m}^i$, $0 \leq i < n$ ($\boldsymbol{m}^{n-1}$ zero-padded if necessary). Set $\boldsymbol{m}^n = \boldsymbol{0}^{w/2}||\boldsymbol{L}$, where $(\boldsymbol{L}) = L$ and $|\boldsymbol{L}| = w/2$. As in Subsection 2.9.1, map the vectors $\boldsymbol{k}_{\text{A}}$ and $\boldsymbol{m}^i$ to elements $k_{\text{A}}(x)$ and $m^i(x)$ in some fixed representation of $\mathbb{F}_{2^w}$. Let

$$\text{GHASH}(\boldsymbol{k}_{\text{A}}, \boldsymbol{m}, \boldsymbol{m}^n) = \sum_{i=0}^{n} m^i(x) \cdot k_{\text{A}}(x)^{n+1-i}$$

be the internal keyed hash where all calculations are performed in the finite field. The resulting element in the finite field can then be mapped to a w-bit vector in a natural way. Then GHASH is $n_{\max}2^{-w}$-AXU where $n_{\max}$ is the maximum allowed message block count and w the length of the output. Thus, the tag is produced as $\text{GHASH}(\boldsymbol{k}_{\text{A}}, \boldsymbol{m}, \boldsymbol{m}^n) + \boldsymbol{k}_{\text{B}}$.

An implementation would consist of five w-bit registers as in Figure 5.2 where the next message block is loaded in parallel with processing the current one. The final xoring of random bits cannot be performed ahead of time, which means that a special register is needed for keeping this value until the very end of the process.

A rough calculation of the hardware cost gives $5w$ flip flops, w ANDs, $2w$ XORs and a small number of multiplexers, e.g., 1500 gates for $w = 32$. To save on one register, the message buffer could be removed at the cost of lowering the speed to half.

5.5.2 UMTS: UIA2

In UIA2, used in UMTS, there are large similarities with GCM, but also some differences [ETS09]; we ignore the differing block sizes in the original specifications, and the difference in length-encoding. Further, $\boldsymbol{k}_{\text{A}}$ needs to select two random values $k_1(x), k_2(x) \in \mathbb{F}_{2^{2w}}$. Then the tag is calculated as

$$\boldsymbol{t} = \text{trunc}_w(\text{GHASH}(\boldsymbol{k}_1, \boldsymbol{m}, \boldsymbol{m}^n) \cdot k_2(x)) + \boldsymbol{k}_{\text{B}}.$$

Note in particular the truncation by half the bits after the multiplication by $k_2(x)$. The use of k_2 cannot be moved to before the processing of the message, meaning that it must be stored somewhere during the entire process. Combined with the use of registers of size $2w$, the gate count is more than doubled compared to GMAC.

While GHASH is a $n_{\max}2^{-2w}$-AXU hash family, the final multiplication and truncation is 2^{-w}-AXU. This makes UIA2 $(n_{\max}2^{-2w} + 2^{-w})$-AXU. Comparing GMAC and UIA2, it seems one protects longer messages better at the cost of

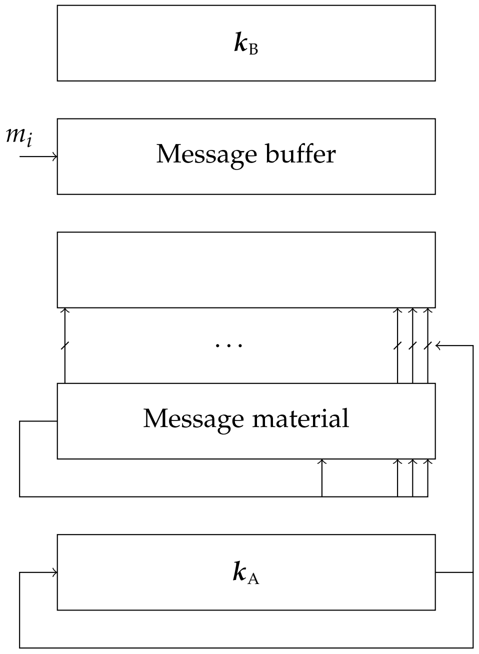


Figure 5.2: A rough hardware implementation of GHASH.

some additional postprocessing which unfortunately is costly in the setting considered here.

5.5.3 LH AND UH

Sarkar [Sar08] writes about a construction which they claim can be implemented very efficiently using a word-oriented LFSR. Furthermore, they suggest an application using a stream cipher to generate keystream, some of which is used for encryption and some of which is used for authentication.

Example 5.1 This is a simplified version of an example in [Sar08] where we avoid word-oriented LFRSs and simply use binary LFSRs. We use a full-period LFSR of length w and a w-bit tag accumulator and start by loading key material into the shift register. For each bit to authenticate, we either xor the accumulator with shift register content or we don't, depending on whether the authenticated bit is 1 or 0, respectively. At each step, we also clock the LFSR. After authenticating w bits, we load new key material and proceed in the same fashion. □

It should be noted that the keystream consumption will be very large and, in fact, the authentication will use just as much keystream as the encryp-

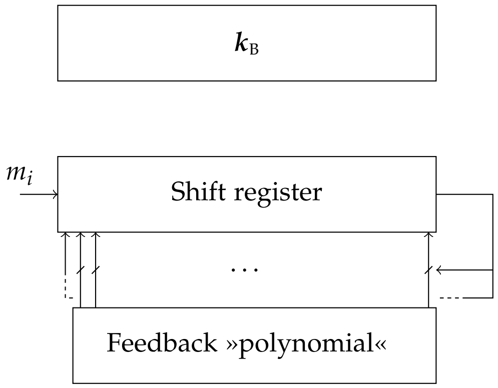


Figure 5.3: A rough hardware implementation of the CRC construction.

tion. Further, the same amount of key material usage can be achieved by the Toeplitz approach from Section 5.2, with the notable exception that it loads key bits one by one, instead of in large chunks every now and then.

5.5.4 CRYPTOGRAPHIC CRC

Krawczyk [Kra94] suggests using a CRC with a secret irreducible polynomial $g(x)$ and treating the message $\boldsymbol{m}$ as a polynomial $m(x)$ to calculate

$$h(x) = m(x) \cdot x^n \bmod g(x).$$

Considering the polynomial $h(x)$ as a bit-vector $\boldsymbol{h}$, the construction is AXU, so $\boldsymbol{t} = \boldsymbol{h} + \boldsymbol{k}_B$.

An implementation is outlined in Figure 5.3. The top register contains the OTP, added during finalization. Message bits are shifted into the middle register. When the bit shifted out of the middle register is a one, the contents of the lower register is added to the middle one and the leftmost bit in the middle register is flipped (this represents the »+1« in the feedback polynomial).

A rough hardware implementation yields $3w$ flip flops, w ANDs, w XORs and a small number of multiplexers, e.g., 900 gates for $w = 32$. One problem is that the feedback polynomial must be chosen randomly, yet it has to be irreducible. This is a major drawback.

5.5.5 AN LFSR-BASED TOEPLITZ CONSTRUCTION

Krawczyk also suggests using the Toeplitz construction in Section 5.2 and producing the sequence through an LFSR with a secret irreducible feedback polynomial and a secret initial state. The sequence it produces is ε-biased

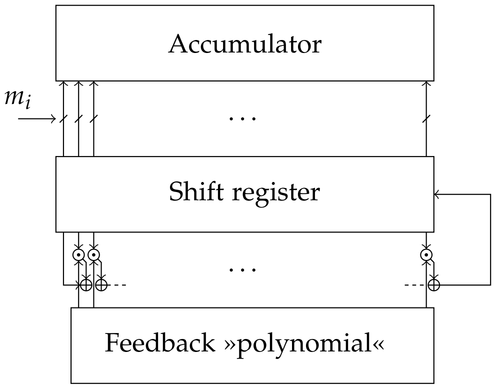


Figure 5.4: A rough hardware implementation of the LFSR-based construction.

with $\varepsilon = \frac{L_{\max}}{2^w}$ [AGHP90] [NN93], where $L_{\max}$ is the maximum allowed message length, and the construction is $2^{-w} + 2\varepsilon$-AXU (cf. Theorem 5.1). In fact, from [Kra94], it is known to be 2ε-AXU.

A rough hardware implementation is outlined in Figure 5.4. It requires $3w$ flip flops, $2w$ ANDs, $2w$ XORs, and some multiplexers, e.g., 1000 gates for $w = 32$.

Note the similarities with Example 5.1 above. In that construction, a higher security was maintained at the price of constantly refilling the state of the LFSR. The price then was keystream consumption; here, one needs to randomly choose the irreducible polynomial.

5.6 A NEW CLASS OF CONSTRUCTIONS

Some algorithms in the overview above require keystream slightly longer than the message, which does not work well in the setting considered here. Others use costly multiplications in $\mathbb{F}_{2^l}$, where l can be the tag size w. Only three registers are required for Krawczyk's CRC and LFSR-based constructions which are very implementation efficient, but where the key needs to select an irreducible polynomial. This is very costly to implement, either using a lookup table of known polynomials, or by testing whether generated polynomials are irreducible or not.

To conclude, the above analysis of existing universal hash function constructions has found that either 1) they consume too much keystream, 2) their hardware implementation is fairly resource consuming, or 3) we need »special randomness,« i.e., random irreducible polynomials.

In the remainder of this chapter, we will present and study a new construction, which creates an ε-biased stream, which is accumulated into a register. This allows the final one-time pad to be added before the actual accumulation, i.e., before we process the message, thus removing the need to buffer $\boldsymbol{k}_\text{B}$ in a separate register.

5.6.1 A NEW CLASS OF TOEPLITZ-BASED CONSTRUCTIONS

Any state-machine generating a sequence that is internally stored in a shift register is suitable to be used with such an accumulator. One could, e.g., use the keystream from the stream cipher Trivium [DP08]. The keystream is not directly stored internally, but quite close, as the output is formed as the xor of a few internal bits. To ensure parallelism in Trivium, such a stored bit is never used within 64 register shifts after having been calculated. Since the keystream is generated through a simple xor of six bits, this means that xoring six 64-bit sections of the internal state, one has a logical register which can be accumulated as previously into a 64-bit tag. Theorem 5.1 then relates the security of this construction to the bias of Trivium's keystream. Finding any large bias would mean a strong distinguishing attack on Trivium, so we can be satisfied from a security point of view.

However, we consider it overkill to use a mechanism as strong and expensive as Trivium. We want a more hardware-efficient construction which might not be secure enough to be a keystream generator in a stream cipher, but which still generates an ε-biased distribution, where ε is small. This is sufficient for a good protection against a substitution attack.

We propose the following construction idea, aiming for three registers, and note the similarities with Grain (cf. Section 2.12).

We create the sequence $\boldsymbol{s}$ using an NFSR of size w, which is fed by an LFSR of size v. $|\boldsymbol{k}_\text{A}| = w + v$ and $|\boldsymbol{k}_\text{B}| = w$, so $|\boldsymbol{k}_\text{MAC}| = 2w + v$. An implementation is outlined in Figure 5.5.

Formalizing this construction, we denote by s_i, $i = 0, 1, \ldots$ the bitstream used in the Toeplitz construction. This bitstream is the output of the NFSR, and the initial state of the NFSR is $(s_0, s_1, \ldots, s_{w-1})$. The LFSR has initial state $(l_0, l_1, \ldots, l_{v-1})$ and generates an LFSR sequence $l_{i+v} = \sum_{j=0}^{v-1} d_j l_{i+j}$, for $i \geq 0$. The initial state of the two shift registers is chosen as

$$(s_0, s_1, \ldots, s_{w-1}, l_0, l_1, \ldots, l_{v-1}) = \boldsymbol{k}_\text{A}.$$

Finally, the NFSR sequence is obtained as

$$s_i = f(s_{i-w}, \ldots, s_{i-1}) + l_{i-w},$$

for $i \geq w$, where f is some nonlinear function. We refer to the two shift registers as the *generator*.

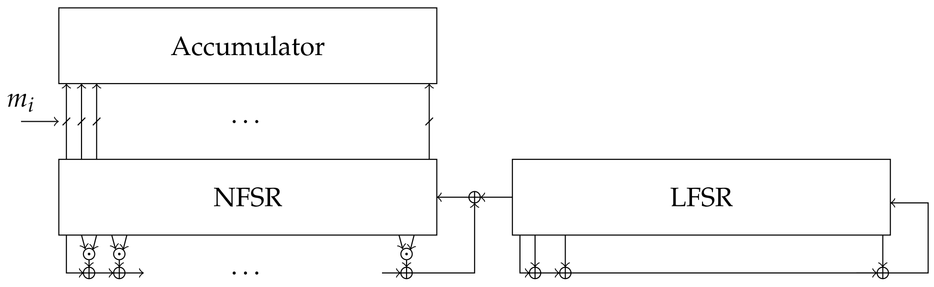


Figure 5.5: A rough hardware implementation of the suggested construction. The top register is the accumulator, where the contents of the left register is added if and only if the message bit is a one. The contents of the NFSR and LFSR are updated with each clock cycle.

We only allow the construction to authenticate messages of at most L bits, where the security parameter L will be studied later. It is strongly advisable to let $f(s_{i-w}, \ldots, s_{i-1})$ depend linearly on s_{i-w}, and this will be assumed below. This ensures that the state is invertible, so that there is no loss of entropy, i.e., at each time instance, 2^{w+v} different states of the generator are possible.

Let us briefly examine the hardware cost of this construction. Assume that both registers are of size w, i.e., the initial states are determined by $2w$ bits. An implementation as outlined in Figure 5.5 requires three registers of length w and some minor combinatorics. We need $3w$ flip flops, $2w$ XORs, $3w/2$ ANDs, and a small number of multiplexers, where the feedback functions have been estimated at w XORs and $w/2$ ANDs.[1] As an example, we need 1000 gates with $w = 32$. This compares well with the approaches above, keeping in mind that the CRC and LFSR-based constructions require random irreducible polynomials. We note that the 32-bit version of the block cipher KATAN can be implemented using around 800 gates [DDK09], although in that paper, flip flops are only 6.25 gates. Further, using KTANTAN and CBC-MAC in the setting considered here (Figure 5.1) would require some message buffering, which would increase the overall gate count of the authentication.

In order to quantify the security of this construction, we turn to Theorem 5.1 and consider the sequence $\boldsymbol{s}$ leaving the generator. If this sequence is random-looking enough in our sense, i.e., ε is low, the construction is se-

[1]The number of ANDs could, e.g., correspond to pairwise multiplication of all NFSR bits. The number of XORs could, e.g., correspond to adding those pairwise products and using half the LFSR bits in the feedback. We consider these approximations to be on the high side.

cure enough. Thus, it is irrelevant whether the construction could be used as a keystream generator (it can't, since observing $(s_0, s_1, \dots, s_{w+v-1})$, we can easily reconstruct the entire initial state), or whether it suits any other cryptographic applications. We intend to use it as a MAC, and all that matters is that it is an efficient construction which is secure in this very application.

5.6.2 ON THE PROBLEM OF FINDING THE BIAS

In order to use Theorem 5.1, we need to find (a bound on) the bias ε. Finding the actual value of the bias is difficult, but we can at least argue around it. If there is a linear approximation f' of the function f with bias ε', i.e., for some $\boldsymbol{\alpha}' \in \mathbb{F}_2^w$,

$$f'(\boldsymbol{x}) = \langle \boldsymbol{\alpha}', \boldsymbol{x} \rangle,$$

$$\left| \frac{|\{\boldsymbol{x} \in \mathbb{F}_{2^w} : \ f'(\boldsymbol{x}) = f(\boldsymbol{x})\}|}{2^w} - \frac{1}{2} \right| = \varepsilon',$$

we can use a linear test $\boldsymbol{\alpha}$, chosen by keeping the taps of the LFSR in mind, to achieve $\varepsilon = 2^{j-1}(\varepsilon')^c$, where $j-1$ is the number of taps in the LFSR or, equivalently, j is the weight of the LFSR polynomial. This means that if we have the bias ε' for the best linear approximation of f, and can rewrite the LFSR into an up-to-L-degree, weight-3 multiple of the feedback polynomial [MS89] [PK95] [Gol96] [Wag02], there is a linear test with a bias of $2^2(\varepsilon')^3$. With $L \leq 2^{v/2}$, we should expect there to be no weight-3 multiple of the LFSR polynomial of degree L [Gol96].

5.6.3 STRUCTURED FUNCTIONS f

The above does not promise that the bias of the best linear approximation $f'(\boldsymbol{x})$ will lead to the bias of the best linear test $\boldsymbol{\alpha}$.

Denote the entire initial state by

$$(r_0, \dots, r_{w-1}, r_w, \dots, r_{w+v-1}) = (s_0, \dots, s_{w-1}, l_0, \dots, l_{v-1}).$$

Each bit s_i, $i \geq 0$ can be described as a function of the initial state: $s_i = f_i(\boldsymbol{r})$. If the feedback function is highly irregular, we would expect the expressions for the different functions f_i to be highly different. Finding a linear combination of f_i's with a high bias seems difficult.

In Section 5.7, it will be seen that a structured feedback function $f(\boldsymbol{x})$ allows a very efficient choice of $\boldsymbol{\alpha}$.

For these reasons, it might be preferable to avoid structured feedback functions. One could, e.g., place the bits used nonlinearly at distances that compose a full difference set.

5.6.4 A CODING THEORY APPROACH

For each possible initial state, output L bits from the generator. Let the sequences form the columns of a matrix. We have now constructed the generator matrix of a code and the problem of finding the bias has been transferred into that of finding a codeword (i.e., a linear combination of rows) with a very high or low weight (high bias). By adding an all-one row to the matrix, this problem reduces to that of finding a low-weight codeword.

Recall from Subsection 3.3.2 that computing this characteristic of the code is generally difficult, which suggests that so is finding a linear test with high bias in this case. The algorithms mentioned in Subsection 3.3.2 can be used on smaller instances, but will not be successful in finding minimum-weight codewords if the initial state is too large. As an example, from [BLP08] it seems as if already with $w + v \gtrapprox 2^{11}$, this will not be feasible.

We note that a random code argument, cf. Subsection 3.3.1, can be used to derive an expected value of the bias, by deriving the expected minimal distance of a random linear code of suitable parameters. This would only give the expected bias and would not promise anything about a particular instance of the construction.

5.6.5 NUMERICAL RESULTS ON SMALLER INSTANCES

As exhaustively searching for the bias is out of the question for any practical tag size, we have instead carried out experiments on smaller instances of our construction. We have done as follows. For each sequence length $L \in \{1, 2, \dots, L_{\max}\}$ we have studied each $\boldsymbol{\alpha} \in \mathbb{F}_2^L$, $\boldsymbol{\alpha} \neq \mathbf{0}$. For each such $\boldsymbol{\alpha}$ in turn, we have evaluated the bias of $\langle \boldsymbol{s}, \boldsymbol{\alpha} \rangle$ over all $\boldsymbol{s}$ generated by all possible keys $\boldsymbol{k}_{\mathrm{A}}$. That is, the bias has been calculated by brute forcing over all initial states $\boldsymbol{\alpha}$ and all linear tests $\boldsymbol{s}$. With larger values of L there is a tendency for growing bias. See Figure 5.6 for the results.

We have used $w = v$ and tags of two different sizes w: 4 and 6 bits. The linear feedback polynomials have been chosen as $x^4 + x + 1$, and $x^6 + x + 1$, respectively (both primitive). The nonlinear feedbacks have been created by

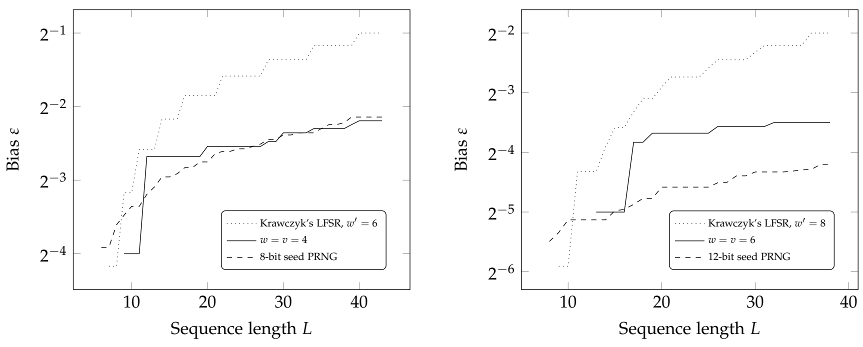
$$f(\mathbf{s}) = s_0 + s_1 s_3 + s_2 s_3,$$

and

$$f(\boldsymbol{s}) = s_0 + s_2 s_4 s_5 + s_3 s_4 s_5 + s_1 s_2 s_4 s_5 + s_2 s_3 s_4 s_5 + s_1 s_2 s_3 s_4 s_5,$$

respectively. With the tags of size four, the construction gave rise to a cycle set that contains some smaller cycles.

To judge the quality of the obtained biases, we have compared to the LFSR-based Toeplitz construction described in Subsection 5.5.5. Since there are not 2^w irreducible polynomials of degree w (for $w > 1$), one does not need w bits



(a) Biases for $w = v = 4$. **(b)** Biases for $w = v = 6$.

Figure 5.6: The bias as it develops for growing sequence lengths. Two different tag sizes w have been studied for the proposed construction (solid lines; $v = w$). It appears as if it achieves smaller biases than the LFSR construction (dotted lines; $w' > w$), but larger than a pseudorandom number generator (dashed lines). For small L, biases $\varepsilon = 0$ can be observed; these are not included in the figures. The PRNG has very small (nonzero) bias for small L; these have also been removed.

to describe the connection polynomial used with the LFSR-based construction. Put differently, using a fixed number $2w$ of random bits, the LFSR-based construction can use a state of size $w' > w$, where $2w = \log(\mathrm{L}_2(w')) + w'$. In order to not discriminate against the LFSR-based construction, we may round w' upwards, i.e., choosing the smallest integer w' such that

$$2w \leq \log(\mathrm{L}_2(w')) + w'.$$

As an example, with $2w = 12$ bits of key used for the proposed construction, the same number of key bits can be used with the LFSR-based construction and a tag size of 8: $\lceil \log(30) \rceil = 5$ bits are enough to enumerate the irreducible feedback polynomials (see Table 2.1), and the key is of total size $\log(30) + 8 \approx 12.9 \geq 12$. (With a tag size of 7, $\log(18) + 7 \approx 11.2$.)

We can, e.g., note that the bias of our proposed construction is below that of the LFSR construction on equal-length strings. Do keep in mind that we allow the LFSR construction to use slightly larger registers in order to compare for a fixed amount of randomness. The gate count is thus a factor approximately $\frac{8-6}{6} \approx 33\%$ larger than in our construction. (This completely ignores the issue of mapping the random key to an irreducible polynomial.)

As an additional comparison, we have derived the bias for a pseudorandom number generator (PRNG), using the key as seed.[2] We have tested several nonlinear feedback functions and can conclude that with higher nonlinearity, the bias is small. We do not repeat all results here as the figure would appear quite busy. This intuitive relation between nonlinearity and bias can be helpful during the design of larger constructions.

5.6.6 WEAK KEYS AND KEY REUSE

If $(l_0, l_1, \ldots, l_{v-1}) = \mathbf{0}$, the LFSR will be stuck in the all-zero state and the generator will only pass through 2^w different states, cf. Subsection 6.3.6. This will only happen with probability 2^{-v}.

No security promises can be made if the key and IV used to initialize the stream cipher with is reused, which would also endanger the encryption. If key–IV reuse is avoided with the stream cipher, $\boldsymbol{k}_{\text{MAC}}$ will still be reused with probability $2^{-w-v} < 2^{-w}$, so any attack that requires such $\boldsymbol{k}_{\text{MAC}}$ reuse to happen is unlikely to succeed.

Several attacks are possible if $\boldsymbol{k}_{\text{MAC}}$ is reused. For example, any message consisting of only zeros reveals the one time pad $\boldsymbol{k}_{\text{B}}$. Knowing the one time pad, the one-bit message (1) reveals the initial state of the NFSR. Finally, the message $\mathbf{0}^w||1$ allows us to easily reconstruct the initial LFSR state if $v \leq w$. Thus, three chosen messages are enough to get a full key recovery (or, for the stream cipher, the first $2w + v$ bits in the keystream). With variable-length messages, one would also have to consider the length padding scheme used, but it is straightforward to generalize the attack outlined here.

The fact that this attack is outside the security model does not, as Handschuh and Preneel point out [HP08], make it uninteresting. On the contrary, it highlights the necessity of always using fresh keys for the authentication, i.e., a fresh IV for the stream cipher.

5.7 AN EXAMPLE OF A SPECIFIC CONSTRUCTION AND AN ATTACK

We define a specific 32-bit construction and analyze it. We use $w = v = 32$, the invertible nonlinear feedback function

$$f(s_{i-w}, \ldots, s_{i-1}) = s_{i-w} + \sum_{j=1}^{w/2-1} s_{i-w+2j-1} s_{i-w+2j},$$

and the linear feedback specified by $l_i = l_{i-32} + l_{i-31} + l_{i-29} + l_{i-1}$ (see Figure 5.5).

[2]We have used the GNU Scientific Library's implementation of the Tausworthe pseudorandom number generator (`gsl_rng_taus2`).

Observe that the NFSR bits are calculated as $s_i = s_{i-32} + S_{i-31} + l_{i-32}$ for $i > 31$, where

$$S_i = \sum_{j=0}^{w/2-1} s_{i-w+2j-1} s_{i-w+2j}.$$

and that there is thus a lot of structure in the feedback function, cf. Subsection 5.6.3. By exploiting the fact that $S_i + S_{i+2} = s_i s_{i+1} + s_{i+w-2} s_{i+w-1}$, and that certain linear combinations of LFSR bits disappear, we can find that

$$\begin{aligned}
&s_0 + s_1 + s_2 + s_5 + s_{31} + s_{32} + s_{33} + s_{34} \\
&+ s_{32} + s_{33} + s_{34} + s_{37} + s_{63} + s_{64} + s_{65} + s_{66} \\
&+ s_1 s_2 + s_2 s_3 + s_4 s_5 + s_{31} s_{32} + s_{33} s_{34} + s_{34} s_{35} + s_{62} s_{63} + s_{63} s_{64} = 0,
\end{aligned}$$

i.e.,

$$\begin{aligned}
&s_0 + s_1 + s_2 + s_5 + s_{31} + + s_{37} + s_{63} + s_{64} + s_{65} + s_{66} \\
&+ s_1 s_2 + s_2 s_3 + s_4 s_5 + s_{31} s_{32} + s_{33} s_{34} + s_{34} s_{35} + s_{62} s_{63} + s_{63} s_{64} = 0. \qquad (5.2)
\end{aligned}$$

Reducing the number of multiplications by observing common factors, we get

$$\begin{aligned}
&s_0 + s_1 + s_2 + s_5 + s_{31} + s_{37} + s_{63} + s_{64} + s_{65} + s_{66} \\
&+ s_2(s_1 + s_3) + s_4 s_5 + s_{31} s_{32} + s_{34}(s_{33} + s_{35}) + s_{63}(s_{62} + s_{64}) = 0. \qquad (5.3)
\end{aligned}$$

There are five bit multiplications, and by replacing, e.g., $s_4 s_5$ by s_5, we get a sum of bits with a bias of 2^{-6}, assuming independence. There are some choices to make, but one can e.g., try to choose canceling bits and obtain a linear sum of bits $0, 1, 34, 37, 64, 65, 66$, i.e.,

$$\mathbf{Pr}\left[s_0 + s_1 + s_{34} + s_{37} + s_{64} + s_{65} + s_{66} = 0\right] = \frac{1}{2} + 2^{-6}.$$

This bias has been confirmed by simulations.

We construct an attack from a 67-bit vector

$$\boldsymbol{a} = (1,1)||\mathbf{0}^{32}||(1,0,0,1)||\mathbf{0}^{26}||(1,1,1), \qquad (5.4)$$

which is 1 precisely in these bits. The details of Theorem 5.1, as analyzed in Subsection 5.2.2, suggest that we choose to replace $\boldsymbol{m}$ by $\boldsymbol{m} + \boldsymbol{a}$ and $\boldsymbol{t}$ with $\boldsymbol{t} + \boldsymbol{b}$, where $\boldsymbol{b} = \mathbf{0}$.

The sum of various ε_ω runs over precisely those nonzero $\boldsymbol{\omega}$ that are linear combinations of shifted versions of $\boldsymbol{\alpha}^0 = 0 \ldots 0\boldsymbol{a}$. Since at any time instance, the generator can be in any of the 2^{w+v} possible states, shifted linear tests, e.g., $\boldsymbol{\alpha}^0$ and $\boldsymbol{\alpha}^1$, have the same bias, so $\varepsilon_{\boldsymbol{\alpha}^i} = \varepsilon_{\boldsymbol{a}}$, $i = 0, 1, \ldots, w-1$. It is then known from Equation 5.1 that

$$\mathbf{Pr}\left[\text{the attack succeeds}\right] \leq 2^{-w} + 2 \cdot 2^{-w} \cdot w \cdot \varepsilon_{\boldsymbol{a}} + \left| \sum_{\boldsymbol{\omega} \in \mathcal{R}_{>1}} \varepsilon_{\boldsymbol{\omega}} \right|. \qquad (5.5)$$

Using $\varepsilon_{\boldsymbol{a}} = 2^{-6}$ and assuming that $\varepsilon_{\boldsymbol{\omega}} = 0$, $\boldsymbol{\omega} \in \mathcal{R}_{>1}$, yields

$$\mathbf{Pr}\left[\text{the attack succeeds}\right] \leq 2^{-32} + 2 \cdot 2^{-32} \cdot 32 \cdot 2^{-6} = 2^{-32} + 2^{-32} = 2^{-31}.$$

Thus, we need to consider the other terms as well, and (as an attacker) hope that they add up to something significant.

From the fact that several of the biases are equal, we conclude that

$$\begin{aligned}\sum_{\boldsymbol{\omega} \in \mathcal{R}_{>1}} \varepsilon_{\boldsymbol{\omega}} = &(w-1) \cdot \varepsilon_{\boldsymbol{\alpha}^0 + \boldsymbol{\alpha}^1} \\ &+ (w-2) \cdot \varepsilon_{\boldsymbol{\alpha}^0 + \boldsymbol{\alpha}^2} \\ &+ \ldots \\ &+ 1 \cdot \varepsilon_{\boldsymbol{\alpha}^0 + \boldsymbol{\alpha}^{w-1}} \\ &+ \ldots \\ &+ 1 \cdot \varepsilon_{\boldsymbol{\alpha}^0 + \ldots + \boldsymbol{\alpha}^{w-1}}.\end{aligned}$$

All terms in the sum represent linear tests $\boldsymbol{\omega} \in \mathcal{R}_{>1}$, whose biases are not necessarily large. We have used a computer to bound the biases for these linear combinations $\boldsymbol{\omega}$, not by summing the shifted approximations $\boldsymbol{\alpha}^j$, but by summing the shifted exact expressions from Equation 5.2 and then approximating, for each sum getting the best linear approximation. This assumes that no linear test $\boldsymbol{\omega}$ is a good linear approximation for some other nonlinear relation in the bits of $\boldsymbol{s}$. That is, while

$$\begin{aligned}\boldsymbol{\alpha}^0 + \boldsymbol{\alpha}^1 =& \mathbf{0}^{30}||(0,1,1)||\mathbf{0}^{31}||(0,1,0,0,1)||\mathbf{0}^{25}||(0,1,1,1) \\ &+ \mathbf{0}^{30}||(1,1,0)||\mathbf{0}^{31}||(1,0,0,1,0)||\mathbf{0}^{25}||(1,1,1,0) \\ =& \mathbf{0}^{30}||(1,0,1)||\mathbf{0}^{31}||(1,1,0,1,1)||\mathbf{0}^{25}||(1,0,0,1),\end{aligned}$$

corresponding to the linear approximation

$$s_{30} + s_{32} + s_{64} + s_{65} + s_{67} + s_{68} + s_{94} + s_{97} = 0, \qquad (5.6)$$

can be found by simulations to have a bias of $\varepsilon_{\boldsymbol{\alpha}^0 + \boldsymbol{\alpha}^1} \approx 2^{-9}$, it is much faster to consider the nonlinear terms from Equation 5.2, which add up to

$$\begin{aligned}& s_{32}s_{33} + s_{33}s_{34} + s_{35}s_{36} + s_{62}s_{63} + s_{64}s_{65} + s_{65}s_{66} + s_{93}s_{94} + s_{94}s_{95} \\ &+ s_{31}s_{32} + s_{32}s_{33} + s_{34}s_{35} + s_{61}s_{62} + s_{63}s_{64} + s_{64}s_{65} + s_{92}s_{93} + s_{93}s_{94} \\ &= s_{31}s_{32} + s_{34}(s_{33} + s_{35}) + s_{35}s_{36} + s_{62}(s_{61} + s_{63}) \\ &+ s_{63}s_{64} + s_{65}s_{66} + s_{92}s_{93} + s_{94}s_{95}.\end{aligned}$$

Since there are 8 bit multiplications, the best linear approximation has bias 2^{-9}, assuming independence, so the bound is $\varepsilon_{\alpha^0+\alpha^1} \leq 2^{-9}$.

To be very specific, the sum of the shifted versions of Equation 5.2 is

$$\begin{aligned}
&s_{31} + s_{32} + s_{33} + s_{36} + s_{62} + + s_{68} + s_{94} + s_{95} + s_{96} + s_{97} \\
&+ s_{32}s_{33} + s_{33}s_{34} + s_{35}s_{36} + s_{62}s_{63} + s_{64}s_{65} + s_{65}s_{66} + s_{93}s_{94} + s_{94}s_{95} \\
&+ s_{30} + s_{31} + s_{32} + s_{35} + s_{61} + + s_{67} + s_{93} + s_{94} + s_{95} + s_{96} \\
&+ s_{31}s_{32} + s_{32}s_{33} + s_{34}s_{35} + s_{61}s_{62} + s_{63}s_{64} + s_{64}s_{65} + s_{92}s_{93} + s_{93}s_{94} \\
&= s_{30} + s_{33} + s_{35} + s_{36} + s_{61} + s_{62} + + s_{67} + s_{68} + s_{93} + s_{97} \\
&+ s_{31}s_{32} + s_{33}s_{34} + s_{34}s_{35} + s_{35}s_{36} + s_{61}s_{62} \\
&+ s_{62}s_{63} + s_{63}s_{64} + s_{65}s_{66} + s_{92}s_{93} + s_{94}s_{95} = 0.
\end{aligned}$$

From this, one can construct a best linear approximation with bias 2^{-9}, but also the linear approximation in Equation 5.6, which has bias $\leq 2^{-9}$. Independence is assumed when deriving this bound, but simulations have shown the bias to be very close to 2^{-9} which makes the assumption reasonable.

In this way, one can bound $\sum \varepsilon_\omega$ and obtain

$$\mathbf{Pr}\left[\text{the attack succeeds}\right] < 2^{-32} + 2^{-27}.$$

This bound applies to all choices of $\boldsymbol{a}$ with $\varepsilon_{\boldsymbol{a}} = 2^{-6}$ based on Equation 5.3. Running the attack, using $\boldsymbol{a}$ from Equation 5.4 on 2^{42} random keys, this probability was found experimentally as $3407 \cdot 2^{-42} \approx 2^{-30.3}$.

There might be other values of $\boldsymbol{a}$ that give a higher success probability (for some $\boldsymbol{b}$), but it should be kept in mind that this attack is constructed directly from a linear test that exploits the structure of the construction.

5.8 CONCLUSION

We have proposed a new type of MAC construction, which appears interesting from a hardware perspective. The substitution probability is expected to be low, as it is bounded by the bias of the output stream and we demonstrate how this bound can be expected to be far from tight.

This class of constructions appears very promising and it would be interesting to find hardware-efficient designs where we can derive the bias explicitly, so that we can get a better understanding of the largest possible success probability for the substitution attack.

6 Grain-128a: A New Stream Cipher with Optional Authentication

This chapter presents a new version of Grain-128, namely Grain-128a, originally proposed in [ÅHJM11]. The new stream cipher has native support for authentication, and is expected to be comparable to the old version in hardware performance. Any differences between the algorithms presented in this chapter and [ÅHJM11] are unintentional, and [ÅHJM11] is the canonical version.

The authentication supports variable tag sizes w up to 32 bits, and varying $w \neq 0$ does not affect the keystream generated by Grain-128a. With $w = 0$, i.e., no authentication, the keystream is different compared to using $w \neq 0$ as the construction can then be more efficient.

Grain-128a uses slightly different nonlinear functions in order to strengthen it against the known attacks and observations on Grain-128. Existing implementations of Grain-128 can be reused to a very large extent as the changes, summarized in Section 6.5, are modest. This also allows us to have a high confidence in Grain-128a, as the cryptanalysis carries over from Grain-128.

The details of the design are specified in Section 6.1. The throughput is discussed in Section 6.2, and a security analysis is performed in Section 6.3. The design choices are motivated theoretically in Section 6.4, and Section 6.5 details the differences to Grain-128. The hardware performance is discussed in Section 6.6. Section 6.7 makes recommendations regarding the various members of the Grain family of stream ciphers. Section 6.8 contains several test vectors, before Section 6.9 concludes the chapter.

6.1 DESIGN DETAILS

Grain-128a consists of a mechanism that produces a preoutput stream, and two different modes of operation: with or without authentication. Figure 6.1 depicts an overview of the building blocks of the preoutput generator, which is constructed using three main building blocks, namely an LFSR, an NFSR and a preoutput value calculator. We denote by $s_i, s_{i+1}, \ldots, s_{i+127}$ the contents of the LFSR. Similarly, the content of the NFSR is denoted by $b_i, b_{i+1}, \ldots, b_{i+127}$. Together, the 256 memory elements in the two shift registers represent the state of the preoutput generator.

The primitive feedback polynomial of the LFSR, denoted $f(x)$, is defined as

$$f(x) = 1 + x^{32} + x^{47} + x^{58} + x^{90} + x^{121} + x^{128}.$$

To remove any possible ambiguity we also give the corresponding update function of the LFSR as

$$s_{i+128} = s_i + s_{i+7} + s_{i+38} + s_{i+70} + s_{i+81} + s_{i+96}.$$

The nonlinear feedback polynomial of the NFSR, $g(x)$, is defined as

$$\begin{aligned} g(x) = 1 &+ x^{32} + x^{37} + x^{72} + x^{102} + x^{128} + x^{44}x^{60} \\ &+ x^{61}x^{125} + x^{63}x^{67} + x^{69}x^{101} \\ &+ x^{80}x^{88} + x^{110}x^{111} + x^{115}x^{117} \\ &+ x^{46}x^{50}x^{58} + x^{103}x^{104}x^{106} + x^{33}x^{35}x^{36}x^{40}. \end{aligned}$$

To once more remove any possible ambiguity we also give the rule for updating the NFSR.

$$\begin{aligned} b_{i+128} = s_i &+ b_i + b_{i+26} + b_{i+56} + b_{i+91} + b_{i+96} \\ &+ b_{i+3}b_{i+67} + b_{i+11}b_{i+13} + b_{i+17}b_{i+18} \\ &+ b_{i+27}b_{i+59} + b_{i+40}b_{i+48} + b_{i+61}b_{i+65} \\ &+ b_{i+68}b_{i+84} + b_{i+88}b_{i+92}b_{i+93}b_{i+95} \\ &+ b_{i+22}b_{i+24}b_{i+25} + b_{i+70}b_{i+78}b_{i+82}. \end{aligned}$$

Note that the update rule contains the bit s_i which is output from the LFSR and masks the input to the NFSR, while it was left out in the feedback polynomial.

Nine state variables are taken as input to a Boolean function, $h(\boldsymbol{x})$: two bits come from the NFSR and seven from the LFSR. This function is defined as

$$h(\boldsymbol{x}) = x_0x_1 + x_2x_3 + x_4x_5 + x_6x_7 + x_0x_4x_8,$$

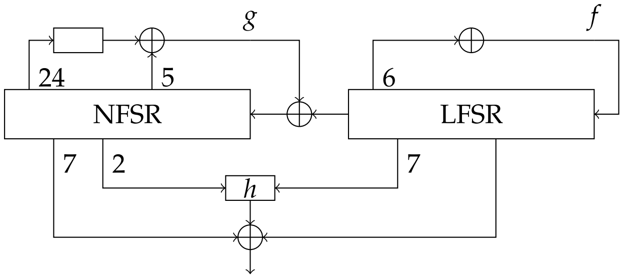


Figure 6.1: An overview of the preoutput generator.

where the variables $x_0, \ldots, x_8$ correspond to, respectively, the state variables b_{i+12}, s_{i+8}, s_{i+13}, s_{i+20}, b_{i+95}, s_{i+42}, s_{i+60}, s_{i+79} and s_{i+94}. The preoutput value y_i is defined as

$$y_i = h(\boldsymbol{x}) + s_{i+93} + \sum_{j \in \mathcal{A}} b_{i+j},$$

where $\mathcal{A} = \{2, 15, 36, 45, 64, 73, 89\}$. How the preoutput bits are used for keystream generation and possibly authentication depends on the mode of operation and is detailed in Subsections 6.1.2–6.1.4.

6.1.1 KEY AND IV INITIALIZATION

Before keystream is generated and authentication is performed, the cipher must be initialized with the key and the IV. Denote the bits of the key as k_i, $0 \le i \le 127$ and the IV bits v_i, $0 \le i \le 95$. Initialization of the key and IV is done as follows. The 128 NFSR elements are loaded with the key bits, $b_i = k_i$, $0 \le i \le 127$, and the first 96 LFSR elements are loaded with the IV bits, $s_i = v_i$, $0 \le i \le 95$. The last 32 bits of the LFSR are filled with ones and a zero, $s_i = 1$, $96 \le i \le 126$, $s_{127} = 0$. Then, the cipher is clocked 256 times without producing any keystream. Instead the preoutput value is fed back and xored with the input, both to the LFSR and to the NFSR, see Figure 6.2.

6.1.2 MODES OF OPERATION

Grain-128a supports two different modes of operation: with and without authentication. Authentication is mandatory when $v_0 = 1$, and forbidden when $v_0 = 0$ (see Subsection 6.4.7 for security details). Exactly how this is enforced is up to the application—if an implementation that does not support authentication is loaded with $v_0 = 1$, it may, e.g., emit some error indicator, force a protocol termination, or in some other way refuse to continue. An application

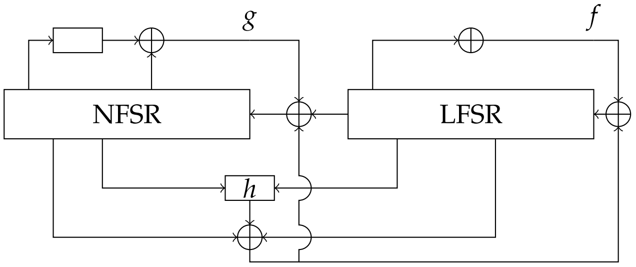


Figure 6.2: The key initialization.

that never (always) uses authentication may choose not to transmit the value of v_0 as it is already uniquely determined as 0 (1).

With $v_0 = 0$, it is still possible and allowed to use some other, separate authentication algorithm. What is forbidden is using Grain-128a with authentication when $v_0 = 0$.

6.1.3 KEYSTREAM GENERATION

With $v_0 = 1$, the output function is defined as

$$z_i = y_{64+2i},$$

meaning that we use every second bit as output of the cipher after ignoring the first 64 bits. Those 64 initial bits and the other half will be used for authentication, see Subsection 6.1.4.

With $v_0 = 0$, the output function is defined as simply

$$z_i = y_i,$$

meaning all preoutput bits are used directly as keystream. This mode of operation is the same as in Grain-128.

6.1.4 AUTHENTICATION

Assume that we have a plaintext of length L defined by the bits $p_0, \dots, p_{L-1}$. Set $p_L = 1$. Note that $p_L = 1$ is the padding, which is crucial for the security of the authentication as it ensures that $\boldsymbol{p}$ and $\boldsymbol{p}||0$ (of lengths L and $L+1$, respectively) have different tags with very high probability.

In order to provide authentication, two registers of size 32 are used. They are called the accumulator and the shift register. The content of the accumulator at time i is denoted by $a_0^i, \dots, a_{31}^i$, and the content of the shift register at time i is denoted by $r_i, \dots, r_{i+31}$. The accumulator is initialized

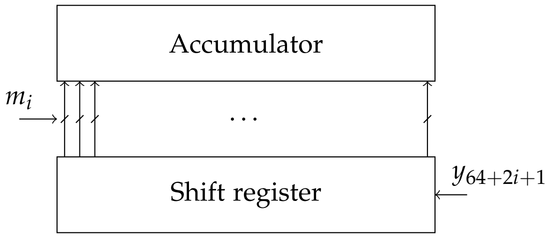


Figure 6.3: An overview of the authentication as it is clocking in plaintext and preoutput bits.

through $a_j^0 = y_j$, $0 \leq j \leq 31$, and the shift register is initialized through $r_i = y_{32+i}$, $0 \leq i \leq 31$. The shift register is updated as $r_{i+32} = y_{64+2i+1}$. The accumulator is updated as $a_j^{i+1} = a_j^i + p_i r_{i+j}$ for $0 \leq j \leq 31$ and $0 \leq i \leq L$.

The final content of the accumulator, $a_0^{L+1}, \ldots, a_{31}^{L+1}$, is the 32-bit tag to be used for authentication, i.e., $t_i = a_i^{L+1}$, $0 \leq i \leq 31$. See Figure 6.3 for a graphical representation of the authentication mechanism.

To guarantee an implementation-independent use of shorter tags, we define w-bit tags through $t_i^{(w)} = t_{32-w+i}$, $0 \leq i \leq w-1$. This amounts to using the rightmost part of the tag in Figure 6.3. Clearly, the tag size w must be fixed by an implementation, or negotiated in some secure way which is not detailed here.

6.2 THROUGHPUT RATE

All shift registers are regularly clocked so the cipher will output one bit every clock, or every second clock when using authentication. This regular clocking is an advantage, both in terms of performance and resistance to side-channel attacks, compared to using irregular clocking or decimation.

An important feature of the Grain family of stream ciphers is that the speed can be increased at the expense of more hardware. This requires the feedback functions, f and g, and the preoutput value calculation to be implemented several times. To aid this, the last 31 bits of the shift registers in the preoutput generator, s_i, b_i, $97 \leq i \leq 127$ are not used in the respective feedback function or to calculate the preoutput value. This allows the speed to be easily multiplied by up to 32 if a sufficient amount of hardware is available.

An overview of the implementation when the speed is doubled can be seen in Figure 6.4. The shift registers also need to be implemented such that each bit is shifted j steps instead of just one when the speed is increased by a factor of j. The possibilities to increase the speed is limited to powers of

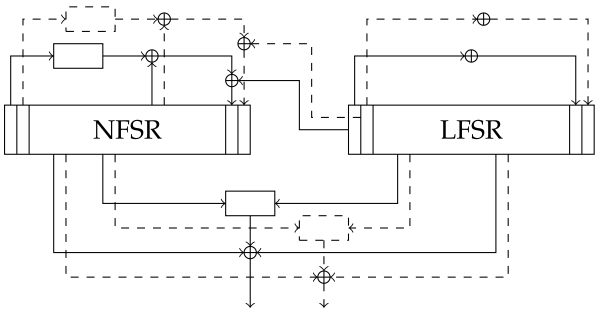


Figure 6.4: The cipher when the speed is doubled.

two as j needs to divide, e.g., the initialization count, which is 256, and the authentication initialization, which is another 64 basic clockings. Since the preoutput and feedback functions are small, it is quite feasible to increase the throughput in this way. By increasing the speed by a factor of 32, the cipher will output 32 bits/clock, or 16 bits/clock when using authentication.

For more discussion about the hardware implementation of Grain-128a, we refer to Section 6.6.

6.3 SECURITY EVALUATION

Excellent hardware performance is of little use if the cipher is not secure. We outline several possible cryptanalytic attacks, and build upon these insights to decide on the different functions and parameters used in Grain-128a.

In the following, we will consider the preoutput stream y_i, as the keystream z_i is just as good from a security point of view (but half the length), and the authentication will rely on the security of the preoutput stream.

6.3.1 LINEAR APPROXIMATIONS

Golić [Gol94] realized that in any stream cipher, one can always find some linear combination of the output bits that has a nonzero bias. In this section, we consider the general Grain design, ignoring specifics such as the exact choices of f, g, and h. The function f is of course restricted to being a primitive polynomial, as it is the feedback function of the LFSR. Updating the NFSR is similarly made through g, and the output is created using h. To simplify notation in this section, we denote by h the entire preoutput function, i.e., h includes the bits added linearly in the preoutput function.

Maximov [Max06] studied this general structure and introduced g' and h'

as linear approximations for g and h with biases ε_g and ε_h, respectively, i.e.,

$$\begin{aligned}\mathbf{Pr}\left[g'(x) = g(x)\right] &= 1/2 + \varepsilon_g,\\ \mathbf{Pr}\left[h'(x) = h(x)\right] &= 1/2 + \varepsilon_h.\end{aligned}$$

Then, a time invariant linear combination of the keystream bits and LFSR bits exists, and the bias of this equation is

$$\varepsilon = 2^{(\eta(h')+\eta(g')-1)} \cdot \varepsilon_g^{\eta(h')} \cdot \varepsilon_h^{\eta(g')}, \tag{6.1}$$

where $\eta(f')$ is the number of the NFSR state variables used in the function f'. The LFSR taps have not been accounted for, and this bias can not be readily used in any attack. However, by summing shifted versions of this function, so that the LFSR contributions add up to zero, a practical attack can be mounted, at least if the bias ε of the new linear equation is large. Finding a low weight parity check equation [MS89] [PK95] [Gol96] [Wag02] for the LFSR improves this ε at the expense of requiring longer keystream, and the precomputation of finding such a parity check equation. Maximov also showed that the strength of Grain against correlation attacks is based on the difficulty of the general decoding problem (GDP), which is well-known to be a hard problem. Various time–memory trade-off approaches to the GDP have been discussed in the literature, e.g., [JJ99] [JJ00] [CJS00] [MFI02] [CJM02].

As one can always find a biased linear approximation a' for any function a, one can never eliminate the biased nature of Grain's output. It thus comes down to choosing particular functions g and h such that this bias is extremely small, so that the resulting attack will be a less promising choice than a simple brute force.

6.3.2 ALGEBRAIC ATTACKS

The individual bits in the preoutput stream can be expressed as functions of the state immediately after initialization. Thus, with access to a stream of such bits, an attacker can attempt to solve the corresponding system of equations. If Grain-128a did not contain the NFSR, i.e., it was a basic filter generator, such algebraic attacks could be very successful. However, Grain-128a *does* use an NFSR, which introduces much more nonlinearity, together with h [CM03]. Solving equations for the initial 256 bit state is not possible due to the nonlinear update of the NFSR and the NFSR state bits used nonlinearly in h [BGJ08].

6.3.3 TIME–MEMORY–DATA TRADE-OFF ATTACK

A generic attack that can be applied to a large class of cryptographic primitives, and on stream ciphers in particular, is the time–memory–data trade-off attack. The cost requirement is $2^{n/2}$ where n is the size of the state [BS00]. As the state in Grain-128a is of size 256, the expected time requirement of such an attack is 2^{128}, which does not compare favorably to brute force.

6.3.4 FAULT ATTACKS

Fault attacks were introduced by Hoch and Shamir [HS04] and have been efficient against many known stream cipher constructions. Whether they are practical is not so clear: one scenario in a fault attack is to allow the adversary to introduce some bit flipping faults to one of the shift registers. We note that faults in the NFSR should be harder to trace than faults in the LFSR, and the strongest assumption possible is therefore that the adversary can introduce a single fault in a location of the LFSR that they can somehow determine. When the fault propagates to position b_{i+95}, the difference has spread to the NFSR-related output, and is soon introducing nonlinearities. Until that point in time, the difference observed in the output is coming only from inputs of h from the LFSR. Allowing the adversary to reset Grain-128a many times, each time introducing a new fault, might enable them to acquire information about some subset of LFSR bits. Slightly more realistic assumptions on the ability to introduce a known number of faults makes it more difficult to deduce LFSR bits from the differences in output.

6.3.5 SIDE-CHANNEL ATTACKS

Any attacker that can observe some signal that is emitted from the implementation of a cryptographic primitive—most often power consumption or some function thereof—and that is dependent on the inner calculations, may be able to deduce the numbers, bits, etc. used in these calculations and thus, e.g., the key or the plaintext.

We note that the authentication mechanism performs work on two vastly different levels of power consumption. Viewing a power diagram of a naive implementation that processes one plaintext bit every clocking, it should be easy to tell apart ones from zeros.

Just as with any other cryptographic primitive, care must be taken to protect an implementation of Grain-128a against side-channel attacks such as differential power analysis [KJJ99] [FGKV07] [MD12].

6.3.6 WEAK KEY–IV PAIRS

Zhang and Wang [ZW09] have shown that there are 2^{96} weak key–IV pairs in Grain-128, each leading to an all-zero LFSR after the initialization phase, cf. Subsection 5.6.6. They have also demonstrated how to distinguish such keystream, and how to recover the initial state.

We note that the IV is normally assumed to be public, and that the probability of using a weak key–IV pair is 2^{-128}. Any attacker guessing this to happen and then launching a rather expensive attack, is much better off just guessing a key.

6.3.7 THE AUTHENTICATION

By Theorem 5.1, an attacker who replaces $(\boldsymbol{p}, \boldsymbol{t})$ with $(\boldsymbol{p} + \boldsymbol{a}, \boldsymbol{t} + \boldsymbol{b})$ has a success probability bounded by $2^{-32} + 2\varepsilon$, where ε measures the randomness in the sequence of bits used for authentication (i.e., a subsequence of the preoutput sequence). Note that an equivalent attack approach is to replace $(\boldsymbol{c}, \boldsymbol{t})$ with $(\boldsymbol{c} + \boldsymbol{a}, \boldsymbol{t} + \boldsymbol{b})$ (cf. Section 2.6).

From Equation 6.1 in Subsection 6.3.1 and specific values of $\varepsilon_g, \varepsilon_h$ given later, we know that $\varepsilon \ll 2^{-32}$ in our case, when we exploit the structure of the preoutput generator. It is therefore not unreasonable to claim that the success probability of this substitution attack is bounded by approximately 2^{-32}, and that the best attack is to basically guess the tag. The attack probability is similarly bounded by approximately 2^{-w} for w-bit tags. As a second argument supporting a negligible bias, we note that if there exists a larger bias, it would give a very strong distinguisher on the preoutput generator. No distinguisher is known on Grain-128 despite extensive research by the cryptographic community and it containing less nonlinearity than the preoutput generator of Grain-128a.

From the work by Handschuh and Preneel [HP08] (cf. Subsection 5.6.6) it is also clear that avoiding reuse of the key–IV pair is crucial to the security of the authentication, just as it is for the encryption. An attacker who is able to tweak a plaintext–tag pair and have it accepted (this happens with probability 2^{-w}) will be able to perform subsequent forgeries with probability 1 if the key–IV pair is reused.

The authentication mechanism is very similar to that in the 3GPP algorithm 128-EIA3 [ETS11a], which uses the stream cipher ZUC [ETS11b]. However, in 128-EIA3, two entirely different instances of ZUC are used. The IVs are similar or even equal, but two different keys are utilized: one for encryption and one for authentication. As encryption and authentication are performed simultaneously, one needs to utilize two implementations of ZUC or an expensive buffering. We consider our approach superior from a hardware point of view as the authentication and encryption share the preoutput stream of a single instance of Grain-128a.

Note that a draft version of 128-EIA3 was broken by Fuhr et al. [FGRV10]. This attack does not apply to Grain-128a as it avoids the exploited problem through the technique mentioned independently by [FGRV10] and in Section 5.4. Thus, Grain-128a extracts the one time pad, used to finalize the MAC, from the beginning of the preoutput stream rather than the end. In a later publication, Fuhr et al. [FGRV12] note that the updated 128-EIA3 contains some subtle weaknesses as the »one time pad« is still not taken from the beginning of the preoutput stream. These unwanted properties are not present in Grain-128a.

Fuhr et al. [FGRV10] also wonder whether the use of the IV affects the security of the authentication negatively—it does not; if the constant key, variable IV used with the authentication mechanism in Grain-128a was a problem, there would exist a strong distinguisher on the preoutput stream when Grain-128a is used as any other modern stream cipher: constant key, variable IV. In particular, there would be a distinguisher on Grain-128a when used without authentication, i.e., when used precisely as Grain-128.

6.4 DESIGN CHOICES

From the above, it is apparent that it is crucial to select design parameters with great care. This section contains the details regarding the choices for the parameters and functions used in Grain-128a.

6.4.1 SIZE OF THE LFSR AND THE NFSR

The size of the key in Grain-128a is 128 bits. Considering the simple and generic time–memory–data trade-off attack, the size of the internal state must be at least twice that of the key. Thus, we decide on an internal state consisting of 256 bits. Dividing these equally between the NFSR and the LFSR is an apparent choice.

6.4.2 SPEED ACCELERATION

As outlined previously, Grain-128a can be made significantly faster by implementing the functions f, g, and h several times. For a simple implementation of this speed acceleration up to a factor 32, these functions should be chosen not to use variables taken from the 31 rightmost taps of the registers, as seen in Figure 6.1.

6.4.3 THE BOOLEAN FUNCTION f

As f should be the generating polynomial for the LFSR, and we want the period to be maximal, we need f to be primitive. It is well-known that polynomials of low weight can be exploited in various correlation attacks [CT00]. This implies that we should use many taps of the LFSR, but on the other hand, it is undesirable to use a very large number of taps, due to the hardware cost.

6.4.4 THE BOOLEAN FUNCTION g

The purpose of this function is to create nonlinear relations between state bits, and we need to avoid the attack described in Subsection 6.3.1. The best linear approximation of g is of considerable interest, and for it to contain many terms, we need the resiliency of the function g to be high. We also need a high nonlinearity in order to obtain a small bias. To construct g, we thus use two functions—one with high nonlinearity and a linear one with high resiliency. The function

$$\begin{aligned} b(x) = {} & x_0x_1 + x_2x_3 + x_4x_5 + x_6x_7 + x_8x_9 + x_{10}x_{11} \\ & + x_{12}x_{13} + x_{14}x_{15}x_{16} + x_{17}x_{18}x_{19} + x_{20}x_{21}x_{22}x_{23}, \end{aligned}$$

collecting the nonlinear terms, has nonlinearity 8356352. In order to strengthen the resiliency, 5 linear terms are added to the function. As a result, g is balanced, has nonlinearity $2^5 \cdot 8356352 = 267403264$ and resiliency 4. The set of best linear approximations is the set of linear functions where at least all the linear terms of g are present, and the monomials of degree at least three are approximated by 0. This set is of size 2^{14} and all the functions in it have bias $\varepsilon_g = 63 \cdot 2^{-15} < 2^{-9}$.

6.4.5 THE PREOUTPUT FUNCTION

In order to make it certain that both registers affect the preoutput in each time step, terms from both registers are added linearly to the function h, which also uses bits from both registers. The nonlinearity of h is 240 and adding 8 variables linearly yields a total nonlinearity of $2^8 \cdot 240 = 61440$. The best linear approximation has bias $\varepsilon_h = 2^{-5}$, and there are in total 2^8 linear approximations of h with that bias.

6.4.6 AUTHENTICATION MECHANISM

From Chapter 5 it appears as if there is a choice to make between

1. the number of gates used in a construction (typically w-bit registers),
2. security (substitution attack success probability), and
3. keystream requirement (using a lot of keystream vs. processing an initial seed).

Since Grain-128a aims to be cost-efficient in hardware yet very secure, the third parameter, keystream consumption during authentication, has been allowed to become high. Indeed, more preoutput bits are used for authentication than for encryption.

There is, however, a very natural explanation for this under the assumption that whoever is about to implement the authentication mechanism in Grain-128a has already implemented its encryption mechanism. As mentioned in Section 6.2, it is quite cheap to double the rate of Grain-128a. Thus, the cost of upgrading from Grain-128a without any authentication to also using authentication amounts to the authentication mechanism itself and some additional gates in order to double the rate.

Note that we could have created two keystreams from the NFSR and LFSR—one for encryption and one for authentication. This would in a sense allow us to double the throughput, but could have disastrous drawbacks if we are not very careful.

6.4.7 TWO MODES OF OPERATION

Grain-128a without authentication is able to produce keystream at twice the rate of Grain-128a with authentication, which is of course very valuable. It is crucial that these two modes of operations are not allowed to use the same preoutput stream, i.e., the same key–IV pair.

For a short while, assume there was no such restriction. Consider now a known plaintext on a version without authentication. This would give the attacker the entire preoutput stream. If the receiver could be tricked into using 32-bit tags, the attacker could not only spoof an encryption (which is of course trivial with known keystream), but also the corresponding authenticating tag, thus elevating the supposed security of the scheme while still breaking it. (An attacker able to shorten the tags is of course very powerful, but that increasing the tag size from 0 to 32 could be a security problem is not at all obvious.)

Regarding which particular IV bit to use for this partitioning, the obvious candidates were v_0 and v_{95}, of which we chose the former. We also considered introducing another bit, separate from the IV, so that the LFSR is loaded with a 96-bit IV, one bit signaling use of authentication, and 31 constant bits.

6.4.8 SUPPORT FOR VARIABLE TAG SIZES

We have selected 32 bits as an upper tag size, as any application using Grain-128a is supposedly operating under some resource constraints and using, e.g., 64 bits seems superfluous. Also, support for 64 bit tags would mean more clockings before keystream generation begins when using shorter tags.

Note that a different approach could have been taken to allowing variable tag sizes: when initializing the authentication mechanism, only use the minimal amount of preoutput bits, i.e., do not discard any preoutput bits. Using a certain key and IV, different tag sizes would naturally lead to different keystreams, but more worryingly, knowledge of a short tag for a plaintext would give knowledge about longer tags, meaning an attacker (similar to above) who could make the receiver consider a longer tag of length w would be able to have it accepted with probability significantly greater than 2^{-w}.

Considering this, we have decided to predetermine which preoutput bits are used for what purpose. This does mean that applications with smaller tags will see a small overhead, but the overall confidence in the algorithm will be greater.

6.4.9 AUTHENTICATION INITIALIZATION

We load the accumulator with the first 32 preoutput bits, and the shift register with the next 32. An alternative would have been to alternately load one bit into each register, i.e., $r_i = y_{2i}$, $a_i^0 = y_{2i+1}$ for $0 \leq i \leq 31$. This would have meant that for shorter w, a chunk of preoutput bits would have been discarded, and another chunk (the »end« before keystream generation begins) used to initialize the authentication mechanism. This could be interpreted as a prolonged initialization of Grain-128a.

Our specification instead uses two separate chunks to load the accumulator and the register. With $w < 32$, this means that the discarded preoutput bits are found in two separate blocks. We note, however, that this allows the accumulator to be loaded through the accumulating mechanism: one can load the first chunk of preoutput bits into the register and then »accumulate« it onto a zeroed accumulator.

Cryptanalytically, we note that the alternative approach would have allowed an attacker to access the xor of the two supposedly »weakest« preoutput bits: $r_0 + a_0^0 = y_0 + y_1$.

Instead, the attacker can only learn these bits masked with bits that are produced later, being even more initialized: $r_0 + a_0^0 = y_0 + y_{32}$. This is not to imply that we do not trust the preoutput bits to be properly initialized—we only note that some bits are even more initialized, and it seems favorable to mix less and more initialized ones.

6.5 DIFFERENCES FROM GRAIN-128

A number of changes have been made compared to Grain-128. In this section, we list and motivate each of these differences.

6.5.1 IV SPACE PARTITIONING

Authentication is either mandatory or forbidden depending on the bit v_0. This partitioning of the IV space has been introduced due to security reasons as outlined above in order to allow Grain-128a without authentication to double the throughput.

6.5.2 THE FUNCTION g

We have added three monomials: two of degree three and one of degree four. This is in response to the results by Aumasson et al. [ADH$^+$09] and Stankovski [Sta10]. Both these results relate to trying to find a set of IV bits which take all possible values, while the remaining key and IV bits are fixed. As an example, with a set of 40 IV bits, one requests the first bit of the 2^{40} keystreams corresponding to the 2^{40} different IVs. The first bit in the keystream is a function of the key and IV bits, and by processing these 2^{40} »first bits«, one might be able to find some information on the secret key, at least if the function describing this bit is not complicated enough. It is natural to study instead the bits that are discarded during the initialization, as it is supposedly easier to find any information in them, and it should be possible to get an idea of whether the initialization is strong enough. More details are available in the papers.

Stankovski defines a nonrandomness threshold and claims that there is nonrandomness throughout the full 256 rounds of initialization of Grain-128. This implies that the key and IV material is not properly mixed before keystream generation starts, and highlights that the initialization used too few clockings and/or too little nonlinearity.

We tried Stankovski's algorithm on variants of Grain-128a, analyzing the initialization, where we used several different candidate polynomials $g_i(x)$. We finally settled on one that had very good behavior in terms of passing the nonrandomness tests of [Sta10]. The results are shown in Figure 6.5. While this does not *prove* that Grain-128a mixes key and IV variables enough, it shows that the new design is less susceptible to this problem.

The upper curve is Stankovski's result on Grain-128, where they start from the optimal bitset of size 6, using only IV bits, and continuously add two bits according to their greedy algorithm in order to find good bitsets (cubes) where many initialization »output« bits xor to zero. Finally they reach a bitset of size 40 such that all initialization output bits xor to zero. The same strategy does not work as well on the initialization of Grain-128a. The curve starts lower and does not rise. We have launched an even more computationally demanding strategy of adding three bits rather than two in each step, but the curve resulting from that experiment shows the same nongrowing tendency and has been excluded to avoid cluttering the figure.

6.5.3 IV INITIALIZATION

Setting $s_{127} = 0$ during IV initialization is a direct response to the observation by Küçük [Küç06], who pointed out that by simply using sixteen ones, `ffff`, to pad the IV register, there was a high probability that two very similar key–IV pairs would produce keystreams that were shifted variants of each other.

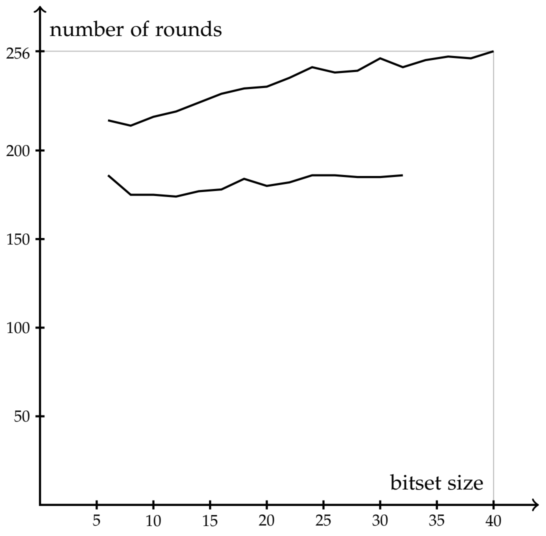


Figure 6.5: Stankovski's algorithm applied to Grain-128 (upper curve) and Grain-128a (lower curve).

As a consequence of this change, the padding constant is instead `fffe`, and the previously known attacks on Grain-128 [Küç06] [DKP08] [LJSH08] are not applicable on Grain-128a.

6.5.4 AUTHENTICATION

We add optional authentication to Grain-128a. Without authentication, the mode of operation of Grain-128a is the same as in Grain-128: preoutput is used as-is for keystream.

6.5.5 THROUGHPUT RATE

With authentication, the throughput rate is lower than in Grain-128, but it is quite easy to double it in response. Without authentication, there is no change in throughput rate.

6.5.6 A TAP IN THE PREOUTPUT FUNCTION h

Dinur and Shamir [DS11] used techniques similar in spirit to Stankovski's in what they dub a dynamic cube attack. For a fraction 2^{-10} of all keys, they are able to break the full key of Grain-128 by requesting, and storing, the first bit of keystreams corresponding to 2^{59} chosen IVs. By nulling state bits, they are able to significantly simplify the equations that need to be solved in order to find the key bits. More recently, Dinur et al. [DGP$^+$11] improved this attack, both in terms of time complexity and the number of keys that could be attacked.

Both attacks exploit partly the low degree of g, and partly the choice of $x_4 = b_{i+95}$ and $x_8 = s_{i+95}$ used to calculate the preoutput value in Grain-128. These bits are multiplied together, but are very similar during the initialization phase when the suppressed preoutput bit is fed back to the registers. To mitigate this weakness, Grain-128a uses $x_4 = b_{i+95}$ and $x_8 = s_{i+94}$ to calculate the preoutput value.

6.6 HARDWARE COMPLEXITY

Grain-128a can be constructed using flip flops, XORs, etc., and like in Chapter 5, this can be used to estimate the gate counts required for an implementation. The list of the gate counts that have been used in deriving hardware numbers are found in Table 5.1.

Table 6.1 gives the gate counts for the larger building blocks of Grain-128a, as well as the total gate count for the entire Grain-128a. Basic combinatorics, e.g., the multiplexers needed to select between, e.g., initialization of the preoutput generator, initialization of the authentication, and keystream generation, have not been included. The few extra XORs needed during initialization have also been left out.

6.6.1 DIFFERENT TAG SIZES

It is possible to make the authentication mechanism consume less hardware resources, at the cost of increasing the success probability of the attack. The intuitive approach to producing a shorter tag is to simply chop the original one, discarding some bits. As Grain-128a aims for large flexibility and efficiency, the construction allows to not calculate these bits in the first place.

Note that care must be taken to discard the correct preoutput bits as to not affect the calculations of the remaining part of the authentication tag as well as the encryption keystream.

6.6.2 THE INCREASE OF HARDWARE FROM GRAIN-128

Let us compare the hardware cost of an implementation that produces one bit per clock to that of Grain-128. This is the smallest possible Grain-128, and the increase in this cost should give us an idea of the cost of the extra flexibility and security added in Grain-128a.

Grain-128 required 2133 gates to implement the basic design, producing one bit of keystream per clocking. Grain-128a without authentication requires 2145.5 gates, according to Table 6.1, meaning the increased hardware is neg-

Table 6.1: The estimated gate count in an actual implementation. The total given for the w-bit MAC only relates to the authentication mechanism itself, not the preoutput generator needed to actually run it. The cost of the »accumulating logic« of the authentication mechanism is the same for speeds 1x and 2x—one implementation makes use of this logic every second clocking, and the other on each one.

Building block	Speed increase					
	1x	2x	4x	8x	16x	32x
LFSR	1024	1024	1024	1024	1024	1024
NFSR	1024	1024	1024	1024	1024	1024
f	12.5	25	50	100	200	400
g	49.5	99	198	396	792	1584
Preoutput function	35.5	71	142	284	568	1136
Accumulator	$8w$	$8w$	$8w$	$8w$	$8w$	$8w$
Shift register	$8w$	$8w$	$8w$	$8w$	$8w$	$8w$
Accumulating logic	$3.5w$	$3.5w$	$7w$	$14w$	$28w$	$56w$
Total (only enc.)	2145.5	2243	2438	2828	3608	5168
Total (only w-bit MAC)	$19.5w$	$19.5w$	$23w$	$30w$	$44w$	$72w$
Total (enc. + 32-bit MAC)	2769.5	2867	3174	3788	5016	7472

ligible. Looking instead at the version that authenticates and produces one bit of keystream (two bits of preoutput) per clocking, the number of gates is 2867. This is a mere 34 per cent increase. Note that while Grain-128 initialized in 256 clockings, authenticating Grain-128a in 2x mode generates keystream after only $(256 + 64)/2 = 160$ clockings.

6.7 THE GRAIN FAMILY OF STREAM CIPHERS

As by the publication of [ÅHJM11], Grain-128 is no longer recommended. The designers instead recommend Grain-128a for 128 bits security. While the 80-bit version, Grain v1, suffers from the deficiency addressed in Subsection 6.5.3, the practical impact is marginal. Grain v1 is still recommended for 80 bits security.

Table 6.2: Test vectors for Grain-128a used without authentication.

	Test vector I	Test vector II
Key	`0000000000000000`	`0123456789abcdef`
	`0000000000000000`	`123456789abcdef0`
IV	`0000000000000000`	`0123456789abcdef`
	`00000000`	`12345678`
Keystream	`c0207f221660650b`	`f88720c13f46e6a4`
	`6a952ae26586136f`	`3c07eeed89161a4d`
	`a0904140c8621cfe`	`d73bd6b8be8b6b11`
	`8660c0dec0969e94`	`6879714ebb630e0a`
	`36f4ace92cf1ebb7`	`4c12f0399412982c`

6.8 TEST VECTORS

Test vectors for Grain-128a without authentication are given in Table 6.2. Test vectors for Grain-128a with authentication are given in Table 6.3. While each key–IV is used to authenticate five different plaintexts, this is solely for the purpose of giving several test vectors.

Plaintext 0, $\boldsymbol{p}^0$, is the plaintext of length 0. Plaintexts 1 and 2 are both of length 1: $\boldsymbol{p}^1 = \boldsymbol{p}^2 + 1 = 0$. These three plaintexts are supposedly helpful in verifying the initialization and basic functioning of the MAC algorithm.

Plaintext 3 is of length 20 and its hexadecimal representation is $\boldsymbol{p}^3 =$ `12340`. Plaintext 4 is 41 bits long and can, using slightly abused notation, be represented as $\boldsymbol{p}^4 =$ `123456789e8`. To avoid any confusion we also give the bit representation of $\boldsymbol{p}^4$: 00010010001101000101011001111000100111101.

The test vectors named »macstream« are the sequences shifted into the register, i.e., the preoutput bits $y_{65}, y_{67}, \ldots$.

The 16-bit tag for $\boldsymbol{p}^4$ authenticated using the key and IV in the rightmost column is $\boldsymbol{t}^{(16)} =$ `b196`.

6.9 CONCLUSION

A new stream cipher, Grain-128a, has been presented. The design is a new member in the family of Grain stream ciphers. The size of the key is 128 bits and the size of the IV is 96 bits. The design parameters have been chosen based on theoretical arguments for various possible attacks, and in light of known observations on older members of the family. Grain-128a is very well-suited for hardware environments, and the speed of the cipher can be

Table 6.3: Test vectors for Grain-128a used with authentication.

	Test vector I	Test vector II
Key	`0000000000000000` `0000000000000000`	`0123456789abcdef` `123456789abcdef0`
IV	`8000000000000000` `00000000`	`8123456789abcdef` `12345678`
Preoutput stream	`564b362219bd90e3` `01f259cf52bf5da9` `deb1845be6993abd` `2d3c77c4acb90e42` `2640fbd6e8ae642a`	`7f2acdb7adfb701f` `8d2083b3c32b43f1` `962b3dcabf679378` `db3536bfc25bed48` `3008e6bcb395a156`
Accumulator	`564b3622`	`7f2acdb7`
Shift register	`19bd90e3`	`adfb701f`
Keystream	`0d2b1f2ebc83da7e` `6658ee3150f9ef47`	`a49d971c976bf596` `b45f93e242ded8c1`
Macstream	`1cdbc7f1e52da547` `36fa252828de82a0`	`3015919d61787b5c` `d7678db840a6571e`
Tag($\boldsymbol{p}^0$)	`4ff6a6c1`	`d2d1bda8`
Tag($\boldsymbol{p}^1$)	`653017e4`	`24dc2d89`
Tag($\boldsymbol{p}^2$)	`7c8d8707`	`89275d96`
Tag($\boldsymbol{p}^3$)	`522ab34f`	`379d2899`
Tag($\boldsymbol{p}^4$)	`4b7821c9`	`9226b196`

increased very easily at the expense of extra hardware. Grain-128a is slightly more expensive in hardware than Grain-128, but offers better security and the possibility of adding authentication. To the best of our knowledge, there is no 128-bit cipher offering the same level of security as Grain-128a and a smaller gate count in hardware.

7 Cryptanalysis of the Stream Cipher BEAN

BEAN [KOJL09] is a stream cipher that has some superficial similarities with Grain (cf. Section 2.12): the NFSR and the LFSR are replaced by two FCSRs. There is a sound motivation behind this idea since the LFSR in Grain is used to provide large period and to guarantee random-like properties while the NFSR is used to provide nonlinearity. An FCSR combines both these properties, while still being efficient in hardware. Thus, BEAN has two shift register components, both providing nonlinearity, large period and random-looking sequences.

There have been several FCSR-based stream ciphers proposed in the literature. One notable design is F-FCSR-H [AB04] [ABL06] which was selected for the final portfolio in the eSTREAM project. The hardware performance is good and the design is very simple in that it only uses a linear filter together with one FCSR. The performance of, and interest in, the F-FCSR-H stream cipher made it evident that FCSRs are attractive building blocks for stream ciphers, even though F-FCSR-H was later cryptanalyzed in [HJ08]. The attack on F-FCSR-H in [HJ08] took advantage of the fact that 8 keystream bits were produced in each clocking of the register. As BEAN produces 2 keystream bits in 3 FCSR updates, that attack is not applicable to BEAN. Further, the attack in [HJ08] used an observation on the behavior of FCSRs in Galois architecture, while BEAN uses Fibonacci FCSRs. Although a relation between states in the two architectures can be found in [FMS08], this will not be used here.

In this chapter two attacks are presented on the BEAN stream cipher. While the design idea can be well-motivated, as described above, the design of the output function allows for attacks. First, a distinguishing attack is presented based on the low correlation immunity order of the output function. This attack is very efficient as it can distinguish the keystream of BEAN from a random sequence using only about 2^{18} keystream bits, or 32 KiB[1]. Second, a key recovery attack is given that is based on information leakage in the output function. By guessing a carefully chosen subset of the state bits, a portion of the keystream can be used to verify the guess, resulting in a divide-and-conquer attack on the state. A trade-off between the computational resources and keystream required is presented. As an example, the state can be recovered using 6 KiB of keystream and 2^{73} computations, each computation being as complex as testing one key.

With these attacks as background, the specific design choices made in the BEAN stream cipher are discussed and some ideas on how the cipher can be improved in the future are presented. The results in this chapter can thus be seen as a foundation for studying the security of BEAN-like stream ciphers.

While the BEAN specification [KOJL09] is sometimes ambiguous, for example as to whether the FCSRs are really FCSRs or merely LFSRs, the reference implementation can be used to clarify such uncertainties. In this chapter, we use a very conservative approach and always make the more reasonable, secure interpretation in these cases. It is known that constructing a decent stream cipher using only LFSRs and an output function (i.e., a nonlinear combiner) is practically impossible (cf. Section 4.6). Indeed, if BEAN is implemented with LFSRs instead of FCSRs, independent work [PL11] has shown that it is susceptible to algebraic attacks. Similarly, correlation attacks would be a natural approach to attack such weakened versions. We will deal exclusively with the stronger version of BEAN that results from always making the sane choice in case of ambiguities.

This chapter is organized as follows. Section 7.1 describes the stream cipher BEAN, before Section 7.2 gives a distinguishing attack. Section 7.3 outlines the standard brute force attack to establish an attack cost to compare subsequent findings to. Section 7.4 then describes how to find the key slightly faster than brute force, and Section 7.5 introduces a time–data trade-off that needs more keystream but is significantly faster. Section 7.6 outlines what needs to be reconsidered in the BEAN design, before Section 7.7 concludes the chapter.

7.1 BEAN SPECIFICATION

BEAN is very similar to Grain in that it consists of two shift registers and one output function, taking input from both registers. The size of the secret key is 80 bits. While Grain uses one LFSR and one NFSR, BEAN instead has two FCSRs, both implemented in Fibonacci architecture. These are denoted FCSR-I and FCSR-II, see Figure 7.1. An overview of the design of BEAN is

[1] A byte (B) is 8 bits, and a KiB (»kibibyte«) is precisely 2^{10} B. Similarly, a MiB (»mebibyte«) is 2^{20} B, while a GiB (»gibibyte«) is 2^{30} B. [Int08]

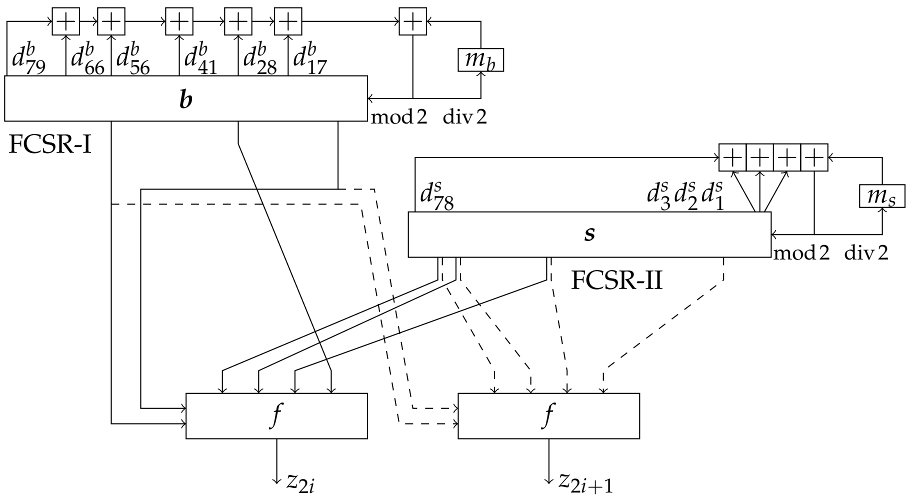


Figure 7.1: An overview of BEAN. Non-zero d_i's are named next to the corresponding taps.

given in the design document [KOJL09]. In order to avoid any ambiguity or misinterpretation of the specification, the implementation provided by the designers has been studied to establish the specification below.

7.1.1 KEYSTREAM GENERATION

Both FCSRs are 80 bits in size, i.e., the same as the key size. The state of FCSR-I at time instance i is denoted by $\mathbf{B}^i$ and correspondingly the state of FCSR-II at time instance i by $\mathbf{S}^i$. Thus,

$$\begin{aligned}\mathbf{B}^i &= (\mathbf{b}^i, m_b^i) = (b_i, \ldots, b_{i+79}, m_b^i),\\ \mathbf{S}^i &= (\mathbf{s}^i, m_s^i) = (s_i, \ldots, s_{i+79}, m_s^i).\end{aligned}$$

FCSR-I is updated according to

$$\begin{aligned}\sigma_b^i &= b_{i+62} + b_{i+51} + b_{i+38} + b_{i+23} + b_{i+13} + b_i + m_b^i,\\ b_{i+80} &= \sigma_b^i \bmod 2,\\ m_b^{i+1} &= \sigma_b^i \operatorname{div} 2.\end{aligned}$$

Similarly, FCSR-II is updated as

$$\begin{aligned}\sigma_s^i &= s_{i+78} + s_{i+77} + s_{i+76} + s_{i+1} + m_s^i,\\ s_{i+80} &= \sigma_s^i \bmod 2,\\ m_s^{i+1} &= \sigma_s^i \operatorname{div} 2.\end{aligned}$$

Table 7.1: The Boolean function f used in BEAN. The input $\boldsymbol{x}$ is split as $\boldsymbol{x} = \boldsymbol{x}^{\mathrm{r}}||\boldsymbol{x}^{\mathrm{c}}$, $|\boldsymbol{x}^{\mathrm{r}}| = 2$, $|\boldsymbol{x}^{\mathrm{c}}| = 4$. The output $f(\boldsymbol{x})$ is then found at row $(\boldsymbol{x}^{\mathrm{r}})$ and column $(\boldsymbol{x}^{\mathrm{c}})$.

		$(\boldsymbol{x}^{\mathrm{c}})$															
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$(\boldsymbol{x}^{\mathrm{r}})$	0	0	0	1	0	0	1	1	0	0	1	0	1	1	1	0	1
	1	1	0	1	0	0	0	1	1	1	0	0	1	0	1	0	1
	2	0	0	1	0	1	1	1	1	1	1	0	0	0	0	1	0
	3	0	1	0	1	1	1	0	0	0	1	1	1	1	0	0	0

That is, the sets of tap positions are given by

$$\begin{aligned} \mathcal{T}_b &= \{17, 28, 41, 56, 66, 79\}, \\ \mathcal{T}_s &= \{1, 2, 3, 78\}. \end{aligned}$$

Thus, m_b and m_s are realized using 3 and 2 bits, respectively (cf. Proposition 2.3).

A Boolean function $f(x_0, x_1, \ldots, x_5)$ is used to produce the keystream. This is given as a 6-to-4-bit Sbox in the original description [KOJL09] but as only one bit from each word is taken as output, it is easier to analyze it if it is considered as a 6-to-1 Boolean function. The keystream bits $z_0, z_1, \ldots$ are then given by

$$z_{2i} = f(b_{i+23}, b_{i+73}, s_{2i+5}, s_{2i+9}, s_{2i+29}, b_{i+51}), \tag{7.1}$$

$$z_{2i+1} = f(b_{i+23}, b_{i+73}, s_{2i+6}, s_{2i+10}, s_{2i+30}, s_{2i+68}). \tag{7.2}$$

The keystream generation algorithm is given in Algorithm 7.1.

The algebraic normal form of f is

$$\begin{aligned} f(\boldsymbol{x}) = & x_0 + x_3 + x_0x_1 + x_0x_2 + x_0x_3 + x_0x_4 + x_1x_4 + x_1x_5 + \\ & x_2x_3 + x_2x_4 + x_2x_5 + x_3x_5 + x_4x_5 + x_0x_1x_4 + \\ & x_0x_1x_5 + x_0x_2x_3 + x_0x_2x_5 + x_0x_3x_4 + x_0x_4x_5 + \\ & x_1x_2x_4 + x_1x_2x_5 + x_1x_3x_4 + x_1x_3x_5 + x_1x_4x_5 + \\ & x_2x_3x_4 + x_2x_3x_5 + x_3x_4x_5 + x_0x_1x_2x_4 + x_0x_1x_2x_5 + \\ & x_0x_1x_3x_4 + x_0x_1x_3x_5 + x_0x_1x_4x_5 + x_0x_2x_3x_4 + \\ & x_0x_2x_3x_5 + x_0x_2x_4x_5 + x_0x_3x_4x_5 + \\ & x_0x_1x_2x_3x_5 + x_0x_1x_3x_4x_5. \end{aligned}$$

The input $\boldsymbol{x}$ to the Boolean function f is split as $\boldsymbol{x} = \boldsymbol{x}^{\mathrm{r}}||\boldsymbol{x}^{\mathrm{c}}$, where $|\boldsymbol{x}^{\mathrm{r}}| = 2$ and $|\boldsymbol{x}^{\mathrm{c}}| = 4$. Referring to Table 7.1, the output $f(\boldsymbol{x})$ is then found at row $(\boldsymbol{x}^{\mathrm{r}})$

Algorithm 7.1. BEAN keystream generation

```
    Input: initialized FCSR-I and FCSR-II and keystream length $n$
    Output: $n$ bits of keystream

 1. Set $i = 0$.
 2. Output an even-numbered keystream bit.
 3. Set $i \leftarrow i + 1$.
 4. Exit if $i = n$.
 5. Update FCSR-II.
 6. Output an odd-numbered keystream bit.
 7. Set $i \leftarrow i + 1$.
 8. Exit if $i = n$.
 9. Update FCSR-I and FCSR-II.
10. Go to step 2.
```

and column ($\boldsymbol{x}^{\mathrm{c}}$). In the sequel, we will refer to the function f and its input by referring to »rows« and »columns.« This approach arises naturally as the row is always selected by **B**, while the column is (almost) exclusively selected by **S**.

7.1.2 BEAN INITIALIZATION

BEAN is initialized by loading the key $\boldsymbol{k} = (k_0, k_1, \ldots, k_{79})$ into the registers as

$$\begin{aligned} b_i &= k_{i+81}, \quad i = -81, \ldots, -2, \\ s_i &= k_{i+81}, \quad i = -81, \ldots, -2. \end{aligned}$$

The carries are set to $m_s^{-81} = m_b^{-81} = 0$. Then, both FCSRs are initialized by updating them 81 times. After initialization, the registers contain $b_0, b_1, \ldots, b_{79}$ and $s_0, s_1, \ldots, s_{79}$ respectively and keystream is ready to be produced according to Algorithm 7.1.

Observe that the FCSRs are initialized at time »−81.« This is only a matter of notational convenience.

7.2 A DISTINGUISHING ATTACK ON BEAN

In this section we give a very efficient distinguishing attack on BEAN. The designers performed several statistical tests, provided by NIST [NIS10], on BEAN. The keystream showed no deviation from random behavior when the tests were applied to sequences of about 2^{23} bits. However, these tests are generic and do not take the internal structure of the stream cipher into account. Taking the structure into account, we show that with as few as 2^{18} keystream bits, a deviation from random can be observed.

Recall the Boolean output function given in Table 7.1. Analyzing the Boolean function and its Walsh transform, the following properties can be found.

Balanced	Yes
Algebraic Degree	5
Resiliency	0
Nonlinearity	22
Best Linear Approximations	$\ell_1(f)$: $1 + x_0 + x_3 + x_4$, $\ell_2(f)$: $x_1 + x_2 + x_4 + x_5$

Using $\ell_1(f)$ one can, e.g., write

$$\mathbf{Pr}\left[z_{2i} = b_{i+23} \oplus s_{2i+9} \oplus s_{2i+29}\right] = \frac{1}{2} - 5 \cdot 2^{-5}. \tag{7.3}$$

The common approach in a linear distinguishing attack is then to find a relation in the b_i and s_i variables that sum to zero, leaving only an expression involving keystream bits, see, e.g., [CHJ02] [HJB08]. However, the FCSRs are nonlinearly updated and it is very difficult to find such a relation unless the full period is considered. As the period is large, this attack is not applicable to BEAN. However, as the resiliency of the Boolean function is 0, there are biased linear approximations of weight 1. Studying the Walsh transform of the Boolean function, it can be found that

$$\mathbf{Pr}\left[f(x) = x_2 + 1\right] = \frac{1}{2} + 2^{-5}, \tag{7.4}$$

and similarly,

$$\mathbf{Pr}\left[f(x) = x_4 + 1\right] = \frac{1}{2} + 2^{-5}. \tag{7.5}$$

From Equations 7.1 and 7.4,

$$\mathbf{Pr}\left[z_{2i+24} = s_{2i+29} \oplus 1\right] = \frac{1}{2} + 2^{-5},$$

and in the same way

$$\mathbf{Pr}\left[z_{2i} = s_{2i+29} \oplus 1\right] = \frac{1}{2} + 2^{-5}.$$

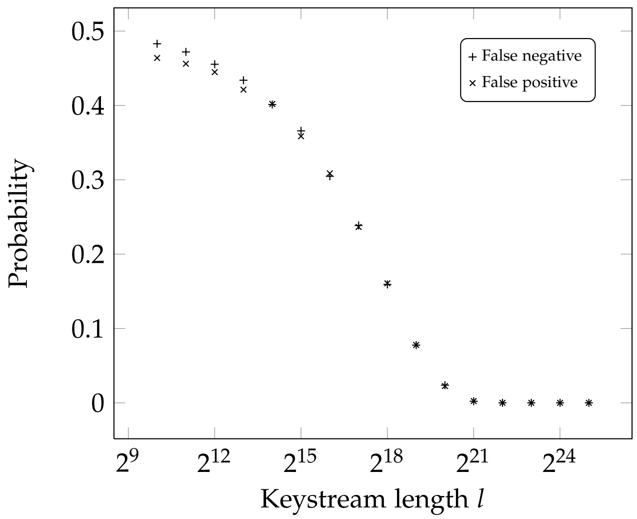


Figure 7.2: Using the distinguisher on longer keystreams results in smaller probabilities of false positives. Each trial was conducted over 2^{15} different keys.

Combining the last two equations and assuming independence yields

$$\mathbf{Pr}\left[z_{2i} = z_{2i+24}\right] = \frac{1}{2} + 2^{-9}.$$

The same holds for $z_{2i+1} = z_{2i+25}$, so

$$\mathbf{Pr}\left[z_i + z_{i+24} = 0\right] = \frac{1}{2} + 2^{-9}.$$

Thus, the distinguishing attack requires only about $\varepsilon^{-2} = 2^{18}$ keystream bits to succeed. Observe that the samples, e.g., $z_0 + z_{24}$ and $z_{24} + z_{48}$, are clearly dependent. However, the small time and data requirements allow for a thorough simulation of the attack. Figure 7.2 shows how the error probabilities depend on the keystream length l when the distinguisher is applied to equally many BEAN keystreams and pseudorandom bitstreams. The distinguisher uses a threshold $\theta = n\left(\frac{1}{2} - \frac{\varepsilon}{2}\right)$, where $n = l - 24$ is the number of samples constructed from the keystream. (The differing false positive and negative probabilities for very small keystream lengths is due to the low »resolution:« we try to distinguish between distributions centered around, e.g., 500 and (approximately) 498 using a threshold (approximately) 499.)

7.3 A STANDARD BRUTE FORCE KEY RECOVERY

In the remainder of the chapter, several known-plaintext key-recovery attacks will be derived.

Before giving the details of the attacks, the cost of the generic attack must be known. Due to this, a reasonable measure will be presented on the time (computations) spent in a brute force approach to find the secret key behind a given keystream.

We consider the initialization to consist of 80 clockings, because this is easier to work with than 81. This simplification has virtually no effect on any of the measurements in this chapter.

From the specification in Section 7.1, it can be seen that one specific key bit never affects the contents of **S** as $d_{79} = 0$: k_0 is thrown out of the state in the very first update without influencing the feedback. Furthermore, **S** never affects **B**. These properties will we used in the »trivial« brute force attack; they are not necessarily flaws in and of themselves, and any attacker attempting a standard brute force would probably use this property: to test two keys that only differ in k_0, **B** must be initialized twice, but **S** only once.

Thus, during a standard brute force attack on BEAN, one expects to perform $2^{79} \cdot 80 + 2^{78} \cdot 80 = 2^{80} \cdot 60$ FCSR updates. This will be the reference point in this chapter. When it is claimed that an attack requires time, e.g., 2^{79}, it requires $2^{79} \cdot 60$ FCSR updates (and some additional work that is negligible in comparison).

The remainder of this chapter will use a slightly modified definition of $\mathbf{s}^i$:

$$\mathbf{s}^i = (s_{i+1}, \ldots, s_{i+79}).$$

We redefine $\mathbf{S}^i$ correspondingly. The new FCSR produces the same sequence except for k_0 (which never reaches the output function), but the state is one bit smaller.

7.4 AN IMPROVED KEY-RECOVERY ATTACK

In this section, structural flaws are presented and put together into a key recovery faster than brute force, i.e., cheaper than $2^{80} \cdot 60$ FCSR updates. The fundamental observation is that the 79-bit key for **S** can be brute forced before taking care of **B**. To see this, assume a guess for $\mathbf{S}^j$, meaning that this FCSR can be tracked for all future time. One can then observe the keystream and try to find contradictions.

Example 7.1 Without loss of generality, consider $2i = 0$. Let

$$(s_5, s_6, s_9, s_{10}, s_{29}, s_{30}, s_{68}) = (1, 0, 0, 0, 0, 0, 0).$$

Table 7.2: Integer representations of nine-bit vectors of state and output that cannot appear in BEAN.

9	13	24	34	41	45	55
56	66	87	94	136	156	157
163	168	170	178	188	189	192
195	207	210	256	259	263	267
271	274	277	282	329	333	344
380	385	386	394	400	406	407
414	415	456	476	477	480	493

This means that the column for z_0 is $(1,0,0,\cdot)$, where the last bit is unknown, while z_1 comes from column $(0,0,0,0)$. If $z_1 = 1$ is observed, the row is $(1,0)$. If also $z_0 = 1$ is observed, the row *cannot* be $(1,0)$, meaning there is a contradiction. $\square$

To summarize this example, by looking at nine bits, one can conclude whether or not they pose a contradiction. We have implemented this. By using a small lookup table of 2^9 bits, we only need to perform 12 FCSR updates on average before rejecting a bad guess. Table 7.2 lists the impossible state–output combinations that can be used to reject guesses of $\mathbf{S}$.

For every vector $\boldsymbol{v} = (s_{2i+5}, s_{2i+6}, s_{2i+9}, s_{2i+10}, s_{2i+29}, s_{2i+30}, s_{2i+68}, z_{2i}, z_{2i+1})$, Table 7.2 contains $(\boldsymbol{v})$ if and only if those bits pose a contradiction. As an example, for $\boldsymbol{v} = (1,0,0,0,0,0,0,1,1)$, which corresponds to the specific numerical values studied in Example 7.1, $(\boldsymbol{v}) = 385$.

The following summarizes the above as it is a central discovery relating to BEAN and will be used later in the chapter.

Claim 7.1 Given keystream and a guess for $\mathbf{S}^j$, the expected number of FCSR updates needed before rejecting this guess is 20. The additional work is negligible, as it only amounts to a small number of table lookups.

In Claim 7.1 the expected cost of the verification has been exaggerated. This is because the number 20 is easier to work with, but can also be seen as a buffer for implementation overhead.

There is obviously one guess which will never be rejected, no matter how many times the FCSR is updated: the correct guess. Thus, an implementation may use a maximum number s_{lim} of FCSR updates, after which the correct state is assumed to have been found. During simulations, no incorrect guess required more than 400 FCSR updates. From this, it seems reasonable that, e.g., $s_{\text{lim}} = 2^{10}$ will give a very low probability of false positives.

The strategy of the attack is now clear: guess the 79 bits of the key used in s. Each guess is expected to be cheap to verify according to Claim 7.1, and once 79 bits of the key are found, the remaining key bit is trivial to brute force through **B** with additive and negligible cost. The cost of the attack is $2^{78} \cdot (80 + 20) \approx 2^{78.7} \cdot 60$ FCSR updates. This attack takes less than half the time of a brute force and requires no significant amount of memory.

7.5 A TIME–DATA TRADE-OFF

The major cost of the above key recovery lies in the guess of key for **S**: this requires 2^{79} initializations of **S** and some clockings in order to determine whether the state is correct or not. One observation is that rather than initializing the FCSR 2^{79} times, it might be faster to recover the state in the middle of operation and then (only once, for the correct guess) revert the state back to the key. This will be done below.

For this approach to improve the above attack, guesses must be correct with some probability better than 2^{-79}.

This requires making informed guesses from the keystream. Note that guessing **S** in the middle of operation does not allow the attacker to assume $m_s = 0$, so these two bits have to be recovered as well. Thus, it appears as if 81 bits must be guessed. All in all, a 164-bit state must be recovered faster than 2^{79}. To succeed with this, several observations will we presented below.

7.5.1 GUESSING S

In Subsection 7.5.2, it will be seen that it is straightforward to recover **B** once **S** is known, and Subsection 7.5.4 shows how to revert the state to find the key. This section is dedicated to what turns out to be the most expensive part of the attack, namely recovering **S**.

Given a guess of $\mathbf{s}$, there are four possibilities for the two bits of m_s. One can then use Claim 7.1 to reject three or four guesses. However, assume that the carry m_s and the leftmost bit of $\mathbf{s}$ have been guessed. There are in total eight such guesses. But as the new state is calculated using

$$\sigma_s^i = s_{i+78} + s_{i+77} + s_{i+76} + s_{i+1} + m_s^i,$$

fixing $(s_{i+76}, s_{i+77}, s_{i+78})$, one can only produce $|\mathcal{T}_s| + 1 = 5$ distinct **S** in the next time step. Thus, rather than guessing 79 bits and guessing the carry 4 times, one can guess 78 bits and then the carry (rather, the rest of the information) 5 times. As an example of this, note that $(s_{i+1}, m_s^i) = (1, 1)$ produces the same new state as $(s_{i+1}, m_s^i) = (0, 2)$, where we have moved the 1 from s_{i+1} into the carry.

The above can be generalized as the following lemma. Its precise formulation is specific to the special nature of **S**, where s_{i+0} is included in neither the update rule nor the redefined state, but it can be easily stated for the more common type of FCSR.

Lemma 7.1 Let $\mathbf{S}$ be an FCSR of length n using taps $\mathcal{T}_s$ and with state $\mathbf{S}^i = (s_{i+1}, \ldots, s_{i+n-1}, m_s^i)$. Let $l_0 = \max \mathcal{T}_s = n-2$ and $l_1 = \max \mathcal{T}_s \backslash \{l_0\}$. For each j, $0 < j \leq l_0 - l_1$ and each state $(s_{i+1}, \ldots, s_{i+j}, s_{i+j+1}, \ldots, s_{i+n-1}, m_s^i)$, there is an equivalent state $(0, \ldots, 0, s_{i+j+1}, \ldots, s_{i+n-1}, m_s^i + \sum_{l=0}^{j-1} 2^l s_{i+1+l})$. The states are equivalent in the sense that they produce the same future state

$$(s_{i+j+1}, \ldots, s_{i+j+n-1}, m_s^{i+j}).$$

Thus, having guessed the bits $(s_{i+j+1}, \ldots, s_{i+n-1})$, one can assume that $s_{i+1} = \ldots = s_{i+j} = 0$. It then remains to find the carry $m_s^i \in \{0, \ldots, 2+2^j\}$ where the set of possible m_s^i has been extended.

This lemma is highly applicable to $\mathbf{S}$ in BEAN as the first two taps are $l_0 - l_1 = 78 - 3 = 75$ bits apart. This means one can guess $(s_{i+76}, \ldots, s_{i+79})$, assume $(s_{i+1}, \ldots, s_{i+75}) = \mathbf{0}$ and then recover the equivalent carry at cost $3 + 2^{75} \approx 2^{75}$. In total, this attack would require work $2^4(3 + 2^{75}) \approx 2^{79}$. Obviously, the gain of applying this trick to yet another bit quickly becomes unimpressive. Certainly, one can also claim that $2^{54}(3 + 2^{25}) \approx 2^{79}$, so the marginal gain of applying this to 75 rather than 25 bits is negligible. Further, any bits that have been set to zero in this way cannot be tapped for the output function. This suggests that using Lemma 7.1 to move a large number of bits into m_s^i in this way, requires a large number of FCSR updates.

Four sets of bit indices are used in the recovery of $\mathbf{S}$, partitioning the 79-bit state of $\mathbf{s}^{2i+26}$:

$$\begin{aligned}
\mathcal{A} &= \{2i + 30 + 2j \mid j = 0, 1, \ldots, 18\}, & |\mathcal{A}| &= 19,\\
\mathcal{B} &= \{2i + 68 + 2j \mid j = 0, 1, \ldots, 18\}, & |\mathcal{B}| &= 19,\\
\mathcal{C} &= \{2i + 31 + 2j \mid j = 0, 1, \ldots, 36\}, & |\mathcal{C}| &= 37,\\
\mathcal{D} &= \{2i + 26, \ldots, 2i + 29\}, & |\mathcal{D}| &= 4.
\end{aligned}$$

The general idea is to brute force bits of $\mathcal{B} \cup \mathcal{C}$. Due to the linear approximation

$$\mathbf{Pr}\left[f(\cdot) = x_4 + x_5\right] = \frac{38}{64},$$

which yields

$$\mathbf{Pr}\left[s_{2i+30} + s_{2i+68} = z_{2i}\right] = \frac{38}{64}, \tag{7.6}$$

the bits indexed by $\mathcal{A}$ can be correctly guessed with probability

$$\left(\frac{38}{64}\right)^{19} \approx 2^{-14.3},$$

Table 7.3: The bits s_{2i+l}, $l \in \{30, 31, \ldots, 104\}$ that affect keystream bits $(z_{2(i+j)}, z_{2(i+j)+1})$, $j \geq 0$. Bits are only listed the first time they appear.

j	l	j	l	j	l
0	31, 32, 35, 36, 55, 56, 94	8	51, 52, 71, 72	16	87, 88
1	33, 34, 37, 38, 57, 58, 96	9	30, 53, 54, 73, 74	17	89, 90
2	39, 40, 59, 60, 98	10	75, 76	18	91, 92
3	41, 42, 61, 62, 100	11	77, 78	19	93
4	43, 44, 63, 64, 102	12	79, 80	20	95
5	45, 46, 65, 66, 104	13	81, 82	21	97
6	47, 48, 67, 68, 103	14	83, 84	22	99
7	49, 50, 69, 70, 101	15	85, 86		

when the bits indexed by $\mathcal{B}$ are correctly guessed. Simulations over 2^{35} different states has given this probability as approximately $2^{-14.4}$.

All bits indexed by $\mathcal{D} = \{2i + 26, \ldots, 2i + 29\}$ will be assumed to be zero, and m_s^{2i+25} is guessed $3 + 2^4$ times.

The above together with Claim 7.1 suggest that the entire $\mathbf{S}^{2i+25}$ can be recovered using an expected $2^{14.5} 2^{56}(3 + 2^4) \cdot 20$ number of FCSR updates and data $\approx 2^{14.5} \cdot 2$ which is approximately 6 KiB. (The factor 2 shows up because the attack strategy is applied to windows of s beginning with even-numbered bits.)

It is not obvious that Claim 7.1 holds in this attack. The properties of f used to derive it »conflict« with Equation 7.6. The attack strategy is to guess states that are more likely than uniformly random guesses, and so it is more rare that impossible state-keystream combinations are obtained than in the uniform case. However, simulations suggest that on average 16 FCSR updates are required to reject a bad guess, so Claim 7.1 can be used also with these biased guesses.

From an implementation perspective, it should be noted that there is a possibility of early aborts; from simulations, a biased guess of

$$(s_{2(i+j)+5}, s_{2(i+j)+6}, s_{2(i+j)+9}, s_{2(i+j)+10}, s_{2(i+j)+29}, s_{2(i+j)+30}, s_{2(i+j)+68}),$$

$j = 0, 1, \ldots, 5$, can be rejected with probability .05. This can be used six times, for $j = 0, 1, \ldots, 5$ without clocking the FCSR, i.e., without introducing any memory guess. Thus, having guessed a state, it only happens with probability approximately $.95^6 \approx .75$ that the memory be guessed. Table 7.3 lists when state bits begin to affect the output.

The complete algorithm for recovering $\mathbf{S}$ is given as Algorithm 7.2. When a state equivalent to the true state $\mathbf{S}^{2i+27}$ has been found, by Lemma 7.1, the

Algorithm 7.2. Recover **S**

```
Input: keystream
Output: corresponding FCSR state $\mathbf{S}^{2i+30}$ for some $i$

1. Set $i = 0$.
2. Assume $s_j = 0, j \in \mathcal{D}$.
3. Guess the bits indexed by $\mathcal{B} \cup \mathcal{C}$.
4. For each guess of the bits indexed by $\mathcal{A}$ using Equation 7.6 {
     For each $m_s \in \{0, 1, \ldots, 2 + 2^4\}$ {
       If no contradiction found between current guess and keystream {
         Load the guess.
         Clock three times.
         Return as recovered FCSR state $\mathbf{S}^{2i+30}$.
       }
     }
   }
5. Increase $i$ by one and go to step 2.
```

true $\mathbf{S}^{2i+30}$ can be produced by clocking three times.

A similar early-abort strategy as used above can be used after the memory has been introduced. If, for a particular guess of $\boldsymbol{s}$, each of the 19 memory guesses m_s are rejected in r_{m_s} clocks, $0 \le m_s < 3 + 2^4$, the state guess must have been wrong in at least one of the bits affecting the output within $\max_l r_l$ clocks. Simulations suggest that such an algorithm would be at most 10% faster than the one described here.

7.5.2 FINDING B GIVEN S

Assume that the correct **S** has been recovered, so that this FCSR can be tracked for all future time. The next part of the attack is to recover **B**. The strategy is to first derive about 20 bits somewhere in a window of eighty bits of $\boldsymbol{b}$. A concluding brute force will require guessing about $80 + 3 - 20 = 63$ bits, so that the total cost is negligible compared to the previous parts of the attack.

Note that for z_{2i+1}, it is known precisely which *column* is used in f. There are 10 unbalanced columns in Table 7.1, and with probability $\frac{1}{4}$, such a column uniquely identifies a row, i.e., two bits of $\boldsymbol{b}$. Thus two bits of $\boldsymbol{b}$ can be recovered with probability

$$\frac{10}{16} \cdot \frac{1}{4} = \frac{5}{32}.$$

More likely, with probability $\frac{6}{16}$, one bit of $\boldsymbol{b}$ is learned, either as b_j, or as a parity bit $b_{j_0} + b_{j_1}$. While nonlinear equations can also be derived, those are ignored here.

This part has been implemented. We have found that with probability .5, at least 20 bits can be recovered. Similarly, with probability .022, at least 30 bits can be recovered. With bad luck, only some small number of bits from $\boldsymbol{b}$ can be recovered—one can then fast-forward in $\mathbf{S}$ and $\boldsymbol{z}$ and make another try. Within just a few trials, enough bits can be found to make the total recovery cost of $\mathbf{B}$ negligible.

7.5.3 A GENERAL CONCLUSION

If $j > 2$, then $(3 + 2^j)\frac{20}{60} < 2^{j-1}$. Therefore, assume that Lemma 7.1 is being used on $j > 2$ bits. If one is able to guess b of the remaining bits with probability 2^{-l}, an attack can be constructed that requires 2^{l+1} data and time

$$2^l 2^{79-j-b}(3 + 2^j)\frac{20}{60} < 2^{78+l-b},$$

assuming Claim 7.1 holds.

A particular time–data trade-off, using time $2^{57.50}$ and data $2^{59.94}$, has recently been found by Wang et al. [WHJÅ12].

7.5.4 RECOVERING THE KEY

We will show how one can recover the key from the state. This is mostly of interest for stream ciphers with IV. Nonetheless, the key-recovery presented here provides some understanding of the security of the general construction.

The obvious approach is to reverse the cipher from the recovered state. By reversing both FCSRs to their respective states $n + 4 = 84$ clockings after key loading, one knows from Proposition 2.2 that these particular states, $\mathbf{B}^3$ and $\mathbf{S}^3$, were visited by the BEAN FCSRs shortly after initialization finished.

One can then revert $\mathbf{B}^3$ all the way back to $\mathbf{B}^{-81}$. The state then reached was not necessarily the state that the FCSR was put into at key loading. The state after reversion is on the cycle, while the state being sought for could be on some tail leading into it. A first approach is to use Lemma 7.1 to make the carry 0, which it was at key loading. The state of $\boldsymbol{b}$ reached in this way will be referred to as the key stub $\widehat{\boldsymbol{k}}$.

It is straightforward to test if the key stub is the correct key. When loaded with it, the two registers reach $\mathbf{B}^3$ and $\mathbf{S}^3$, respectively. In simulations, $\widehat{\boldsymbol{k}} = \boldsymbol{k}$ with large probability; in 2^{17} runs, $\widehat{\boldsymbol{k}} \neq \boldsymbol{k}$ only $34 \approx 2^5$ times. In such an unlikely event, $\widehat{\boldsymbol{k}}$ can still be used as the basis for a brute force through the register, creating key candidates. By changing more and more bits, from left to right, eventually the correct key will be found.

Algorithm 7.3. Time–data trade-off attack

```
    Input: keystream
    Output: corresponding key

 1. Recover $\mathbf{S}$ using Algorithm 7.2.
 2. Derive bits from $\mathbf{B}$, using $\mathbf{S}$ and $z$.
 3. If less than 20 bits found, {
        Fast-forward a few clocks and go to step 2.
    }
 4. Brute force the rest of $\mathbf{B}$.
 5. Revert registers to $\mathbf{B}^3$ and $\mathbf{S}^3$.
 6. Revert FCSR-I to $i = -81$.
 7. Ensure $m_b^{-81} = 0$ using Lemma 7.1.
 8. Set $\widehat{\boldsymbol{k}} = \boldsymbol{b}$.
 9. If $\widehat{\boldsymbol{k}}$ does not yield $\mathbf{B}^3$ {
        Alter $\widehat{\boldsymbol{k}}$ until $\widehat{\boldsymbol{k}}$ yields $\mathbf{B}^3$.
    }
10. Return $\widehat{\boldsymbol{k}}$.
```

While the worst case scenario is that the entire key must be brute forced, simulations have shown that in those 34 cases from above, at most 3 key candidates had to be tried. Proposition 2.2 really describes the worst case behavior [ABM08]. From these simulations, it appears as if the expected number of key candidates that need to be tried is practically 1, and there appears to be only some very small probability that this part of the attack becomes non-marginal.

Summarizing, the complete attack is outlined in Algorithm 7.3. In order to completely formalize the attack, several indices and variables are required. To not clutter the algorithm, only the broad picture is given.

Example 7.2 The following is the output from an implementation of the attack. Notably, the attack is given $|\mathcal{B} \cup \mathcal{C}| = 56$ bits of $\boldsymbol{s}$ at each time instance, so that it only needs to guess $|\mathcal{A}| = 19$ bits correctly and verify them while recovering the carry. Further, once some reasonable number of linear equations in bits of $\boldsymbol{b}$ have been derived, the attack is given the full content of $\mathbf{B}$, since the concluding brute force would take time, e.g., 2^{50}.

```
Recovered s after producing 9668 keystream bits (1.2 KiB).
Will try to find linear equations in B-bits.
b_08 = 1   b_23 = 1   b_25 = 1   b_36 = 1
b_55 = 1   b_58 = 1   b_67 = 1   b_72 = 0
b_73 = 1   b_75 = 0
b_01 + b_51 = 1       b_10 + b_60 = 1
b_24 + b_74 = 1
13 bits recovered. Fast-forward 80 keystream bits and retry.
b_01 = 1   b_05 = 1   b_18 = 1   b_24 = 1
b_26 = 1   b_29 = 0   b_51 = 1   b_55 = 0
b_60 = 1   b_62 = 0   b_65 = 0   b_67 = 0
b_68 = 1   b_73 = 1   b_74 = 1   b_76 = 1
b_77 = 0
b_03 + b_53 = 1       b_04 + b_54 = 1
b_13 + b_63 = 1       b_19 + b_69 = 0
b_25 + b_75 = 0
22 bits of B recovered.
Brute-force the remaining 83-22=61 bits...
B found. State completely recovered.
Reverting FCSRs to 84 updates after key-loading.
Reverting FCSRs to 0 updates after key-loading.
B: 1111111111111111111010001100010111110110
       1001010011011001011011100101110110100101 mem: 3
S:  1111111111111111111010001100010111110110
       1001010011011001011011100101110110100101  mem: 1
Moving memory to state to bring memory to 0.
B: 0100000000000000000110001100010111110110
1001010011011001011011100101110110100101 mem: 0
S:  0000000000000000000110001100010111110110
1001010011011001011011100101110110100101  mem: 0
Concentrating on B from now on.
Use current B as key stub.
Evaluate key stub by clocking 84 times.
Key stub was not correct key.
Need to flip more and more bits.
Found key after 2 adjustments.
Recovered key: 1000000000000000000110001100010111110110
                1001010011011001011011100101110110100101
True key:      1000000000000000000110001100010111110110
                1001010011011001011011100101110110100101
```

□

A few further observations are illustrated in Example 7.2.

- **S** is recovered using the linear approximation in Equation 7.6. When looking for linear equations in $\boldsymbol{b}$, the properties of f that are used tend to »conflict« with Equation 7.6, so that a relatively small number of equations can be derived from the same window of keystream.

- All cases encountered during the simulations where the key stub was incorrect are similar to above. The key contains a large number of zeros in the left part, which after clocking forwards and backwards turns into a sequence of ones. After forcing the memory to 0, $(\widehat{\boldsymbol{k}}) = (\boldsymbol{k}) + 1$.

- When $\boldsymbol{b} \neq \boldsymbol{k}$, 79 bits of the key can often be found in $\boldsymbol{s}$.

7.6 PROTECTING AGAINST THE ATTACKS

The first approach to strengthening BEAN should be to select a better output function f. Resiliency at least one would have excluded the very straightforward distinguisher presented in Section 7.2. Furthermore, with resiliency at least two, it would not have been possible to involve only two bits of $\boldsymbol{s}$ in each guess as in Equation 7.6.

It might also be necessary to let f depend on more than six variables. A third factor to consider might be suitability for hardware implementation. For more on this, see the paper by Wang et al. [WHJÅ12].

One notable difference between BEAN and the various incarnations of Grain is that the latter use a feed-forward from the NFSR to the LFSR. One idea might thus be to strengthen BEAN by adding a feed-forward from **S** to **B**. However, this does not protect from any of the attacks in this chapter as they first derive **S** completely before turning to **B**. Also, doing this would alter the behavior of **B** from well-known to unknown.

Any future design should include an IV, that can be transmitted in the clear and significantly simplifies key management. This approach is taken in most modern stream ciphers. Initialization could perhaps be done similarly to the Grain ciphers, with suppressed output being fed into both registers.

Also, one should probably reselect the feedback taps as there are some unfortunate properties of the current choices:

- One initial state bit of **S** is completely disregarded as $79 \notin \mathcal{T}_S$.

- The taps in **S** are clustered, as opposed to spread out somewhat evenly (cf. **B**).

- The connection integers q are not optimal. As indicated in Subsection 2.9.3, q should ideally have certain number theoretic properties.

While only the first of these features was exploited in this chapter, a serious redesign should address these potential weaknesses.

7.6.1 IMPROVING THE ATTACK

The approach taken in this chapter for deriving, e.g., $\mathbf{s}$ is not necessarily optimal. One can easily make assumptions on some bits of $\mathbf{s}$, increasing the probability of guessing correctly. By assuming some specific configuration of d bits, which should occur with probability 2^{-d}, one can make more advanced guesswork recovering the remaining bits with probability 2^{-e}, where e should be »small.« The data requirement would then be 2^{d+e}, and the time requirement might be lower than what we have presented in this chapter. One such attack is presented by Wang et al. [WHJÅ12]. Note that to find the »optimal« attack, one might need to consider several properties of f and bits that reoccur in the equations, and build decision trees that allow the implementation to adapt its behavior to the guesses already made.

7.7 CONCLUSION

It has been seen that the nonlinear function in BEAN, combined with other properties of the construction, allows for an efficient distinguisher and a key-recovery faster than brute force. It was also seen how access to more keystream allows for a faster key recovery. Already at a very modest 6 KiB, the 80-bit key is recovered in time 2^{73}. While the distinguisher requires slightly more data, and thus is information theoretically inferior to the distinguisher inherent in the key-recovery attack, it has a practical time requirement, making it interesting in its own right.

8 Related-Key Attacks on KTANTAN

KTANTAN is a hardware-oriented block cipher designed by De Cannière, Dunkelman and Knežević. It is part of the KATAN family of six block ciphers [DDK09]. There are three variants KTANTANn where $n \in \{32, 48, 64\}$. All ciphers consist of 254 very simple, hardware-efficient rounds. Each round key consists of two bits.

The only difference between KATAN and KTANTAN is the key schedule. The goal with KTANTAN is to allow an implementation to use a burnt-in key, which rules out loading the key into a register and applying some state updates to it in order to produce round keys. Instead, round keys are chosen as two bits of the key, selected according to a fixed schedule. This schedule is the same for all three variants.

Aiming for a lightweight cipher, the designers of KTANTAN did not provide the key schedule as a large table of how to select the key bits. Rather, a small state machine generates numbers between 0 and 79. In this way, key bits can hopefully be selected in an irregular fashion. As shown by Bogdanov and Rechberger [BR10], the sequence in which the key bits are used has some unwanted properties.

We will revisit the result of Bogdanov and Rechberger. We adjust the presentation slightly, before using their observation to launch a related-key attack. Bogdanov and Rechberger noted this as a possible direction of research, but did not look into it further.

Like most other related-key attacks, the ones presented in this chapter are quite academic in their nature. They can still be considered a good measurement of the security of the cipher, which should appear as an ideal permutation, and several notable properties make the attacks in this chapter very interesting:

1. They only require flipping one bit in the key and in several cases, it is enough for the attacker to use only one triplet: one plaintext and two ciphertexts.
2. They can find a large number of key bits in time equivalent to just a few encryptions. For KTANTAN32, the entire key can be found in half a minute on a current CPU.
3. They never fail. All the properties exploited in this chapter have probability one, meaning the correct (partial) key will always be found.
4. They directly contradict the designers' claims. We will discuss why this is, and what can be learned from this.

The remainder of this chapter is organized as follows. In Section 8.1 we describe the cipher KTANTAN, and Section 8.2 discusses the result by Bogdanov and Rechberger [BR10]. Section 8.3 develops our attacks on KTANTAN32, while we summarize our results on all versions of KTANTAN in Section 8.4. In Section 8.5 we compare our results to the designers' original claims on related-key security before concluding the chapter in Section 8.6.

8.1 KTANTAN

The n-bit plaintext $\boldsymbol{p}$ is loaded into the state of the cipher, which consists of two shift registers, $\boldsymbol{L^0}$ and $\boldsymbol{L^1}$, see Figure 8.1. For KTANTAN32, these are of lengths $|\boldsymbol{L^0}| = 19$ and $|\boldsymbol{L^1}| = 13$. The other variants use longer registers.

Each round uses two key bits, $k_0^r = k_{i_0^r}$ and $k_1^r = k_{i_1^r}$, $r = 0, 1, \ldots, 253$, which are selected from the 80-bit master key using indices $i_0^r, i_1^r \in \{0, 1, \ldots, 79\}$. The key schedule is provided in Table 8.1 and is the same for all versions of KTANTAN.

Table 8.1: The key schedule of KTANTAN. In each round, two key bits are added to the state.

r	i_0^r	i_1^r	r	i_0^r	i_1^r	r	i_0^r	i_1^r	r	i_0^r	i_1^r
0	63	31	1	31	63	2	31	63	3	15	47
4	14	14	5	60	76	6	40	40	7	49	17
8	35	67	9	54	22	10	45	77	11	58	26
12	37	69	13	74	10	14	69	69	15	74	10
16	53	21	17	43	43	18	71	7	19	63	79
20	30	62	21	45	45	22	11	11	23	54	70
24	28	60	25	41	41	26	3	19	27	38	70
28	60	28	29	25	73	30	34	34	31	5	21

32	26	74	33	20	52	34	9	41	35	2	18
36	20	68	37	24	56	38	1	33	39	2	2
40	52	68	41	24	56	42	17	49	43	3	35
44	6	6	45	76	76	46	72	8	47	49	17
48	19	51	49	23	55	50	15	63	51	14	46
52	12	28	53	24	72	54	16	48	55	1	49
56	2	34	57	4	20	58	40	72	59	48	16
60	17	65	61	18	50	62	5	53	63	10	58
64	4	36	65	8	8	66	64	64	67	64	0
68	65	1	69	51	19	70	23	55	71	47	47
72	15	15	73	78	78	74	76	12	75	73	9
76	67	3	77	55	23	78	47	47	79	63	31
80	47	79	81	62	30	82	29	77	83	26	58
84	5	37	85	10	26	86	36	68	87	56	24
88	33	65	89	50	18	90	21	69	91	42	42
92	5	5	93	58	74	94	20	52	95	25	57
96	3	51	97	6	38	98	12	12	99	56	72
100	16	48	101	33	33	102	3	3	103	70	70
104	60	28	105	41	41	106	67	3	107	71	71
108	78	14	109	77	13	110	59	27	111	39	39
112	79	15	113	79	79	114	62	30	115	45	45
116	59	27	117	23	71	118	46	46	119	13	29
120	42	74	121	52	20	122	41	73	123	66	2
124	53	69	125	42	42	126	53	21	127	27	75
128	38	38	129	13	13	130	74	74	131	52	20
132	25	57	133	35	35	134	7	7	135	62	78
136	44	44	137	73	9	138	51	67	139	22	54
140	29	61	141	11	43	142	6	22	143	44	76
144	72	8	145	65	65	146	50	18	147	37	37
148	75	11	149	55	71	150	46	46	151	77	13
152	75	75	153	70	6	154	61	29	155	27	59
156	39	39	157	15	31	158	46	78	159	76	12
160	57	73	161	34	34	162	69	5	163	59	75
164	38	38	165	61	29	166	43	75	167	70	6
168	77	77	169	58	26	170	21	53	171	43	43
172	7	23	173	30	78	174	44	44	175	9	25
176	18	66	177	36	36	178	9	9	179	50	66
180	36	36	181	57	25	182	19	67	183	22	54
184	13	45	185	10	10	186	68	68	187	56	24
188	17	49	189	19	51	190	7	39	191	14	30

Table 8.2: Parameter values for KTANTANn, $n \in \{32, 48, 64\}$.

n	$\lvert \boldsymbol{L^0} \rvert$	$\lvert \boldsymbol{L^1} \rvert$	x_0	x_1	x_2	x_3	y_0	y_1	y_2	y_3	y_4
32	19	13	5	4	7	9	11	6	8	10	15
48	29	19	6	3	11	12	9	7	15	13	22
64	39	25	9	4	13	15	13	5	17	24	29

192	28	76	193	40	40	194	1	1	195	66	66
196	68	4	197	57	25	198	35	35	199	55	23
200	31	79	201	30	62	202	13	61	203	10	42
204	4	4	205	72	72	206	48	16	207	33	33
208	51	19	209	39	71	210	78	14	211	61	77
212	26	58	213	21	53	214	11	59	215	6	54
216	12	44	217	8	24	218	32	64	219	64	0
220	49	65	221	18	50	222	37	37	223	11	27
224	22	70	225	28	60	226	9	57	227	2	50
228	4	52	229	8	40	230	0	0	231	48	64
232	32	32	233	65	1	234	67	67	235	54	22
236	29	61	237	27	59	238	7	55	239	14	62
240	12	60	241	8	56	242	0	32	243	0	16
244	16	64	245	32	32	246	1	17	247	34	66
248	68	4	249	73	73	250	66	2	251	69	5
252	75	11	253	71	7						

The contents of the registers are shifted, and the new bit in each register ($\boldsymbol{L^0}/\boldsymbol{L^1}$) is created from five or six bits from the other register ($\boldsymbol{L^1}/\boldsymbol{L^0}$), through some simple functions of degree two. For all versions of KTANTAN, the update is specified by

$$\begin{aligned} L^0_{|\boldsymbol{L^0}|-1} &\leftarrow f^r_0(\boldsymbol{L^1}, \boldsymbol{k}) = L^1_0 + L^1_{x_0} + (L^1_{x_1} \cdot L^1_{x_2}) + (L^1_{x_3} \cdot \mathrm{IR}_r) + k^r_0, \\ L^1_{|\boldsymbol{L^1}|-1} &\leftarrow f^r_1(\boldsymbol{L^0}, \boldsymbol{k}) = L^0_0 + L^0_{y_0} + (L^0_{y_1} \cdot L^0_{y_2}) + (L^0_{y_3} \cdot L^0_{y_4}) + k^r_1. \end{aligned}$$

The indices x_i and y_i are given by Table 8.2.

There is a round constant IR_r, 0 or 1, which decides whether a certain bit from $\boldsymbol{L^1}$ is included in the update of $\boldsymbol{L^0}$ or not. It is taken from a sequence with long period in order to rule out slide attacks [BW99] and similar approaches.

For KTANTAN32, one state update is performed per round. In KTANTAN48 and KTANTAN64, there are two resp. three updates per round using the same key bits and round constant. This larger amount of state updates means the state mixing is faster, making our attacks slightly more expensive

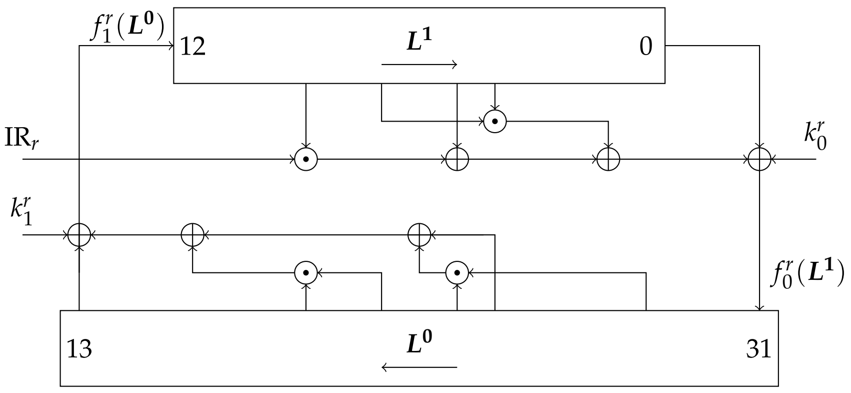


Figure 8.1: An overview of KTANTAN32. In each clocking, one shift is made and two key bits, k_0^r and k_1^r, are added to the state. IR_r is a round constant which decides whether or not L_3^1 is used in the state update or not. Indices denote how bits in the plaintext/ciphertext are identified. $\boldsymbol{L^0}$ is shifted to the left and $\boldsymbol{L^1}$ to the right.

on the larger versions of KTANTAN. We use KTANTAN32 to describe our attacks but also give the characteristics for the attacks on KTANTAN48/64.

Note how the key bits are added linearly to the state. Only after three clockings will they start to propagate nonlinearly. This gives a very slow diffusion, which we will be able to use in our attacks.

8.1.1 ON BIT ORDERING AND TEST VECTORS

Test vectors for KTANTAN can be produced by the reference code given by the designers. As an example, the all-ones key and the all-zeros plaintext produce the ciphertext `22ea3988`. Unfortunately, this does not highlight the bit order in the plaintext and, more importantly in this chapter, the key. By editing the reference code slightly, one can find that the key `7fffffffffffffffffff` and the plaintext `00000001` produce the ciphertext `8b4f0824`. The work described in this chapter has been performed on an implementation of KTANTAN which matches this test vector.

8.2 A PREVIOUS RESULT ON KTANTAN

Bogdanov and Rechberger [BR10] note that some key bits are not used until very late during encryption, while some others are never used after some relatively small number of rounds, see Table 8.3. Given a plaintext–ciphertext

Table 8.3: The nine most extreme key bits in both directions during encryption. Six bits do not appear before round 111, while six others are not used after round 131.

Key bit	Used first in round	Key bit	Used last in round
k_{13}	109	k_{38}	164
k_{27}	110	k_{46}	158
k_{59}	110	k_{15}	157
k_{39}	111	k_{20}	131
k_{66}	123	k_{74}	130
k_{75}	127	k_{41}	122
k_{44}	136	k_{3}	106
k_{61}	140	k_{47}	80
k_{32}	218	k_{63}	79

pair, this results in a guess-and-determine attack, where the »determine« part is a meet-in-the-middle. Guess 68 key bits. Of the twelve remaining key bits, six are not used in the first part of the cipher, which means there are only $2^{12-6} = 2^6$ different states after calculating $F_{0,111}$ from the plaintext. Similarly, there are 2^6 possible states after calculating $F^{-1}_{132,254}$ from the ciphertext. By checking the 2^{12} combinations for matches, one can find the key. In KTANTAN32, one can use eight bits in the midcipher state to judge equality, so false positives should appear with rate 2^{-8}. Some additional plaintext–ciphertext pairs will help rule out the false positives, but they are needed anyway since $|\boldsymbol{k}| > |\boldsymbol{c}|$ (cf. Subsection 4.1.2).

Bogdanov and Rechberger dub this a 3-subset meet-in-the-middle attack. By defining

$$\mathcal{A}_f = \{3, 20, 41, 47, 63, 74\},$$
$$\mathcal{A}_b = \{32, 39, 44, 61, 66, 75\},$$

their attack can be formulated as in Algorithm 8.1. They also give similar attacks for KTANTAN48 and KTANTAN64.

8.2.1 REFORMULATING THE ATTACK

Step 4 is not trivial as the computations that need to be carried out in order to check for matches are similar to calculating the round functions themselves. Further, while the original authors choose to only use eight bits for matching, we have found that one can use twelve bits, given by the mask `2a03cd44`. This slightly lowers the time required for the attack as one can expect fewer false positives. Summing up, we prefer to view the attack as in Algorithm 8.2.

Algorithm 8.1. 3-subset meet-in-the-middle attack on KTANTAN, take I

Input: a few KTANTAN32 plaintext–ciphertext pairs
Output: the secret key $\boldsymbol{k}$

```
1. Guess the 68 key bits indexed by $\{0,1,\dots,79\}\backslash(\mathcal{A}_f \cup \mathcal{A}_b)$.
2. Compute $2^6$ partial encryptions $s^0_0,\dots,s^0_{63}$ using $F_{0,111}$ for each choice of
   bit assignments for bits indexed by $\mathcal{A}_f$.
3. Compute $2^6$ partial decryptions $s^1_0,\dots,s^1_{63}$ using $F^{-1}_{132,254}$ for each choice of
   bit assignments for bits indexed by $\mathcal{A}_b$.
4. For the $2^{12}$ combinations, use eight state bits to check whether they match.
   Alarms will be raised with probability $2^{-8}$, so we expect $2^4$ alarms.
5. Use some additional plaintext–ciphertext pairs to rule out false alarms, by
   simple trial encryptions.
6. Return the key found or go to step 1.
```

Algorithm 8.2. 3-subset meet-in-the-middle attack on KTANTAN, take II

Input: a few KTANTAN32 plaintext–ciphertext pairs
Output: the secret key $\boldsymbol{k}$

```
1. Guess the 68 key bits indexed by $\{0,1,\dots,79\}\backslash(\mathcal{A}_f \cup \mathcal{A}_b)$.
2. Compute $2^6$ partial encryptions $s^0_0,\dots,s^0_{63}$ using $F_{0,127}$ for each choice of
   bit assignments for bits indexed by $\mathcal{A}_f$.
3. Compute $2^6$ partial decryptions $s^1_0,\dots,s^1_{63}$ using $F^{-1}_{127,254}$ for each choice of
   bit assignments for bits indexed by $\mathcal{A}_b$.
4. For the $2^{12}$ combinations, check twelve state bits for equality through
           if ( (($s^0_i$^$s^1_j$)&0x2a03cd44) == 0 ) { ... }.
   Alarms will be raised with probability $2^{-12}$, so we expect one alarm.
5. Use some additional plaintext–ciphertext pairs to rule out false alarms, by
   simple trial encryptions.
6. Return the key found or go to step 1.
```

An implementation improvement is to only calculate those 12 bits that we actually need. We have then reached something similar to the original formulation of the attack, with the notable difference that the computations corresponding to $F_{111,127}$ and $F^{-1}_{127,132}$ are not performed an unnecessarily large number of times.

We can split at any round between and including 123 and 127, and still get twelve known (but different) bit positions to look at. We opted for 127 as it makes both halves equally expensive to calculate.

8.3 RELATED-KEY ATTACKS ON KTANTAN32

We first outline some preliminaries, including how attack time and data requirements will be derived. We then further study how k_{32} enters the key schedule very late. After this, we formulate our attack idea and derive various attacks that find some parts of the key.

8.3.1 TRUNCATED DIFFERENTIALS AND ATTACK REQUIREMENTS

We only use differentials with probability one, which means there are only false positives and no false negatives. The false positives can be ruled out by repeated filtering. As a result, all attacks given in this chapter have probability one of succeeding. When we give data requirements, these will be the expected number of samples needed to obtain a unique solution. Similarly, time requirements will account for the work needed to rule out false alarms. We assume that an alarm is raised with probability 2^{-b} for a differential that involves b bits.

As above, we will always need some extra material in order to find a unique key.

8.3.2 ON THE BAD MIXING OF k_{32}

Key bit 32 is particularly weak as it appears for the first time in round 218 of 254. We have thus studied this bit closer. It is worth noting that if the cipher had used 253 rounds rather than 254, one ciphertext bit would have been linear in k_{32}. That is, there is a probability-one, 253-round differential $(\mathbf{0}, k_{32}) \to [\texttt{00040000 : 00040000}]$. The single bit involved is state bit 13 in Figure 8.1, i.e., the leftmost bit in $\boldsymbol{L^0}$. This bit is shifted out of the state in the very last round, so such a probability-one differential is not available on the full KTANTAN. However, there are some high-probability truncated differentials on the full KTANTAN as given in Table 8.4. We do not exploit these differentials in this chapter, but note that they indicate a nonrandom behavior of the cipher.

Table 8.4: Probabilistic truncated differentials on the full KTANTAN32.

Differential	Probability
$(\mathbf{0}, k_{32}) \to [00020000 : 00020000]$	$.687 = .5 + .187$
$(\mathbf{0}, k_{32}) \to [40000000 : 00000000]$	$.640 = .5 + .140$
$(\mathbf{0}, k_{32}) \to [40020000 : 00020000]$	$.453 = .25 + .203$

8.3.3 THE GENERAL ATTACK IDEA

We will present several related-key attacks that recover some or all key bits. The general outline of our attacks can be formulated as follows: We group key bit indices into disjoint subsets $\mathcal{A}_0, \dots, \mathcal{A}_{l-1}$ of sizes $a_i = |\mathcal{A}_i|$, $i = 0, \dots, l-1$. These subsets do not necessarily need to collectively contain all 80 key bits. Define $a = \sum_i a_i$.

We attack these subsets one after another, i.e., when attempting to find the correct bit assignments for $\mathcal{A}_j$, we assume that we already know the correct bit assignments for bits indexed by $\mathcal{A}_i$, $i = 0, \dots, j-1$. We then follow this simple outline:

1. Guess the bit assignments for bits indexed by $\mathcal{A}_j$.

2. If the (truncated) differential matches, we have a candidate subkey.

3. If the (truncated) differential does not match, we discard the candidate subkey.

In the first step, we can make 2^{a_j} guesses for the subkey. Note that the last step can be performed without risk, since all our differentials have probability one. Due to this, we can immediately discard many of the guesses.

The second step of the attack can, however, give false positives. As already noted, we assume that a false alarm is raised with probability 2^{-b} for a differential that involves b bits. To discard the false alarms, we can recheck the differential on more material, so we expect to need a_i/b triplets.

After finding the key bits specified by $\cup_i \mathcal{A}_i$, we can conclude by a brute force for the remaining $80 - a$ key bits. A simple calculation of the time required would yield $2^{a_0} + \ldots + 2^{a_{l-1}} + 2^{80-a}$. However, the different operations represented by these terms have different costs as they require different numbers of (inverse) round function evaluations. All time requirements in this chapter will be normalized to KTANTAN calls, and also incorporate the expected increase of calculations due to false positives. We will denote this time measurement T and it will, depending on context, refer to the time required to recover either the full key or only some part of it.

8.3.4 A FIRST APPROACH: FINDING 28 BITS OF THE KEY

Assume that we have a known plaintext $\boldsymbol{p}$, and two ciphertexts $\boldsymbol{c}^0, \boldsymbol{c}^1$, where the difference is that k_{32} has been flipped in the unknown key between the calculations of the two ciphertexts. During the calculations of these two ciphertexts, the first 218 states followed the same development. Only after k_{32} entered could the calculations diverge to produce different ciphertexts.

Bogdanov and Rechberger give the probability-1 differential $(0, k_{32}) \rightarrow 0$ for 218 rounds. We note that this differential can be easily extended to 222 rounds, still with probability 1: $(0, k_{32}) \rightarrow 00000008$. The flipped bit in $\Delta \boldsymbol{s}$ is the linear appearance of k_{32}.

We will use »triplets« consisting of one plaintext and *two* ciphertexts to exploit these differentials. A first attempt to use such a plaintext–ciphertexts triplet in an attack could be as follows. We note that there are 42 key bits used when decrypting into round 222, indexed by

$$\{0,1,2,4,5,7,8,9,11,12,14,16,17,22,27,28,29,32,34,37,40,48,\\ 50,52,54,55,56,57,59,60,61,62,64,65,66,67,68,69,70,71,73,75\}.$$

We guess these bits and put the guess in $\boldsymbol{k}$ (nonguessed bits can be given any value). Denote by $\boldsymbol{k}'$ the key which differs from $\boldsymbol{k}$ only by $k'_{32} \neq k_{32}$. Calculate $\boldsymbol{s}^0 = F^{-1}_{222,254}(\boldsymbol{c}^0, \boldsymbol{k})$ and $\boldsymbol{s}^1 = F^{-1}_{222,254}(\boldsymbol{c}^1, \boldsymbol{k}')$. For a correct guess, both ciphertexts will decrypt into the same state $\boldsymbol{s}^0 = \boldsymbol{s}^1$.

However, several of the key bits, e.g., k_{37}, only appear linearly, and if the other 41 key bits are correct, $\boldsymbol{s}^0 = \boldsymbol{s}^1$ no matter how k_{37} is guessed. It turns out that only 28 bits can be recovered, indexed by

$$\mathcal{A}_0 = \{0,1,2,4,5,7,8,11,12,14,16,17,22,27,29,\\ 32,34,55,56,60,62,64,66,68,69,71,73,75\}.$$

As a consequence, only 28 bits need to be guessed. For each guess, two partial decryptions of 32 out of 254 rounds are performed. The total number of round function calls is expected to be $2^{28} \cdot 2 \cdot 32 = 2^{34}$, which corresponds to $2^{34}/254 \approx 2^{26.01}$ full KTANTAN evaluations. Thus the total time required for finding 28 bits is $T \approx 2^{26}$. All time requirements in the remainder of the chapter will be calculated in this way.

By using brute-force for the remaining key bits, the entire key can be found in time $T \approx 2^{26} + 2^{62} \approx 2^{62}$.

8.3.5 MAKING IT FASTER

Rather than guessing 28 bits at once, we note that we can apply a divide-and-conquer approach to these bits, determining a few bits at a time. This will significantly improve the time required for the attack. Due to the slow diffusion, we cannot find any truncated differential on 247 rounds or more for our purposes, but for 246 rounds, there is $(\mathbf{0}, k_{32}) \to [\texttt{80050800 : 00000800}]$. This differential can be used to find three bits indexed by $\mathcal{A}_0 = \{11, 66, 71\}$, in time $T \approx 2^{-0.9}$. (That this work is performed in time less than one unit results from the fact that we only perform a small number of round calculations, compared to the full 254 rounds that are calculated in one time unit.)

We now know the three bits indexed by $\mathcal{A}_0$ and attempt to find more bits. There is no useful 245-round differential, but the 244-round truncated differential $(\mathbf{0}, k_{32}) \to [\texttt{20054200 : 00000200}]$ can be used to obtain one additional key bit, indexed by $\mathcal{A}_1 = \{2\}$, with $T \approx 2^{-0.5}$.

Continuing with such small chunks, we can find the 28 bits with

$$T \approx 2^{-0.9} + 2^{-0.5} + \ldots \approx 2^{3.0}.$$

All differentials involved are listed in Table 8.5. PCC means that the differential is of type $(\Delta \boldsymbol{p}, \Delta \boldsymbol{k}) \to \Delta \boldsymbol{s}$, where $\boldsymbol{s}$ is the state some rounds into the encryption. Similarly, CPP means a differential $(\Delta \boldsymbol{c}, \Delta \boldsymbol{k}) \to \Delta \boldsymbol{s}$, extending some rounds into the decryption. (The 'Rnds' column then denote the round into which we decrypt, not the number of decryption rounds.) The '#Bits' column counts how many key bits that need to be guessed. We also give the reduced number of guessed key bits in $\mathcal{A}_j$ when we have already acquired a part of the key, $\cup_{i<j} \mathcal{A}_i$, by using the differentials found earlier in the table.

8.3.6 USING TRIPLETS WITH ONE CIPHERTEXT AND TWO PLAINTEXTS

The key bit k_{32} appeared very late in the encryption, and we exploited this above. Similarly, k_{63} is only used in the first 80 rounds, meaning that during *de*cryption it shows similar properties. With triplets consisting of one ciphertext and two plaintexts, corresponding to a secret key with a flipped k_{63}, we can launch an attack similar to that above, with a truncated differential involving a single bit. With

$$\begin{aligned}\mathcal{A}_0 = \{&7, 10, 11, 14, 15, 17, 19, 21, 22, 25, 26, 28, 30, 31, 34, 35, 37, 38, 40, 41,\\ &43, 45, 47, 49, 52, 53, 54, 58, 60, 62, 63, 67, 68, 69, 70, 71, 74, 76, 77, 79\},\end{aligned}$$

using $F_{0,43}$, we guess and obtain 40 bits, using 40 triplets and $T \approx 2^{39.44}$. We can then exploit k_{63} for more subsets $\mathcal{A}_1, \ldots, \mathcal{A}_{15}$ and partial encryptions $F_{0,45}, \ldots, F_{0,71}$, finding in total 65 bits of the key still with $T \approx 2^{39.44}$. Concluding with a brute force for the remaining bits, we can find the entire key in $T \approx 2^{39.44} + 2^{15} \approx 2^{39.44}$. All subsets, truncated differentials, etc. can be found in Table 8.6.

Table 8.5: The differentials used on KTANTAN32 in Subsections 8.3.5 and 8.3.7.

Type	Rnds	#Bits	$\mathcal{A}_j$	Differential
PCC	246	3	$\{11,66,71\}$	$(\mathbf{0},k_{32}) \rightarrow [\texttt{80050800} : \texttt{00000800}]$
PCC	244	4/1	$\{2\}$	$(\mathbf{0},k_{32}) \rightarrow [\texttt{20054200} : \texttt{00000200}]$
PCC	243	7/3	$\{5,7,73\}$	$(\mathbf{0},k_{32}) \rightarrow [\texttt{1006a100} : \texttt{00000100}]$
PCC	242	8/1	$\{4\}$	$(\mathbf{0},k_{32}) \rightarrow [\texttt{08075080} : \texttt{00000080}]$
PCC	241	11/3	$\{32,68,75\}$	$(\mathbf{0},k_{32}) \rightarrow [\texttt{8407a840} : \texttt{00000040}]$
PCC	239	14/3	$\{1,34,69\}$	$(\mathbf{0},k_{32}) \rightarrow [\texttt{a107ea10} : \texttt{80000010}]$
PCC	238	15/1	$\{0\}$	$(\mathbf{0},k_{32}) \rightarrow [\texttt{d087f508} : \texttt{40000008}]$
PCC	237	17/2	$\{8,16\}$	$(\mathbf{0},k_{32}) \rightarrow [\texttt{e847fa84} : \texttt{20040004}]$
PCC	236	19/2	$\{12,17\}$	$(\mathbf{0},k_{32}) \rightarrow [\texttt{f427fd42} : \texttt{10020002}]$
PCC	234	20/1	$\{64\}$	$(\mathbf{0},k_{32}) \rightarrow [\texttt{bd0fff50} : \texttt{04008000}]$
PCC	233	21/1	$\{27\}$	$(\mathbf{0},k_{32}) \rightarrow [\texttt{de87ffa8} : \texttt{02004000}]$
PCC	232	22/1	$\{29\}$	$(\mathbf{0},k_{32}) \rightarrow [\texttt{ef47ffd4} : \texttt{01002000}]$
PCC	231	24/2	$\{14,62\}$	$(\mathbf{0},k_{32}) \rightarrow [\texttt{f7a7ffea} : \texttt{00801000}]$
PCC	230	25/1	$\{60\}$	$(\mathbf{0},k_{32}) \rightarrow [\texttt{fbd7fff5} : \texttt{00400800}]$
PCC	229	27/2	$\{22,56\}$	$(\mathbf{0},k_{32}) \rightarrow [\texttt{fdeffffa} : \texttt{00200400}]$
PCC	222	28/1	$\{55\}$	$(\mathbf{0},k_{32}) \rightarrow [\texttt{ffffffff} : \texttt{00000008}]$
CPP	43	40/29	$\mathcal{A}_{16}$ (see below)	$(\mathbf{0},k_{63}) \rightarrow [\texttt{00000001} : \texttt{00000001}]$
CPP	45	45/4	$\{3,9,18,33\}$	$(\mathbf{0},k_{63}) \rightarrow [\texttt{00000005} : \texttt{00000004}]$
CPP	46	49/2	$\{20,24\}$	$(\mathbf{0},k_{63}) \rightarrow [\texttt{0000000b} : \texttt{00000008}]$
CPP	51	52/1	$\{6\}$	$(\mathbf{0},k_{63}) \rightarrow [\texttt{0000017f} : \texttt{00000108}]$
CPP	55	54/1	$\{51\}$	$(\mathbf{0},k_{63}) \rightarrow [\texttt{000017ff} : \texttt{00001080}]$
CPP	57	57/1	$\{72\}$	$(\mathbf{0},k_{63}) \rightarrow [\texttt{00085fff} : \texttt{00084200}]$
CPP	58	58/1	$\{46\}$	$(\mathbf{0},k_{63}) \rightarrow [\texttt{0010bfff} : \texttt{00108400}]$
CPP	60	59/1	$\{23\}$	$(\mathbf{0},k_{63}) \rightarrow [\texttt{0042ffff} : \texttt{00421000}]$
CPP	61	60/1	$\{48\}$	$(\mathbf{0},k_{63}) \rightarrow [\texttt{008dffff} : \texttt{00842000}]$
CPP	67	62/1	$\{65\}$	$(\mathbf{0},k_{63}) \rightarrow [\texttt{237fffff} : \texttt{21080000}]$
CPP	68	64/1	$\{50\}$	$(\mathbf{0},k_{63}) \rightarrow [\texttt{46ffffff} : \texttt{42100000}]$
CPP	71	65/1	$\{36\}$	$(\mathbf{0},k_{63}) \rightarrow [\texttt{37ffffff} : \texttt{10800000}]$
CPP	83	68/1	$\{78\}$	$(\mathbf{0},k_{3}) \rightarrow [\texttt{00000155} : \texttt{00000040}]$
CPP	98	70/1	$\{42\}$	$(\mathbf{0},k_{41}) \rightarrow [\texttt{000017ff} : \texttt{00001080}]$
CPP	102	71/1	$\{57\}$	$(\mathbf{0},k_{41}) \rightarrow [\texttt{00217fff} : \texttt{00210800}]$
CPP	115	72/1	$\{59\}$	$(\mathbf{0},k_{74}) \rightarrow [\texttt{046955ff} : \texttt{04214008}]$
CPP	116	73/1	$\{13\}$	$(\mathbf{0},k_{74}) \rightarrow [\texttt{08daabff} : \texttt{08428010}]$
CPP	118	75/1	$\{39\}$	$(\mathbf{0},k_{74}) \rightarrow [\texttt{237aafff} : \texttt{210a0040}]$
PCC	172	70/2	$\{44,61\}$	$(\mathbf{0},k_{61}) \rightarrow [\texttt{00050000} : \texttt{00040000}]$
$\mathcal{A}_{16} = \{10,15,19,21,25,26,28,30,31,35,37,38,40,41,43,45,47,49,52,53,54,58,63,67,70,74,76,77,79\}$				

8.3.7 GOING IN BOTH DIRECTIONS: A PRACTICAL-TIME KEY-RECOVERY

We will now describe a full key-recovery attack with very low time requirement. First, we go backwards in time $T \approx 2^{3.0}$ to find 28 bits as outlined above. Then, we go forwards using k_{63}. Here, it should be noted that of the 40 bits we needed to guess above, we have learned 11 while using k_{32}, so we only need to guess 29 bits, indexed by

$$\begin{aligned}\mathcal{A}_{16} = \{&10,15,19,21,25,26,28,30,31,35,37,38,40,41,\\ &43,45,47,49,52,53,54,58,63,67,70,74,76,77,79\}.\end{aligned}$$

We have $T \approx 2^{28.44}$. Finally, we brute force the remaining $80 - 28 - 29 = 23$ bits. The total time required for finding the entire 80-bit key is given by $T \approx 2^{3.0} + 2^{28.44} + 2^{23} \approx 2^{28.47}$.

A similar attack has been implemented, and requires less than five minutes to recover the complete key using a single thread on a 2 Xeon E5520 (2.26 Ghz, quadcore) system. Utilizing the possibility of running eight threads in parallel, the attack runs in 35 seconds. The implementation uses the more naive approaches for finding the first 28 bits, as this is easier to implement and leads to a total time of about $T \approx 2^{28.71}$, which represents a negligible change from the attack described in this section.

We can use k_{63} for finding more key bits, and also exploit several different key bits. Doing so, there is no need for a concluding brute force, and the entire key is recovered in $T \approx 2^{28.44}$. The truncated differentials involved can be found in Table 8.5.

In Table 8.5, note especially the differential on a single state bit involving 29 unknown key bits. This gives a large data requirement in order to rule out false positives, and the time required for this step dominates all other parts of the full key recovery attack. Any time improvements we make in other partial key recoveries will only be minor compared to this dominating term.

This leads to the interesting observation that if k_{32} had been *stronger*, i.e., appeared *earlier* in the key schedule, we might have been able to find more key bits at a higher cost ($> 2^3$) using it. This would then have lowered the data and time requirements for utilizing k_{63} which would have made the entire cipher *less* secure. Of course, had both key bits been stronger, the attack would instead have become more expensive.

8.3.8 MINIMIZING THE DATA REQUIREMENTS

We have tried to minimize the data requirements by using differentials involving a larger number of bits than above. For the forward direction, we can use the 62-round differential $(\mathbf{0}, k_{63}) \to [\texttt{011bffff : 01084000}]$. It requires guessing 41 bits and the false-alarm probability is 2^{-21}. The total time required for

Table 8.6: The differentials used on KTANTAN32 in Subsection 8.3.6.

Type	Rnds	#Bits	$\mathcal{A}_j$	Differential
CPP	43	40	$\mathcal{A}_0$ (see below)	$(\mathbf{0}, k_{63}) \to$ [`00000001` : `00000001`]
CPP	45	45/5	$\{3,5,9,18,33\}$	$(\mathbf{0}, k_{63}) \to$ [`00000005` : `00000004`]
CPP	46	49/4	$\{2,20,24,73\}$	$(\mathbf{0}, k_{63}) \to$ [`0000000b` : `00000008`]
CPP	47	51/2	$\{1,56\}$	$(\mathbf{0}, k_{63}) \to$ [`00000017` : `00000010`]
CPP	51	52/1	$\{6\}$	$(\mathbf{0}, k_{63}) \to$ [`0000017f` : `00000108`]
CPP	53	53/1	$\{8\}$	$(\mathbf{0}, k_{63}) \to$ [`000005ff` : `00000420`]
CPP	55	54/1	$\{51\}$	$(\mathbf{0}, k_{63}) \to$ [`000017ff` : `00001080`]
CPP	56	55/1	$\{55\}$	$(\mathbf{0}, k_{63}) \to$ [`00002fff` : `00002100`]
CPP	57	57/2	$\{12,72\}$	$(\mathbf{0}, k_{63}) \to$ [`00085fff` : `00084200`]
CPP	58	58/1	$\{46\}$	$(\mathbf{0}, k_{63}) \to$ [`0010bfff` : `00108400`]
CPP	60	59/1	$\{23\}$	$(\mathbf{0}, k_{63}) \to$ [`0042ffff` : `00421000`]
CPP	61	60/1	$\{48\}$	$(\mathbf{0}, k_{63}) \to$ [`008dffff` : `00842000`]
CPP	65	61/1	$\{16\}$	$(\mathbf{0}, k_{63}) \to$ [`08dfffff` : `08420000`]
CPP	67	62/1	$\{65\}$	$(\mathbf{0}, k_{63}) \to$ [`237fffff` : `21080000`]
CPP	68	64/2	$\{4,50\}$	$(\mathbf{0}, k_{63}) \to$ [`46ffffff` : `42100000`]
CPP	71	65/1	$\{36\}$	$(\mathbf{0}, k_{63}) \to$ [`37ffffff` : `10800000`]
$\mathcal{A}_0 = \{7,10,11,14,15,17,19,21,22,25,26,28,30,31,$ $34,35,37,38,40,41,43,45,47,49,52,53,54,$ $58,60,62,63,67,68,69,70,71,74,76,77,79\}$				

obtaining the full key then becomes $T \approx 2^{39.97}$. The data requirement is one and two triplets, respectively, in the backward and forward directions.

8.3.9 POSSIBLE IMPROVEMENTS

We have used a greedy approach for finding the differentials used in the attacks above. As an example, on $F_{0,248}$, there is the truncated differential $(\mathbf{0}, k_{32}) \to$ [`00021000` : `00001000`], but due to the slow diffusion we cannot find any key bits using it with probability one. This forces us to use the differential $(\mathbf{0}, k_{32}) \to$ [`80050800` : `00000800`] on $F_{0,247}$, where three key bits affect the differential so all three bits need to be guessed. We could truncate this truncated differential further to only involve a single bit, possibly allowing us to only guess a single key bit. In this way, we could perhaps partition the 28 bits that can be recovered using k_{32} into 28 subsets $\mathcal{A}_0, \ldots, \mathcal{A}_{27}$, and reach a very small attack time. We have not investigated this optimization as the time requirements are already impressive enough.

Note that for the key recovery attack on KTANTAN32 the attack time is dominated by exploiting k_{63} to find the 29-bit subkey defined by $\mathcal{A}_{16}$ (see Table 8.5). For this, we already use a one-bit truncated differential so this

Table 8.7: Characteristics for some attacks on KTANTAN32.

		80 bits	80 bits	28 bits	3 bits
Low time	Time	$2^{28.44}$	$2^{28.47}$	$2^{3.02}$	$2^{-0.90}$
	Data	$1/29, 1, 1, 1/1$	$1/29$	1	1
Low data	Time	$2^{39.44}$	$2^{39.97}$	as above	as above
	Data	$-/1$	$1/2$	as above	as above

cannot be improved by the technique outlined above.

8.4 SUMMARIZED ATTACKS ON ALL VERSIONS OF KTANTAN

We summarize our results on KTANTAN32 in Table 8.7. Requirements have been optimized in both dimensions: using a small amount of related-key data, and using low time requirements. Differentials and other details are found in Tables 8.5 and 8.6.

Similar attacks can be realized on KTANTAN48 and KTANTAN64. The corresponding requirements are found in Tables 8.8 and 8.9, respectively, and the differentials in Tables 8.10 and 8.11, respectively.

We give full key-recovery attacks, but also some partial-key recoveries with extremely low time requirements, similar to the $2^{3.0}$ attack on KTANTAN32 for 28 bits. We also give costs on finding the smallest possible set of key bits.

Generally, the first step is performed in the backwards direction, exploiting k_{32}. Following this, we switch to the forward direction and k_{63}. For more advanced attacks, we can use more key bits in the forward direction: k_3, k_{41}, k_{74}. We may then end using more backward calculations on k_{61}. Attacks that require less data are completed through a brute force. Slashes indicate shift of direction, commas separate needed triplets for different flipped key bits.

8.5 COMPARISON TO SPECIFICATION CLAIMS

In the specification of KTANTAN, the designers state the design goal that »no related-key key-recovery or slide attack with time [requirement] smaller than 2^{80} exists on the entire cipher« [DDK09]. They also claim to have searched for related-key differentials on KTANTAN. However, it appears the approach has been randomized over the huge space of differences in plaintext and key,

$$(\Delta \boldsymbol{p}, \Delta \boldsymbol{k}) \in \mathbb{F}_2^n \times \mathbb{F}_2^{80}.$$

With hindsight, the designers should have made sure to try differentials where we flip only some small number of plaintext or key bits. This strategy would

Table 8.8: Characteristics for some attacks on KTANTAN48.

		80 bits	36 bits	3 bits
Low time	Time	$2^{31.77}$	$2^{4.73}$	$2^{0.01}$
	Data	3/32	3	3
Low data	Time	$2^{37.34}$	$2^{31.66}$	as above
	Data	1/1	1	as above

Table 8.9: Characteristics for some attacks on KTANTAN64.

		80 bits	38 bits	13 bits
Low time	Time	$2^{32.28}$	$2^{10.75}$	$2^{10.71}$
	Data	13/17	13	13
Low data	Time	$2^{36.54}$	$2^{30.53}$	as above
	Data	1/1	1	as above

have been a good choice due to the bitwise nature of the key schedule coupled with the slow diffusion of the state. If all key bits had been investigated individually, using round-reduced versions, it would have become apparent, e.g., that k_{32} could not affect encryptions before round 218, that one state bit in KTANTAN32 only contained this key bit linearly until the very last round, and that there are some highly biased truncated differentials on the full KTANTAN32.

8.6 CONCLUSION

We have presented several weaknesses related to the key schedule of KTANTAN. We first noted how the exceptionally weak key bit k_{32} allowed for a nonrandomness result on KTANTAN32.

As the main result, we then derived several related-key attacks allowing for (partial) key recovery on all versions of KTANTAN. Our implementation of one of the attacks verifies the general attack idea and the specific results.

Finally, note that none of these attacks are directly applicable to KATAN. The slow diffusion, which allowed for, e.g., the $2^{3.0}$-attack on 28 bits, is present also in KATAN, but one needs a weak key bit in order to exploit this.

For the design of future primitives with a bitwise key schedule as in KTANTAN, we encourage designers to carefully study how individual key bits are used, either by specifically ensuring that they are used both early and late in the key schedule, or by investigating all differentials of modest weight.

Table 8.10: The differentials used on KTANTAN48.

Type	Rnds	#Bits	$\mathcal{A}_j$	Differential
PCC	246	3/3	$\{7, 11, 73\}$	$(\mathbf{0}, k_{32}) \rightarrow [\texttt{000000010000} : \texttt{000000000000}]$
PCC	242	7/4	$\{2, 4, 32, 71\}$	$(\mathbf{0}, k_{32}) \rightarrow [\texttt{000000010100} : \texttt{000000000000}]$
PCC	241	11/4	$\{5, 64, 66, 75\}$	$(\mathbf{0}, k_{32}) \rightarrow [\texttt{00000001c040} : \texttt{000000000000}]$
PCC	240	18/7	$\mathcal{A}_3$ (see below)	$(\mathbf{0}, k_{32}) \rightarrow [\texttt{000c00007010} : \texttt{000000000000}]$
PCC	239	19/1	$\{17\}$	$(\mathbf{0}, k_{32}) \rightarrow [\texttt{700011c04000} : \texttt{000000000000}]$
PCC	238	20/1	$\{56\}$	$(\mathbf{0}, k_{32}) \rightarrow [\texttt{1c001c701000} : \texttt{000000000000}]$
PCC	237	23/3	$\{12, 14, 60\}$	$(\mathbf{0}, k_{32}) \rightarrow [\texttt{0c7001f1c400} : \texttt{000400000000}]$
PCC	236	24/1	$\{62\}$	$(\mathbf{0}, k_{32}) \rightarrow [\texttt{071c01fc7100} : \texttt{000100000000}]$
PCC	235	25/1	$\{55\}$	$(\mathbf{0}, k_{32}) \rightarrow [\texttt{1c701ff1c040} : \texttt{000040000000}]$
PCC	234	26/1	$\{27\}$	$(\mathbf{0}, k_{32}) \rightarrow [\texttt{871c1ffc7010} : \texttt{000010000000}]$
PCC	233	30/4	$\{29, 54, 61, 67\}$	$(\mathbf{0}, k_{32}) \rightarrow [\texttt{e1c71fff1c04} : \texttt{000004010000}]$
PCC	232	32/2	$\{22, 65\}$	$(\mathbf{0}, k_{32}) \rightarrow [\texttt{f871dfffc701} : \texttt{00000100c000}]$
PCC	230	33/1	$\{48\}$	$(\mathbf{0}, k_{32}) \rightarrow [\texttt{cf871ffffc70} : \texttt{000000100c00}]$
PCC	229	34/1	$\{59\}$	$(\mathbf{0}, k_{32}) \rightarrow [\texttt{f3e1dfffff1c} : \texttt{000000040300}]$
PCC	225	36/2	$\{40, 52\}$	$(\mathbf{0}, k_{32}) \rightarrow [\texttt{fff3ffffffff} : \texttt{000000003000}]$
CPP	54	53/32	$\mathcal{A}_{15}$ (see below)	$(\mathbf{0}, k_{63}) \rightarrow [\texttt{000000000002} : \texttt{000000000000}]$
CPP	55	54/1	$\{6\}$	$(\mathbf{0}, k_{63}) \rightarrow [\texttt{000000000009} : \texttt{000000000000}]$
CPP	57	57/3	$\{23, 46, 51\}$	$(\mathbf{0}, k_{63}) \rightarrow [\texttt{00000000009f} : \texttt{00000000000c}]$

$\mathcal{A}_3 = \{0, 1, 8, 16, 34, 68, 69\}$,
$\mathcal{A}_{15} = \{3, 9, 10, 15, 18, 19, 20, 21, 24, 25, 26, 28, 30, 31, 33, 35, 37, 38, 41, 43, 45, 47, 49, 53, 58, 63, 70, 72, 74, 76, 77, 79\}$

Table 8.11: The differentials used on KTANTAN64.

Type	Rnds	#Bits	$\mathcal{A}_j$	Differential
PCC	241	13/13	$\mathcal{A}_0$ (see below)	$(\mathbf{0}, k_{32}) \to [\mathtt{0000000000000400} : \mathtt{0000000000000000}]$
PCC	237	21/8	$\mathcal{A}_1$ (see below)	$(\mathbf{0}, k_{32}) \to [\mathtt{0000000704000000} : \mathtt{0000000000000000}]$
PCC	236	27/6	$\mathcal{A}_2$ (see below)	$(\mathbf{0}, k_{32}) \to [\mathtt{00c000007e800000} : \mathtt{0000000000000000}]$
PCC	235	29/2	$\{29, 61\}$	$(\mathbf{0}, k_{32}) \to [\mathtt{f800007fc0100000} : \mathtt{0000000e00000000}]$
PCC	234	30/1	$\{22\}$	$(\mathbf{0}, k_{32}) \to [\mathtt{3f00007ff8020000} : \mathtt{00000001c0000000}]$
PCC	233	32/2	$\{54, 67\}$	$(\mathbf{0}, k_{32}) \to [\mathtt{c7e0007fff004000} : \mathtt{0000000038000000}]$
PCC	232	33/1	$\{65\}$	$(\mathbf{0}, k_{32}) \to [\mathtt{78fc007fffe00800} : \mathtt{0000000007000000}]$
PCC	228	34/1	$\{48\}$	$(\mathbf{0}, k_{32}) \to [\mathtt{f8c78ffffffffe00} : \mathtt{0000070038000000}]$
PCC	226	36/2	$\{40, 50\}$	$(\mathbf{0}, k_{32}) \to [\mathtt{ffe31e7ffffffff8} : \mathtt{0000000000e00000}]$
PCC	225	38/2	$\{9, 52\}$	$(\mathbf{0}, k_{32}) \to [\mathtt{fffc63ffffffffff} : \mathtt{00000000001c0000}]$
CPP	58	55/33	$\mathcal{A}_{10}$ (see below)	$(\mathbf{0}, k_{63}) \to [\mathtt{0000000000000003} : \mathtt{0000000000000001}]$
CPP	59	59/2	$\{46, 51\}$	$(\mathbf{0}, k_{63}) \to [\mathtt{000000000000001f} : \mathtt{000000000000000e}]$
CPP	69	65/1	$\{36\}$	$(\mathbf{0}, k_{63}) \to [\mathtt{00000407ffffffff} : \mathtt{0000040380000000}]$

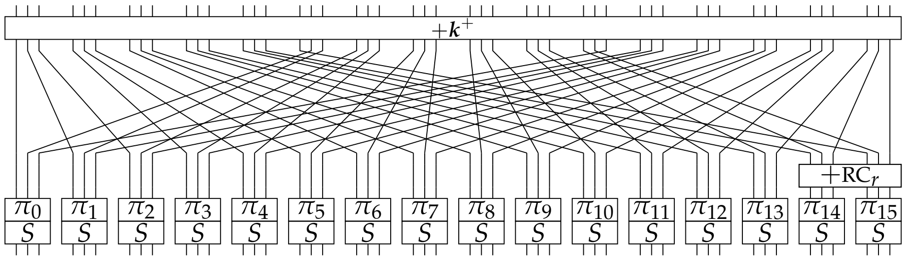
$$\mathcal{A}_0 = \{2,4,5,7,11,17,32,64,66,69,71,73,75\},$$
$$\mathcal{A}_1 = \{1,16,34,55,56,60,62,68\},$$
$$\mathcal{A}_2 = \{0,8,12,14,27,59\},$$
$$\mathcal{A}_{10} = \{3,6,10,15,18,19,20,21,23,24,25,26,28,30,31,33,$$
$$35,37,38,41,43,45,47,49,53,58,63,70,72,74,76,77,79\}$$

9 Linear Cryptanalysis of Round-Reduced PRINTcipher

One of the most recent lightweight cryptographic designs to appear is PRINTcipher [KLPR10]. It is designed by Knudsen et al. and is quite similar to the well-studied PRESENT. All rounds use the same key and differ only by a round counter. The linear layer is partly key-dependent and as a result, 48-bit PRINTcipher uses keys of 80 bits, while 96-bit PRINTcipher uses 160-bit keys. We will focus exclusively on PRINTcipher-48 in this chapter, noting that very similar results can be derived for PRINTcipher-96.

Our first observation relates to the key-dependent bit permutation. We show how there exist several linear trails in PRINTcipher that are biased for some keys but unbiased for most keys, allowing us to distinguish between classes of keys. In order to attack several rounds of PRINTcipher, we need to find many samples. Our second observation uses the identical round-structure, including identical keys, to obtain several samples per plaintext–ciphertext pair. By guessing key bits to do partial encrypting and decrypting, we eventually reach 29 rounds of 48.

Two recent attacks are similar to our work in that they identify classes of weak keys. As a fundamental idea behind PRINTcipher is that the key is burnt into the device, it is straightforward to protect against these attacks by avoiding the weak keys. Avoiding the 2^{52} keys attacked in [LAAZ11] the size of the key space shrinks from 2^{80} to $2^{80} - 2^{52} \approx 2^{80}$ so the entropy is still 80 bits in a practical sense. Similarly, to protect against the attack in [KDH12] the number of keys needs to be lowered to approximately $2^{79.8}$ so there is a loss of one fifth of a bit. In this independent work, we find several classes that are very probable (e.g., probability one half), and even avoiding only the largest classes leads to a key space of size approximately 2^{78}, meaning two bits of the key entropy are effectively lost. This makes our observations very interesting



$+k^+$
$+\mathrm{RC}_r$
π_0 π_1 π_2 π_3 π_4 π_5 π_6 π_7 π_8 π_9 π_{10} π_{11} π_{12} π_{13} π_{14} π_{15}
S

Figure 9.1: One round of PRINTCIPHER.

compared to the previously mentioned results.

This chapter is organized as follows: Section 9.1 describes PRINTCIPHER. Section 9.2 discusses the importance of finding many samples. Some initial, basic observations are given in Section 9.3, before Section 9.4 gives our fundamental observation: a key bit distinguisher on 23 rounds of PRINTCIPHER. Section 9.5 then derives attacks on 27 and 28 rounds of PRINTCIPHER, and shows that several classes of weak keys exist, making the attack very general. In Section 9.6, we show how one can find many samples, and use this to provide an attack on 29-round PRINTCIPHER. Section 9.7 concludes this chapter.

9.1 PRINTCIPHER

We focus on PRINTCIPHER-48, which uses blocks of 48 bits and 80-bit keys.

The 48-bit plaintext is loaded into the state, where we denote the 48 bit positions using indices $0 \leq i < 48$ as usual, but also write $(\lfloor i/3 \rfloor, i \bmod 3)$, e.g., the bit position 14 may be given as $(4,2)$. Thus, the leftmost bit is $(0,0)$ while $(15,2)$ is the rightmost bit.

There are 48 rounds where each round uses a distinct round constant RC_r, $r = 0, 1, \ldots, 47$ (see Table 9.1), a 48-bit xor key $\boldsymbol{k}^+$ (the same in all rounds) and a 32-bit permutation key $\boldsymbol{k}^\pi$ (the same in all rounds). Each round consists of key addition, linear layer (bit permutation), round constant addition, key-dependent bit permutation and an Sbox, see Figure 9.1. The Sbox is given in Table 9.2 and takes input (x_0, x_1, x_2) to produce output (y_0, y_1, y_2).

We denote the plaintext $\boldsymbol{p} = (p_0, \ldots, p_{47})$ and the state after r rounds of encryption $(0 < r \leq 48)$ by $\boldsymbol{c}^r = (c_0^r, \ldots, c_{47}^r)$. For possibly round-reduced versions of PRINTCIPHER, we have $\boldsymbol{c} = \boldsymbol{c}^R$, with $R = 48$ for the original PRINTCIPHER.

The key $\boldsymbol{k} = \boldsymbol{k}^+ || \boldsymbol{k}^\pi = (k_0^+, \ldots, k_{47}^+) || (k_0^\pi, \ldots, k_{31}^\pi)$ is split into an xor key $\boldsymbol{k}^+$

Table 9.1: The six-bit round constants RC_r, given through the hexadecimal representation of $(0,0)||\mathrm{RC}_r$.

r	0	1	2	3	4	5	6	7	8	9	10	11
RC_r	01	03	07	0F	1F	3E	3D	3B	37	2F	1E	3C
r	12	13	14	15	16	17	18	19	20	21	22	23
RC_r	39	33	27	0E	1D	3A	35	2B	16	2C	18	30
r	24	25	26	27	28	29	30	31	32	33	34	35
RC_r	21	02	05	0B	17	2E	1C	38	31	23	06	0D
r	36	37	38	39	40	41	42	43	44	45	46	47
RC_r	1B	36	2D	1A	34	29	12	24	08	11	22	04

Table 9.2: The PRINTcipher Sbox mapping input $\boldsymbol{x} = (x_0, x_1, x_2)$ to output $\boldsymbol{y} = S(\boldsymbol{x}) = (y_0, y_1, y_2)$.

$\boldsymbol{x}$	(0,0,0)	(1,0,0)	(0,1,0)	(1,1,0)
$S(\boldsymbol{x})$	(0,0,0)	(1,1,1)	(0,1,1)	(1,0,1)
$\boldsymbol{x}$	(0,0,1)	(1,0,1)	(0,1,1)	(1,1,1)
$S(\boldsymbol{x})$	(0,0,1)	(1,0,0)	(1,1,0)	(0,1,0)

(48 bits) and a permutation key $\boldsymbol{k}^{\pi}$ (32 bits).

The linear layer is a bit permutation Π, given by Table 9.3 and seen in Figure 9.1.

Further, a bit permutation π_b is applied to each disjoint triplet of bits in the state of PRINTcipher. The new positions of bits $(b,0)$, $(b,1)$, and $(b,2)$ are $(b,\pi_b(0))$, $(b,\pi_b(1))$, and $(b,\pi_b(2))$, respectively, where $(\pi_b(0),\pi_b(1),\pi_b(2))$ are determined by the two key bits $(k^{\pi}_{2b}, k^{\pi}_{2b+1})$. Note that π_b with an integer argument is an integer describing the behavior of π_b with a vector argument. This mapping is given in Table 9.4. Note in particular how one of the permutations is trivial while the others fix one bit while switching the two remaining bits. Thus, the two permutations that shift the three-bit word cyclically have been excluded from PRINTcipher and can not be selected by the key.

9.1.1 EXISTING WORK ON PRINTcipher

Abdelraheem et al. have given a differential attack on 22-round PRINTcipher [ALZ11]. Using the entire code book, they study the single-bit differentials in order to learn how the bits are permuted through the entire cipher, i.e., R rounds. Finding the Rth root of this permutation then gives them the single-round permutation $\pi \circ \Pi$ and thus $\boldsymbol{k}^{\pi}$.

We note that it is straightforward to invert the last Sbox upon retrieving a

Table 9.3: The linear layer Π in PRINTcipher. Bits at positions 'In' are moved to positions 'Out'.

In	Out	In	Out	In	Out	In	Out
(0,0)	(0,0)	(4,0)	(12,0)	(8,0)	(8,1)	(12,0)	(4,2)
(0,1)	(1,0)	(4,1)	(13,0)	(8,1)	(9,1)	(12,1)	(5,2)
(0,2)	(2,0)	(4,2)	(14,0)	(8,2)	(10,1)	(12,2)	(6,2)
(1,0)	(3,0)	(5,0)	(15,0)	(9,0)	(11,1)	(13,0)	(7,2)
(1,1)	(4,0)	(5,1)	(0,1)	(9,1)	(12,1)	(13,1)	(8,2)
(1,2)	(5,0)	(5,2)	(1,1)	(9,2)	(13,1)	(13,2)	(9,2)
(2,0)	(6,0)	(6,0)	(2,1)	(10,0)	(14,1)	(14,0)	(10,2)
(2,1)	(7,0)	(6,1)	(3,1)	(10,1)	(15,1)	(14,1)	(11,2)
(2,2)	(8,0)	(6,2)	(4,1)	(10,2)	(0,2)	(14,2)	(12,2)
(3,0)	(9,0)	(7,0)	(5,1)	(11,0)	(1,2)	(15,0)	(13,2)
(3,1)	(10,0)	(7,1)	(6,1)	(11,1)	(2,2)	(15,1)	(14,2)
(3,2)	(11,0)	(7,2)	(7,1)	(11,2)	(3,2)	(15,2)	(15,2)

Table 9.4: The key-dependent bit permutation. The bits at positions $(b,0)$, $(b,1)$, and $(b,2)$ are moved to positions $(b,\pi_b(0))$, $(b,\pi_b(1))$, and $(b,\pi_b(2))$, respectively where $(\pi_b(0),\pi_b(1),\pi_b(2))$ are determined by $(k^{\pi}_{2b},k^{\pi}_{2b+1})$.

$(k^{\pi}_{2b},k^{\pi}_{2b+1})$	$(\pi_b(0),\pi_b(1),\pi_b(2))$
(0,0)	(0,1,2)
(0,1)	(1,0,2)
(1,0)	(0,2,1)
(1,1)	(2,1,0)

ciphertext (it is present only to make hardware implementations smaller as it does not require any special logic for the last round as in, e.g., AES). Thus, an attacker can extend the 22-round attack to 23 rounds at a very low cost. The Sbox only has to be inverted if the three bits in its output are the only bits that have a difference.

Leander et al. [LAAZ11] showed how an »invariant subspace attack« allowed for a class of 2^{52} keys to be distinguished regardless of the number of rounds, so in particular for the full PRINTcipher. This will be covered more in Chapter 10. Karakoç et al. [KDH12] combined differential and linear cryptanalysis to reach 29 rounds on 4.54% and 31 rounds on .036% of the keys.

9.2 ON THE IMPORTANCE OF FINDING MANY SAMPLES

In this chapter, we will exclusively deal with single-bit trails such as

$$\mathbf{Pr}\left[p_0 = c_0\right] = \frac{1}{2} + \varepsilon,$$

possibly involving the xor of one bit of key,

$$\mathbf{Pr}\left[p_0 = c_0 + \langle \gamma, \boldsymbol{k} \rangle\right] = \frac{1}{2} + \varepsilon,$$

although it is no doubt possible to find many more trails by using multiple-bit trails. The reason we do this is that the single-bit trails appear very naturally in PRINTcipher.

An attacker will try to find relations with as large bias as possible. The single-bit characteristics used have correlation $\pm 2^{-1}$, and assuming independence, the piling-up lemma gives that linear trails over r' rounds of PRINTcipher have correlation $2^{-r'}$, i.e., bias $2^{-r'-1}$.

Recall from Section 3.4 that trying to distinguish a bias ε requires about ε^{-2} samples, e.g., $p_0 + c_0$. Thus, we will need to obtain $2^{2r'+2}$ samples to use an approximation over r' rounds.

One can only obtain 2^{48} distinct plaintext–ciphertext pairs on PRINTcipher, which seems to indicate that only 2^{48} samples can be found and that only 23-round trails can be used, i.e., less than half the number of rounds. If we want to use a trail on $(23+s)$ rounds, we need to obtain $2^{2(23+s)+2} = 2^{48+2s}$ samples, i.e., 2^{2s} samples per plaintext–ciphertext pair.

In this chapter, we will note how some particular features of PRINTcipher allow us to find trails where we can access several samples per plaintext–ciphertext pair. We also see how these samples are independent (enough) to make them usable in a cryptanalytic setting.

We will only consider iterated trails, i.e., trails beginning and ending at a common bit position. This is for simplicity: iterated trails can be used to trivially create trails on larger numbers of rounds. One can also see that by using iterated trails, the number of distinct π_b involved in the trail is kept to a minimum, which keeps the involved number of key bits decently small.

A sample $s_j = p_0^j + c_0^j$ (say) is a bit obtained by comparing a plaintext bit to a ciphertext bit, and the attacker will compute $S = \sum_j s_j$. Kaliski and Robshaw [KR94] noted that if one can find several linear approximations that involve the exact same key bits, i.e., the same bitmask γ, so that one can get several counts $S^i = \sum_j s_j^i$, one can use a weighted sum of these counts S^i—this measurement has the same expected value but a smaller variance. In particular, when the bias is the same for all linear approximations, the

Table 9.5: The Sbox evaluated for all possible permutations of the input.

(x_0,x_1,x_2)	$S(x_0,x_1,x_2)$	$S(x_1,x_0,x_2)$	$S(x_0,x_2,x_1)$	$S(x_2,x_1,x_0)$
(0,0,0)	(0,0,0)	(0,0,0)	(0,0,0)	(0,0,0)
(1,0,0)	(1,1,1)	(0,1,1)	(1,1,1)	(0,0,1)
(0,1,0)	(0,1,1)	(1,1,1)	(0,0,1)	(0,1,1)
(1,1,0)	(1,0,1)	(1,0,1)	(1,0,0)	(1,1,0)
(0,0,1)	(0,0,1)	(0,0,1)	(0,1,1)	(1,1,1)
(1,0,1)	(1,0,0)	(1,1,0)	(1,0,1)	(1,0,0)
(0,1,1)	(1,1,0)	(1,0,0)	(1,1,0)	(1,0,1)
(1,1,1)	(0,1,1)	(0,1,1)	(0,1,1)	(0,1,1)

weighted sum is simply the average, which up to a multiplicative constant is the same as $\sum_{i,j} s_j^i$, i.e., the overall number of samples that are 1. It is then natural to think of the different s_j^i (with varying i and j) as different samples from the same underlying distribution.

9.3 SOME INITIAL OBSERVATIONS

The correlation matrix for the Sbox is

$$C_S = 2^{-3}\begin{pmatrix} 8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 4 & 0 & 4 & 0 & 4 & 0 & -4 \\ 0 & 0 & 4 & 4 & 0 & 0 & -4 & 4 \\ 0 & 4 & 4 & 0 & 0 & -4 & 4 & 0 \\ 0 & 0 & 0 & 0 & -4 & 4 & 4 & 4 \\ 0 & -4 & 0 & 4 & 4 & 0 & 4 & 0 \\ 0 & 0 & 4 & -4 & 4 & 4 & 0 & 0 \\ 0 & 4 & -4 & 0 & 4 & 0 & 0 & 4 \end{pmatrix}, \tag{9.1}$$

where the single-bit biased linear approximations have been grayed. Through the remainder of this chapter, we will focus on them:

$$\mathbf{Pr}\left[y_0 = x_0\right] = \mathbf{Pr}\left[y_1 = x_1\right] = \mathbf{Pr}\left[y_2 = x_2 + 1\right] = \frac{1}{2} + 2^{-2}.$$

They are conveniently all from x_i to y_i, which is not strictly necessary but simplifies the presentation of the subsequent observations and attacks.

9.3.1 THE PERMUTATION π_b AND THE SBOX

With $(y_0, y_1, y_2) = S(x_0, x_1, x_2)$ and $(y_0', y_1', y_2') = S(x_0, x_2, x_1)$, one can quite easily see that we always have $y_0 = y_0'$, see Table 9.5. This means that if we are only interested in tap 0 out of the Sbox, it does not matter whether x_1, x_2 are swapped or not before entering the Sbox.

As a consequence, if we

- know three bits that enter $S \circ \pi_b$,
- want to know y_0 out of the Sbox, and
- need to guess the permutation π_b, i.e., $(k^\pi_{2b}, k^\pi_{2b+1})$,

then we only need to make three guesses on π_b.

The same property shows up on y_2 also, but not on y_1, see Table 9.5. We will use this observation to reduce the amount of guesswork we need to perform during partial encryption. We will use the notation π^3_b to mark that we only guess a ternary digit, a trit, for π_b due to these properties.

Similarly, when we guess for a partial decryption, we often do not need to guess the whole permutation π_b, i.e., two bits, but only how it permutes one particular bit. Keeping the notation from Section 9.1, we will write $\pi_b(i)$, $0 \leq i \leq 2$, to indicate that we guess how x_i is permuted by π_b.

9.4 A KEY BIT DISTINGUISHER

We will use a variant of linear cryptanalysis. We study single-bit trails that are biased for certain classes of keys and nonbiased for other keys. As a very nondetailed example, consider a trail from the leftmost bit to the leftmost bit. It is readily apparent from Figure 9.1 that such a trail exists and that it is iterated (although it is of course not obvious from the figure that it is biased). We claim that we can distinguish individual bits of $\boldsymbol{k}^\pi$ using this trail. It is biased for half the keys and nonbiased for the other half. Thus, if we can distinguish between these two distributions (i.e., if the bias is large enough and we have sufficiently many samples) we can determine the value of this key bit.

9.4.1 A DETAILED EXAMPLE

We now describe how to distinguish between two distributions. One where k^π_1 is zero, and one where it is one. This allows for a partial-key recovery, i.e., learning one bit of the key, faster than brute force.

Note that $\Pi(0,0) = (0,0)$, and that for two of four keys, $\pi_0(0) = 0$. This happens precisely when $k^\pi_1 = 0$ (see Table 9.6).

Thus, with $k^\pi_1 = 0$, $(\pi \circ \Pi)(0,0) = (0,0)$. The probability that this bit then passes the Sbox unaltered is $\frac{3}{4}$, so after a single round of encryption, we have

$$\mathbf{Pr}\left[c^1_0 = p_0 + k^+_0\right] = \frac{1}{2} + 2^{-2}.$$

Table 9.6: How the individual bits $(0,1,2)$ are moved by the key-dependent permutation π_b, and for which keys $(k^{\pi}_{2b}, k^{\pi}_{2b+1})$ it happens.

Bit Move	$(k^{\pi}_{2b}, k^{\pi}_{2b+1})$	Bit Move	$(k^{\pi}_{2b}, k^{\pi}_{2b+1})$	Bit Move	$(k^{\pi}_{2b}, k^{\pi}_{2b+1})$
$0 \to 0$	(0,0),(1,0)	$1 \to 0$	(0,1)	$2 \to 0$	(1,1)
$0 \to 1$	(0,1)	$1 \to 1$	(0,0),(1,1)	$2 \to 1$	(1,0)
$0 \to 2$	(1,1)	$1 \to 2$	(1,0)	$2 \to 2$	(0,0),(0,1)

For two rounds, we have

$$\mathbf{Pr}\left[c_0^2 = p_0\right] = \frac{1}{2} + 2^{-3},$$

assuming independence, as the xor key bits cancel. Generalizing to any even number of rounds, we have

$$\mathbf{Pr}\left[c_0^R = p_0\right] = \frac{1}{2} + 2^{-R-1}.$$

For PRINTcipher on 22 rounds, we would need almost the entire code book, 2^{46} plaintext–ciphertext pairs.

We can also use the full code book, of size 2^{48}, to attack 23 rounds. We then have an odd number of rounds, and the key bit k_0^+ shows up, so we utilize the relation

$$\mathbf{Pr}\left[c_0^R = p_0 + k_0^+\right] = \frac{1}{2} + 2^{-R-1}, \tag{9.2}$$

with $R = 23$. Things then get slightly more tricky, as we can learn more about the key but would need to distinguish between three distributions:

1. $c_0^R = p_0$ with probability $\frac{1}{2}$, implying $k_1^{\pi} = 1$.
2. $c_0^R = p_0$ with high probability, implying $k_1^{\pi} = 0$ and $k_0^+ = 0$.
3. $c_0^R = p_0$ with low probability, implying $k_1^{\pi} = 0$ and $k_0^+ = 1$.

9.4.2 MORE LINEAR TRAILS ON ONE ROUND OF PRINTcipher

There are in total four iterated single-bit, single-round trails, and we list them in Table 9.7. Some constants arise as the Sbox flips bit 2 with probability $\frac{3}{4}$ rather than preserves it, and as bits of RC_r enter. We define $I_R = R \bmod 2$.

Table 9.7: The iterated single-round trails on PRINTcipher, extended to several rounds. All trails have bias 2^{-R-1}.

Trail	Requirement
$c^R_{47} = p_{47} + k^+_{47} I_R + d_R$	$k^\pi_{30} = 0$
$c^R_{24} = p_{24} + k^+_{24} I_R$	$(k^\pi_{16}, k^\pi_{17}) = (0,1)$
$c^R_{23} = p_{23} + k^+_{23} I_R + I_R$	$(k^\pi_{14}, k^\pi_{15}) = (1,0)$
$c^R_0 = p_0 + k^+_0 I_R$	$k^\pi_1 = 0$
$d_R = \left(R + 1 + \sum_{0 \le r < R} \mathrm{RC}_r\right) \bmod 2$	

Table 9.8: The bits and trits required for encryption, decryption, and both, when encrypting/decrypting two rounds to access the bits at position $(0,0)$.

Encryption	k^+_5, k^+_{10}, k^+_{16}, k^+_{21}, k^+_{26}, k^+_{32}, k^+_{37}, k^+_{42}, π_5, π^3_{10}
Decryption	k^+_1, k^+_2, $\pi_1(0)$, $\pi_2(0)$
Both	k^+_0

9.5 GUESSING KEY BITS FOR PARTIAL ENCRYPTION AND DECRYPTION

The above observation can be used as-is to mount an attack on 23-round PRINTcipher, recovering up to three bits of the key, but it is straightforward to derive an even more powerful attack on 27 rounds of PRINTcipher: if a guessed partial key is correct, we should observe the bias, while if the guess is bad, the behavior should be (more) random.

First, we assume that $k^\pi_1 = 0$, meaning our attack only works for a fraction 2^{-1} of the keys. Then, we aim to decrypt two rounds at the end and encrypt two rounds at the top of PRINTcipher. Thus, we need to guess the bits and trits listed in Table 9.8. There are in total $N = 2^{13} \cdot 3^3 \approx 2^{17.8}$ guesses. See Figure 9.2 for an overview of the partial calculations.

Due to the property observed in Subsection 9.3.1, we do not need to guess k^π_0. We have assumed $k^\pi_1 = 0$ to fix $\pi_0(0) = 0$ and this is enough to predict tap 0 out of the Sbox. It does not matter whether π_0 is trivial or swaps bits 1 and 2.

We call the plaintext (resp. ciphertext) bits that affect the partial encryption (resp. decryption) necessary to derive the state bits we are interested in *active*. There are nine active bits in the plaintext and nine in the ciphertext. For a plaintext–ciphertext pair $(\boldsymbol{p}, \boldsymbol{c})$ we can collect these bits into an eighteen-bit word $\boldsymbol{w} = (p_0, p_5, \ldots, c_8)$.

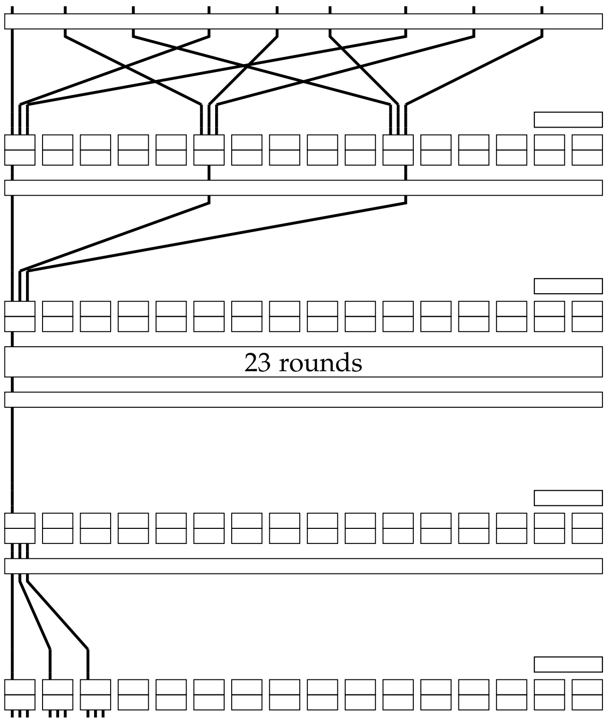


Figure 9.2: Performing two rounds of partial encryption and decryption to access the bits at position $(0,0)$.

We describe the attack. Acquire all 2^{48} plaintext–ciphertext pairs $(\boldsymbol{p}^j, \boldsymbol{c}^j)$. Categorize them according to the active bits, i.e., for each possible word $\boldsymbol{w}$, count how often it appears. Denote these counters $R_{\boldsymbol{w}}$. This is the data collection part of the attack.

We then begin analyzing the data. For each plaintext–ciphertext pair and for each guess of key material, denoted by $\boldsymbol{k}^i$, $0 \leq i < N$, we will calculate two rounds of encryption and decryption, $\hat{\boldsymbol{p}}^j = F_{0,2}(\boldsymbol{p}^j, \boldsymbol{k}^i)$, $\hat{\boldsymbol{c}}^j = F^{-1}_{25,27}(\boldsymbol{c}^j, \boldsymbol{k}^i)$, and count how often $\hat{c}^j_0 = \hat{p}^j_0 + k^{+i}_0$. This is done using N counters S_i. An efficient way of doing this [Mat94a] is to use the counters $R_{\boldsymbol{w}}$. For each word $\boldsymbol{w}$ and each key guess $\boldsymbol{k}^i$, we perform the partial calculations $\hat{\boldsymbol{p}}^{i,w} = F_{0,2}(\boldsymbol{p}^w, \boldsymbol{k}^i)$, $\hat{\boldsymbol{c}}^{i,w} = F^{-1}_{25,27}(\boldsymbol{c}^w, \boldsymbol{k}^i)$. Here, $\boldsymbol{p}^w$ ($\boldsymbol{c}^w$) is some plaintext (ciphertext) with the correct active bits as determined by $\boldsymbol{w}$. If $\hat{c}^{i,w}_0 = \hat{p}^{i,w}_0 + k^{+i}_0$, we add $R_{\boldsymbol{w}}$ to S_i.

By sorting all S_i, we can get a ranking of the different guesses. We choose the most likely guess, brute force all nonguessed bits and hopefully recover the key. If recovery fails, we choose the second most likely guess, etc. The exact number of bits that need to be brute forced will be different for different guesses. Where we guessed a trit, e.g., $\pi_1(0)$, we will have recovered one or two bits of (k_2^π, k_3^π). As long as the correct guess is ranked on the upper half of the sorted list of counters, the entire key will be found faster than what can be expected from a brute force (2^{79} due to one guessed bit).

The counters R_w are used for saving time [Mat94a]. Several other improvements can be made, also from [Mat94a]. We should not make $2^{18} \cdot N$ partial encryptions and decryptions. First, the plaintext and ciphertext operations can be separated completely, so that we only need to make $2^9 \cdot N + 2^9 \cdot N$ encryptions/decryptions. Second, since the overlap in encryption and decryption with respect to the guessed bits is very small, we only need to perform $2^9 \cdot N_\mathrm{e} + 2^9 \cdot N_\mathrm{d}$ encryptions/decryptions where N_e (N_d) is the number of key guesses that actually affect the encryption (decryption). Third, doing two complete PRINTcipher rounds in both directions is unnecessary as we only need to perform partial rounds calculations, i.e., use some small number of Sboxes.

9.5.1 EXPERIMENTAL RESULTS

We have implemented this attack on $7+4=11$ rather than $23+4=27$ rounds of PRINTcipher. This means that we guess the same bits and perform the same partial encryptions, but that the bias is larger so that it is feasible for us to perform many attacks in order to gather statistics.

It turns out that the attack does indeed give an advantage. Attacking 2^{20} different weak keys, the median ranking of the key is approximately $2^{12.2}$ out of $N = 2^{13} \cdot 3^3$, i.e., the median time required for identifying the entire key is approximately $2^{73.5}$.

9.5.2 ANALYZING THE TIME REQUIREMENT

The attack consists of data collection and data analysis. The latter in turn consists of 1) deriving two sets of counters, N_e for encryption and N_d for decryption, and 2) combining these to find N counters. If the number of active bits in the plaintext (ciphertext) is denoted a_e (a_d) and the number of active Sboxes in encryption (decryption) is denoted A_e (A_d), the time requirements are given by

$$\begin{aligned} T^{\text{collect}} &= \varepsilon^{-2}, \\ T^{\text{count}} &= \frac{2^{a_\mathrm{e}} N_\mathrm{e} A_\mathrm{e} + 2^{a_\mathrm{d}} N_\mathrm{d} A_\mathrm{d}}{16 \cdot R}, \\ T^{\text{combine}} &= 2^{a_\mathrm{e}+a_\mathrm{d}} N. \end{aligned}$$

Table 9.9: The bits and trits required for encryption, decryption, and both, when encrypting/decrypting two/three rounds to access the bits at position $(0,0)$. Note the overlap in π_5.

Encryption	$k^+_{10}, k^+_{16}, k^+_{21}, k^+_{26}, k^+_{32}, k^+_{37}, k^+_{42}, \pi_5, \pi^3_{10}$
Decryption	$k^+_1, k^+_2, k^+_3, k^+_4, k^+_6, k^+_7, k^+_8, \pi_1(0), \pi_2(0), \pi_3(0), \pi_4(0), \pi_5(0), \pi_6(0), \pi_7(0), \pi_8(0)$
Both	k^+_5, k^+_0

The first two measurements are normalized to R-round PRINTcipher evaluations, while the last describes the number of simple bit and integer operations needed to calculate the counters S_i.

For the specific attack detailed above, we have $N_e = 2^{11} \cdot 3$ and $N_d = 2^3 \cdot 3^2$. Since $(a_e, a_d, A_e, A_d) = (9, 9, 4, 4)$, the requirements are

$$\begin{aligned}
T^{\text{collect}} &= 2^{2 \cdot 24} = 2^{48},\\
T^{\text{count}} &= \frac{2^9 \cdot 2^{11} \cdot 3 \cdot 4 + 2^9 \cdot 2^3 \cdot 3^2 \cdot 4}{16 \cdot 27} \approx 2^{15},\\
T^{\text{combine}} &= 2^{9+9} \cdot N = 2^{18} \cdot 2^{13} \cdot 3^3 \approx 2^{36}.
\end{aligned}$$

This suggests that the most time consuming part is the data collection where we need to generate and look at 2^{48} plaintext–ciphertext pairs.

9.5.3 A POSSIBLE IMPLEMENTATION IMPROVEMENT

Consider a key with $(k^+_5, k^+_{37}, k^\pi_{10}, k^\pi_{11}) = (0,0,0,0)$. Then

$$c^1_{16} = \langle (0,1,0), S(p_5, p_{21} + k^+_{21}, p_{37}) \rangle$$

(cf. Figure 9.2). If, on the other hand, $(k^+_5, k^+_{37}, k^\pi_{10}, k^\pi_{11}) = (1,1,1,1)$, we have

$$c^1_{16} = \langle (0,1,0), S(p_{37} + 1, p_{21} + k^+_{21}, p_5 + 1) \rangle.$$

A straightforward calculation shows that these two values of c_{16} are always the same. This was observed in [KLPR10]. Thus, these two configurations of $(k^+_5, k^+_{37}, k^\pi_{10}, k^\pi_{11})$ always yield the exact same value of c^2_0 (cf. Figure 9.2). Further, as these key bits are not involved in the guessed decryption, it is not possible to distinguish between two such key candidates. There are several such pairs, and some groups of four keys that are equivalent in this sense. This property allows N_e, N_d, and T^{count} to be reduced to approximately half. (Once key candidates have been ranked, it is not enough to consider only one of such »equivalent« keys.) This property is not considered in the sequel.

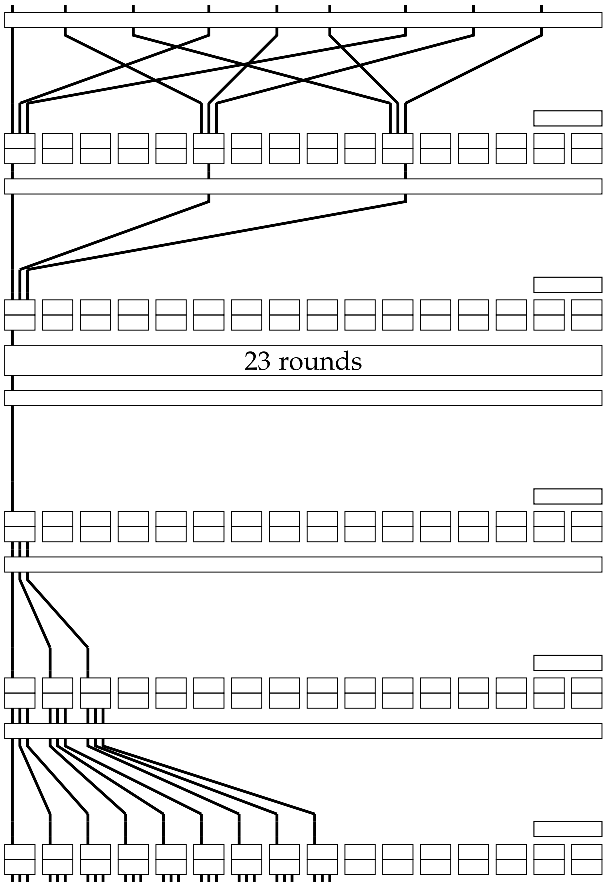


Figure 9.3: Performing two/three rounds of partial encryption/decryption to access the bits at position $(0, 0)$.

9.5.4 REACHING THE LIMIT: 28 ROUNDS

We note that in the attacks on 27 rounds, guessing and encrypting is more expensive than guessing and decrypting. During decryption, we first invert S, and then only need to control one bit in some π_b. On the other hand, during encryption, we need to fully control the permutation, so that we can calculate all three bits that go into the Sbox. This leads to more expensive guesswork on π_b and especially on $\boldsymbol{k}^+$. Thus, the natural approach for extending the attack by one round is to add another round in the partial decryption, see Figure 9.3.

In Table 9.9, we list the bits and trits involved in partially decrypting and encrypting from 28 rounds to 23. The attack requires $N = 2^{18} \cdot 3^8 \approx 2^{30.7}$

Table 9.10: The bits and trits required for encryption, decryption, and both, when encrypting/decrypting two/three rounds to access the bits at position $(3,1)/(11,2)$.

Encryption	$k^+_1, k^+_3, k^+_6, k^+_{17}, k^+_{19}, k^+_{22}, k^+_{38}, k^+_{43}, \pi^3_1, \pi_6,$
Decryption	$k^+_9, k^+_{10}, k^+_{28}, k^+_{29}, k^+_{30}, k^+_{31}, k^+_{32}, k^+_{34}, k^\pi_{23}, \pi_0(2),$ $\pi_1(2), \pi_2(2), \pi_3(2), \pi_9(1), \pi_{12}(1), \pi_{13}(1), \pi_{15}(1)$
Both	$k^+_{11}, k^+_{27}, k^+_{33}, k^+_{35}$

guesses, partitioned as $N_\mathrm{e} = 2^{11} \cdot 3$ and $N_\mathrm{d} = 2^9 \cdot 3^8$. Counting active bits and Sboxes gives $(a_\mathrm{e}, a_\mathrm{d}, A_\mathrm{e}, A_\mathrm{d}) = (9, 27, 4, 13)$, so we have

$$T^{\text{collect}} = 2^{2\cdot 24} = 2^{48},$$
$$T^{\text{count}} = \frac{2^9 \cdot 2^{11} \cdot 3 \cdot 4 + 2^{27} \cdot 2^9 \cdot 3^8 \cdot 13}{16 \cdot 28} \approx 2^{44},$$
$$T^{\text{combine}} = 2^{9+27} \cdot N = 2^{36} \cdot 2^{18} \cdot 3^8 \approx 2^{67}.$$

Attacking 28 rounds seems to be the best we can do. Using a single trail, we have not been able to go beyond 28 rounds while keeping the attack costs below exhaustive search.

9.5.5 MORE ATTACKS ON 27/28 ROUNDS

We can use basically any trail on 23 rounds to create attacks on 27/28 rounds. The trail on bit $(0,0)$ is nice as the partial encryptions and decryptions involve few bits of $\boldsymbol{k}^+$ and $\boldsymbol{k}^\pi$, due to Sbox reuse. Most other trails involve more guesswork. As an example, with $(k^\pi_6, k^\pi_7, k^\pi_{21}, k^\pi_{22}, k^\pi_{28}) = (1, 0, 0, 0, k^\pi_{29})$,

$$\mathbf{Pr}\left[c^{25}_{35} = c^2_{10} + k^+_{35} + 1\right] = \frac{1}{2} + 2^{-24}.$$

Using this, we can build an attack on 28 rounds. Here, $N = 2^{27} \cdot 3^6 \approx 2^{36.5}$, $N_\mathrm{e} = 2^{14} \cdot 3$, $N_\mathrm{d} = 2^{13} \cdot 3^7$ and $(a_\mathrm{e}, a_\mathrm{d}, A_\mathrm{e}, A_\mathrm{d}) = (9, 27, 4, 13)$. The time requirements are

$$T^{\text{collect}} = 2^{2\cdot 24} = 2^{48},$$
$$T^{\text{count}} = \frac{2^9 \cdot 2^{14} \cdot 3 \cdot 4 + 2^{27} \cdot 2^{13} \cdot 3^7 \cdot 13}{16 \cdot 28} \approx 2^{46},$$
$$T^{\text{combine}} = 2^{9+27} \cdot N = 2^{36} \cdot 2^{27} \cdot 3^6 \approx 2^{73}.$$

The bits and trits guessed are listed in Table 9.10.

9.5.6 ON FALSE POSITIVES

By piling the single-round trail on the leftmost bit, we see that, e.g.,

$$\mathbf{Pr}\left[c_0^{10} = p_0\right] = \frac{1}{2} + 2^{-11}$$

when $k_1^\pi = 0$. This clearly assumes independence, i.e., in terms of the linear hull (cf. Equation 4.7), it is assumed that the correlations of all other trails add up to zero. This assumption being wrong might cause false negatives and false positives to occur.

As another source of false positives, we note that there are several other ways of obtaining this bias.

All in all, there are 102 different trails, only using single-bit round characteristics, from $(0,0)$ to $(0,0)$ over ten rounds, each corresponding to a different class of keys. This means that a biased distribution can be explained by any of these trails, and thus by any of these classes. Due to this, an attacker will prefer to use short, iterated trails involving few bits of $\boldsymbol{k}^\pi$.

9.6 ON MORE ROUNDS OF PRINTcipher: MIRRORING TRAILS

We generalize our observation slightly and give an example two-round trail: with $(k_8^\pi, k_9^\pi, k_{25}^\pi) = (1,1,0)$,

$$\mathbf{Pr}\left[c_{36}^2 = p_{36} + k_{36}^+ + k_{12}^+\right] = \frac{1}{2} + 2^{-3}.$$

Note in particular how there is a mirroring trail,

$$\mathbf{Pr}\left[c_{12}^2 = p_{12} + k_{36}^+ + k_{12}^+\right] = \frac{1}{2} + 2^{-3},$$

see Figure 9.4. The mirroring trail depends on the exact same key configuration and allows us to collect *two* samples with every plaintext–ciphertext pair. We show in Subsection 9.6.2 that this works, i.e., the samples can be considered to be independent.

We do not give all the two-round trails on PRINTcipher, as they will not be used in the sequel. We only note that due to the structure of PRINTcipher, every Sbox is used precisely once so far in this chapter—either in one trail on one round (Sboxes 0, 7, 8, 15), or in two mirroring trails on two rounds.

As a particular four-round trail that we will use later, we give

$$\mathbf{Pr}\left[c_{10}^4 = p_{10} + k_{10}^+ + k_{30}^+ + k_{43}^+ + k_{35}^+ + 1\right] = \frac{1}{2} + 2^{-5}, \tag{9.3}$$

which is activated by $(k_6^\pi, k_7^\pi, k_{21}^\pi, k_{22}^\pi, k_{28}^\pi) = (1,0,0,0,k_{29}^\pi)$, see Figure 9.5.

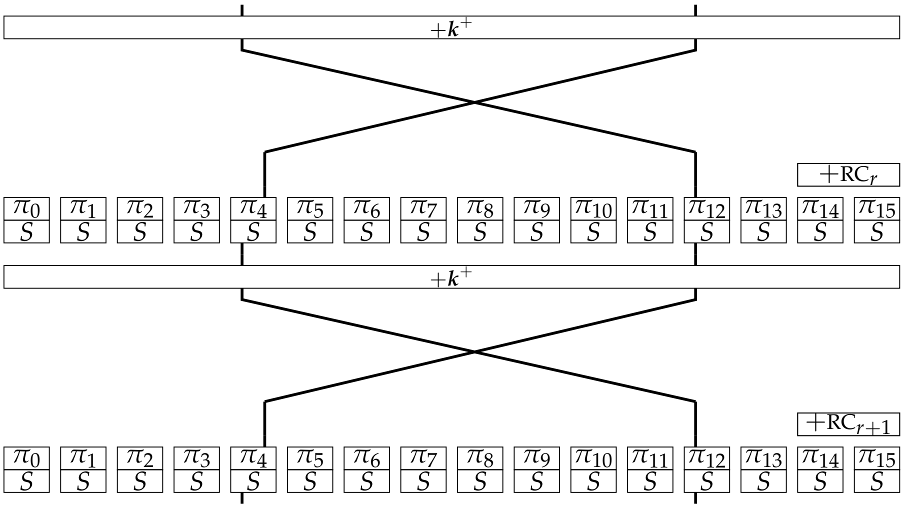

$+k^+$
$+\mathrm{RC}_r$
π_0 π_1 π_2 π_3 π_4 π_5 π_6 π_7 π_8 π_9 π_{10} π_{11} π_{12} π_{13} π_{14} π_{15}
S
$+k^+$
$+\mathrm{RC}_{r+1}$

Figure 9.4: Two mirroring trails on two rounds of PRINTCIPHER. Both trails are activated by $(k_8^\pi, k_9^\pi, k_{25}^\pi) = (1, 1, 0)$.

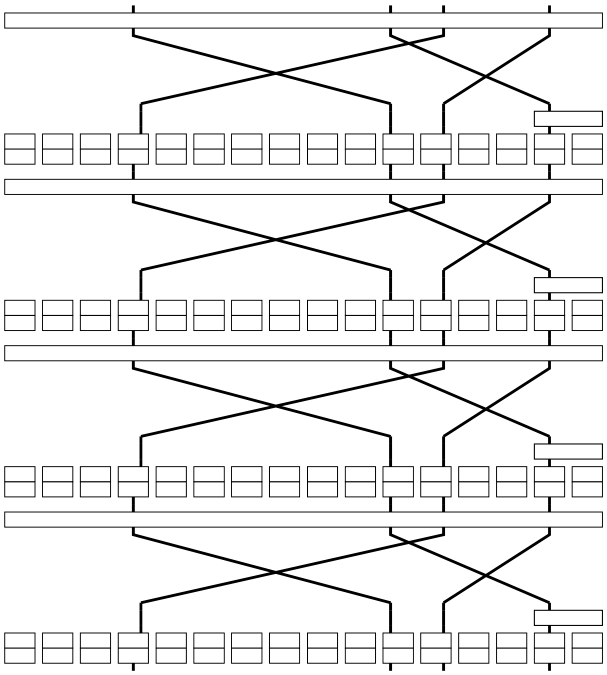


Figure 9.5: Four mirroring trails over four rounds of PRINTCIPHER.

Table 9.11: The iterated trails on eight rounds ($R = 8$) composed from four-round iterated trails, depending only on five bits of $\boldsymbol{k}^{\pi}$. All trails have bias 2^{-R-1}, and the constants e_j^R arise from the round constants RC_r. The trails are easily extended to, e.g., 24 rounds ($R = 24$), in which case only the constants need to be rechecked.

Trail	Sboxes	Trail	Sboxes
$c_{10}^R = p_{10} + e_{10}^R$	$10, 14, 11, 3, \ldots$	$c_4^R = p_4$	$4, 12, 5, 1, \ldots$
$c_{30}^R = p_{30} + e_{30}^R$	$14, 11, 3, 10, \ldots$	$c_{12}^R = p_{12}$	$12, 5, 1, 4, \ldots$
$c_{35}^R = p_{35} + e_{35}^R$	$3, 10, 14, 11, \ldots$	$c_{17}^R = p_{17}$	$1, 4, 12, 5, \ldots$
$c_{43}^R = p_{43} + e_{43}^R$	$11, 3, 10, 14, \ldots$	$c_{37}^R = p_{37}$	$5, 1, 4, 12, \ldots$
$(k_6^\pi, k_7^\pi, k_{21}^\pi, k_{22}^\pi, k_{28}^\pi) = (1, 0, 0, 0, k_{29}^\pi)$		$(k_2^\pi, k_9^\pi, k_{10}^\pi, k_{24}^\pi, k_{25}^\pi) = (k_3^\pi, 0, 0, 0, 1)$	
Trail	Sboxes	Trail	Sboxes
$c_{24}^R = p_{24}$	$8, 9, 13, 8, \ldots$	$c_7^R = p_7$	$7, 7, 6, 2, \ldots$
$c_{25}^R = p_{25}$	$9, 13, 8, 8, \ldots$	$c_{18}^R = p_{18}$	$2, 7, 7, 6, 6, \ldots$
$c_{29}^R = p_{29}$	$13, 8, 8, 9, \ldots$	$c_{22}^R = p_{22}$	$6, 2, 7, 7, \ldots$
$c_{40}^R = p_{40}$	$8, 8, 9, 13, \ldots$	$c_{23}^R = p_{23}$	$7, 6, 2, 7, \ldots$
$(k_{16}^\pi, k_{17}^\pi, k_{18}^\pi, k_{19}^\pi, k_{26}^\pi) = (1, 1, 1, 0, k_{27}^\pi)$		$(k_4^\pi, k_{12}^\pi, k_{13}^\pi, k_{14}^\pi, k_{15}^\pi) = (k_5^\pi, 0, 1, 1, 1)$	
Constants ($R = 8$)		Constants ($R = 24$)	
$(e_{10}^8, e_{30}^8, e_{35}^8, e_{43}^8) = (1, 1, 1, 1)$		$(e_{10}^{24}, e_{30}^{24}, e_{35}^{24}, e_{43}^{24}) = (1, 1, 0, 1)$	

9.6.1 USING MIRRORING TRAILS TO DISTINGUISH ON 24-ROUND TRAILS

We will now construct 24-round trails with bias 2^{-25}. By using trails that allow four samples per plaintext–ciphertext pair, we can get in total 2^{50} samples, allowing us to distinguish the distribution.

The best iterated trails on 24 rounds are given in Table 9.11. They are best in the sense that they use a small number of key bits (5), yet allow four mirroring trails each, so that we can get the required number of samples. They are constructed from iterated four-round trails, that we have piled in order to cancel the bits that appear from $\boldsymbol{k}^{+}$ (cf. Subsection 9.4.1).

9.6.2 SAMPLES ARE INDEPENDENT (ENOUGH)

The connection between the bias ε and the required number of samples ε^{-2} relies on the independence of the samples, and it is not obvious that the samples we get are independent. Most cryptanalysis simply assumes that the

Table 9.12: The attack in Subsection 9.6.2 was carried out on 2^{20} different keys using various numbers of plaintext–ciphertext pairs and samples per pair. »TPR« for »true positive ratio« shows how frequently a key belonging to the class was correctly identified. Similarly, »TNR« for »true negative ratio« shows how often a key not belonging to the class was correctly excluded.

Trail	Pairs	Samples/pair	TPR	TNR
$c_{10}^{8} = p_{10} + 1$	2^{18}	1	.83	.83
$c_{30}^{8} = p_{30} + 1$	2^{18}	1	.83	.86
$c_{35}^{8} = p_{35} + 1$	2^{18}	1	.83	.85
$c_{43}^{8} = p_{43} + 1$	2^{18}	1	.82	.84
All four	2^{16}	2^{2}	.83	.86
All four	2^{18}	2^{2}	.96	.98

samples are independent, or at least independent enough for the attacks to still be possible. Verifying the independence through simulation is common, at least on a smaller number of rounds or reduced-size versions of the algorithm (»PRINTcipher-12,« cf. Chapter 10), where it is practically possible.

We need to be a little bit more wary than usual as we obtain several samples from the same plaintext–ciphertext pair—the calculations behind the four samples have affected each other, and it is not impossible that samples obtained from the same plaintext–ciphertext pair are so dependent that they do not contribute (much) more than one sample would. If this is the case, we would not be able to exploit any bias smaller than (about) 2^{-23}.

Thus, we have performed the following on eight-round PRINTcipher. We use 2^{18} plaintext–ciphertext pairs to derive 2^{18} samples on bit $(14, 1)$, and from this we guess whether the key is in the upper-left class from Table 9.11 or not by comparing the number of samples that are 1 to a threshold $\theta = 2^{18}\left(\frac{1}{2} + \frac{\varepsilon}{2}\right)$, where $\varepsilon = 2^{-9}$. This gives false positives/negatives with certain probabilities over 2^{20} different keys, see Table 9.12. Similar probabilities are observed for the three mirroring trails, when used one on one.

If we instead use only 2^{16} plaintext–ciphertext pairs, but obtain 2^{2} samples from each pair, we are able to carry out the attack with seemingly unchanged success. The probabilities of false positives/negatives are 0.02/0.50.

On the other hand, using 2^{18} plaintext–ciphertext pairs, but 2^{2} samples per pair, the distinguisher achieves much better rates of false positives/negatives.

9.6.3 PARTIAL ENCRYPTION AND DECRYPTION FOR 29 ROUNDS

As in Section 9.5, we aim to guess key bits for partial encryptions and decryptions. Previously, we were able to add five rounds in this way to construct a 28-round attack using a 23-round trail. Now, using the 24-round trails, we reach 29 rounds. Again, we use the upper-left key class in Table 9.11.

The key observation is that we can divide all of the work, so that we deal with the four trails somewhat independently. Numbering the trails, in any order, as $j = 0, 1, 2, 3$, we will have time requirements

$$\begin{aligned} T_j^{\text{collect}} &= \varepsilon^{-2}, \\ T_j^{\text{count}} &= \frac{2^{a_{\text{e}}{}^j} N_{\text{e}}{}^j A_{\text{e}}^j + 2^{a_{\text{d}}{}^j} N_{\text{d}}{}^j A_{\text{d}}^j}{16 \cdot R}, \\ T_j^{\text{combine}} &= 2^{a_{\text{e}}{}^j + a_{\text{d}}{}^j} N, \end{aligned}$$

for producing the four different lists of counters S_i^j. In order to combine all counters S_i^j into N counters S_i we need to perform N rather simple operations. Note that we have made related, but not identical, guesses on the permutations, e.g., by guessing $\pi_1(2)$ and $\pi_1(0)$ for different trails. Some care must be taken here, but it does not affect the cost of this step, which remains at $T^{\text{finalize}} = N$ quite simple operations.

An overview is given in Figure 9.6 and specific guesses are listed in Table 9.13. For all j, we have $(a_{\text{e}}{}^j, a_{\text{d}}{}^j, A_{\text{e}}^j, A_{\text{d}}^j) = (9, 27, 4, 13)$. The total attack requirements are

$$\begin{aligned} T^{\text{collect}} &= 2^{2 \cdot 24} = 2^{48}, \\ T^{\text{count}} &= \sum_j T_j^{\text{count}} \approx 2^{50}, \\ T^{\text{combine}} &= \sum_j T_j^{\text{combine}} \approx 2^{76}, \\ T^{\text{finalize}} &= N = 2^{62} \cdot 3^3 \approx 2^{67}. \end{aligned}$$

Although brute force costs 2^{75}, as we assume five bits of the key, we claim that 2^{76} simple operations compare favorably to 2^{75} evaluations of 29-round PRINTCIPHER.

Note that several of the 2^{50} samples are completely unaffected by certain guessed key bits and trits. While this allowed for the division of work that kept the attack time below brute force, it also makes it less clear whether this attack is successful. However, *some* of the key material guessed for the various trails is related, so one might intuitively expect the attack to give *some* advan-

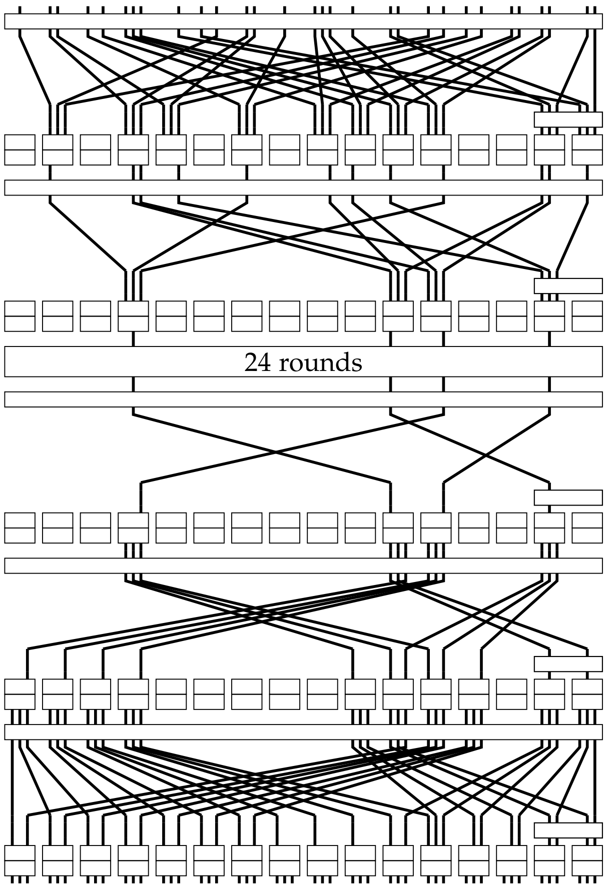


Figure 9.6: Performing two/three rounds of partial encryption/decryption to access the bits at positions $(3,1)$, $(10,0)$, $(11,2)$, and $(14,1)$.

tage over brute force. It might be possible, and favorable, to average the results, e.g., by performing majority voting using several of the highest ranked key guesses.

Let us briefly comment on the possibility of using 25-round trails with bias 2^{-26}: if we can get 16 samples per plaintext–ciphertext pair, we have the necessary 2^{52} samples. As we need to involve all π_b, we would put restrictions on at least 16 bits of the key. This puts the brute force cost at 2^{64} or lower, which seems to be too low for the attack to be meaningful. Another obstacle for this attack is that the mirroring trails are not completely identical, as different bits of $\boldsymbol{k}^+$ will appear.

Table 9.13: The key material guessed in the attack on 29-round PRINTCIPHER by encrypting/decrypting two/three rounds. Sequences of xor key bits are given using »–.«

Pos.	N^j		Key Material
(3, 1)	$2^{31} \cdot 3^6$	Enc.	$k^+_{38}, k^+_{35}, k^+_{33}, k^+_{27}, k^+_{22}, k^+_{19}, k^+_{17}, k^+_{11}, \pi_6, \pi^3_1$
		Dec.	k^+_{47}–$k^+_{44}, k^+_{42}, k^+_{32}$–$k^+_{30}, k^+_{10}, k^+_2$–$k^+_0$, $\pi_{15}, \pi_{13}(2), \pi_{12}(2), \pi^3_4, \pi_2(0), \pi_1(0), \pi_0$
		Both	$k^+_{43}, k^+_6, k^+_3, k^+_1$
(10, 0)	$2^{23} \cdot 3^9$	Enc.	$k^+_{46}, k^+_{40}, k^+_{26}, k^+_{24}, k^+_{19}, k^+_{14}, k^+_{10}, k^+_8, k^+_3, k^\pi_{29}, \pi^3_8$
		Dec.	$k^+_{44}, k^+_{43}, k^+_{38}$–$k^+_{36}, k^+_{34}$–$k^+_{31}, \pi_{15}(1), \pi_{12}(2)$, $k^\pi_{20}, \pi_6(2), \pi_5(2), \pi_4(2), \pi_2(2), \pi_1(2), \pi_0(2)$
		Both	$k^+_{42}, k^+_{35}, k^+_{30}$
(11, 2)	$2^{21} \cdot 3^6$	Enc.	$k^+_{46}, k^+_{43}, k^+_{41}, k^+_{25}, k^+_{19}, k^+_{14}, k^+_3, k^\pi_{29}$
		Dec.	k^+_{34}–$k^+_{31}, k^+_{29}, k^+_{28}, k^+_{10}, \pi_{15}(1), \pi_{13}(1), \pi_{12}(1)$, $k^\pi_{23}, \pi_2(2), \pi_1(2), \pi_0(2)$
		Both	$k^+_{35}, k^+_{30}, k^+_{27}, k^+_{11}, k^+_9, \pi^3_9 \Leftrightarrow \pi_9(0)$
(14, 1)	$2^{29} \cdot 3^7$	Enc.	$k^+_{47}, k^+_{46}, k^+_{42}, k^+_{36}, k^+_{31}, k^+_{30}, k^+_{26}, k^+_{20}, k^+_{15}, k^+_{14}$, $\pi_{15}, k^\pi_{29}, \pi^3_4$
		Dec.	k^+_{43}, k^+_{35}–$k^+_{33}, k^+_{11}, k^+_9$–$k^+_5, k^+_3, k^\pi_{23}, \pi_9(0), \pi_8(0)$, $\pi_7(0), \pi_6(0), \pi_5(0), \pi_4(0), \pi_2(0), \pi_1(0)$
		Both	k^+_{10}, k^+_4, π_4
All	$2^{62} \cdot 3^3$		k^+_{47}–k^+_{40}, k^+_{38}–$k^+_{24}, k^+_{22}, k^+_{20}, k^+_{19}, k^+_{17}, k^+_{15}, k^+_{14}$, k^+_{11}–k^+_0, $\pi_{15}, k^\pi_{29}, \pi_{13}, \pi_{12}, k^\pi_{23}, k^\pi_{20}, \pi^3_9 \Leftrightarrow \pi_9(0)$, $\pi^3_8 \Leftrightarrow \pi_8(0), \pi_7(0), \pi_6, \pi_5, \pi_4, \pi_2, \pi_1, \pi_0$

9.7 CONCLUSION

Table 9.14 summarizes the attacks on 27–29 rounds of PRINTCIPHER outlined in this chapter. Additional attacks are available for several more key classes.

We note some particular observations that all arise from the structure of PRINTCIPHER and the use of the exact same round key throughout the cipher:

- When there is a nondecomposable, iterated r'-round trail there are in fact r' mirroring trails, allowing r' samples per plaintext–ciphertext pair.
- When we guess for a partial encryption/decryption, there is overlap between the bits that activate the trail and those we need for encryption/decryption.

Table 9.14: A summary of the explicit attacks on 27-, 28- and 29-round PRINTCIPHER presented in this chapter. The length of the trail(s) used is denoted r', and R indicates that R-round PRINTCIPHER is attacked. Two rounds are partially encrypted and two or three are partially decrypted. T^{count} and T^{combine} are rounded to the nearest integer power of two.

Trail	r'	R	Key fraction	T^{collect}	T^{count}	T^{combine}
$c_0^{25} = c_0^2 + k_0^+$	23	27	2^{-1}	2^{48}	2^{15}	2^{36}
$c_0^{25} = c_0^2 + k_0^+$	23	28	2^{-1}	2^{48}	2^{44}	2^{67}
$c_{35}^{25} = c_{10}^2 + k_{35}^+ + 1$	23	28	2^{-5}	2^{48}	2^{46}	2^{73}
$c_{43}^{26} = c_{43}^2$ and more	24	29	2^{-5}	2^{48}	2^{50}	2^{76}

We have presented linear cryptanalysis on 29 rounds of PRINTCIPHER. We have used weak key classes which means that we need to carry out several attacks in parallel in order to have a high probability of success. However, we have seen that there are many large key classes and in particular, a number of them only depend on one or two bits of the key. This means our results in a sense invalidate more keys than previous results on PRINTCIPHER. The exception is the differential attack which worked on all keys but only reached 23 rounds.

We have exclusively studied PRINTCIPHER-48, but our observations are without doubt applicable to PRINTCIPHER-96 as well, where it seems reasonable that our techniques could be used to reach around 52–55 rounds.

As a direction for future research, we note that by inverting the $(S \circ \pi_b)$ of the last round where π_b is (partly) assumed, the number of active bits could be reduced. The technique would apply to all attacks in this chapter, but the full gain of this remains to be determined.

The mirroring trails that arise in PRINTCIPHER are very interesting, and allowed us to add one round to the attacks, albeit for a smaller class of keys. It would be very interesting to see if this mirroring property could lead to more observations on PRINTCIPHER.

10 Invariant Subspaces and Linear Correlations

Leander et al. [LAAZ11] have shown that by fixing certain key bits, PRINTcipher acquires a peculiar property. Denote by F_r the one-round PRINTcipher mapping of round r for a fixed key, and let F denote the full PRINTcipher mapping, i.e., $F = F_{47} \circ F_{46} \circ \ldots \circ F_0$. Partially fix the permutation key and xor key as in Figure 10.1. Then the following holds for the progression of the state over one round F_r of PRINTcipher, regardless of the value of the round constant:

Start	00* *10 *** *** 00* *10 *** *** 00* *10 *** *** 00* *10 *** ***
Key xoring	01* *01 *** *** 01* *01 *** *** 01* *01 *** *** 01* *01 *** ***
Lin. layer	00* 11* *** *** 0*0 1*1 *** *** *00 *11 *** *** 00* 11* *** ***
RC	00* 11* *** *** 0*0 1*1 *** *** *00 *11 *** *** 00* 11* *** ***
Perm. layer	00* *11 *** *** 00* *11 *** *** 00* *11 *** *** 00* *11 *** ***
Sbox layer	00* *10 *** *** 00* *10 *** *** 00* *10 *** *** 00* *10 *** ***

That is, with

$$V = \{(\texttt{00* *10 *** *** 00* *10 *** *** 00* *10 *** *** 00* *10 *** ***})\},$$

$F_r(\boldsymbol{v}) \in V$ for all $\boldsymbol{v} \in V$. Since the round function is a permutation, $F_r(V) = V$. (The elements of V are 48-bit vectors: commas have been removed to save space, $*$ denotes any bit value, and spaces have been used to group bits in triplets to improve readability.)

By writing the subspace U, its orthogonal subspace $U^\perp$ and the constant $\boldsymbol{d}$,

$$U = \{(\texttt{00* *00 *** *** 00* *00 *** *** 00* *00 *** *** 00* *00 *** ***})\},$$
$$U^\perp = \{(\texttt{**0 0** 000 000 **0 0** 000 000 **0 0** 000 000 **0 0** 000 000})\},$$
$$\boldsymbol{d} = (\texttt{000 010 000 000 000 010 000 000 000 010 000 000 000 010 000 000}),$$

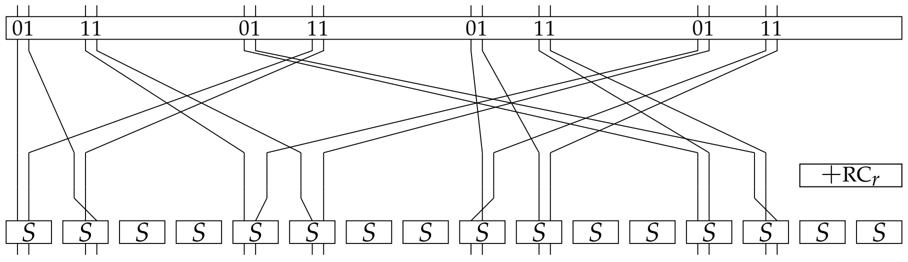


Figure 10.1: One round of PRINTcipher-48 with partly fixed permutation key and xor key. Only the bits that matter for the invariant subspace property are shown.

the invariant subspace[1] property can be expressed as

$$F_r(U + \boldsymbol{d}) = U + \boldsymbol{d}.$$

Since this property is not affected by the round constants, the full cipher F has the same property, regardless of the number of rounds,

$$F(U + \boldsymbol{d}) = U + \boldsymbol{d}.$$

Leander et al. showed that for a fraction 2^{-28} of all keys, PRINTcipher has such an invariant subspace. These keys are called weak keys and can be grouped into four different classes, depending on which bits are fixed. The particular class of weak keys considered here consists of all keys such that

$$\begin{aligned} \boldsymbol{k}^{+} &\in \{(\texttt{01* *11 *** *** 01* *11 *** *** 01* *11 *** *** 01* *11 *** ***})\}, \\ \boldsymbol{k}^{\pi} &\in \{(\texttt{0* 11 ** ** 10 01 ** ** 11 *0 ** ** *0 11 ** **})\}. \end{aligned} \tag{10.1}$$

The immediate consequence of this property is a distinguisher: encrypt 5 distinct plaintexts $\boldsymbol{p}^i \in U + \boldsymbol{d}$, $i = 0, 1, \ldots, 4$. If $F(\boldsymbol{p}^i) \in U + \boldsymbol{d}$, $i = 0, 1, \ldots, 4$, then output »key matches Equation 10.1,« otherwise output »key does not match Equation 10.1.« This distinguisher produces no false negatives, but false positives with probability approximately $(2^{-16})^5 = 2^{-80}$, assuming a uniform distribution of $F(\boldsymbol{p}^i)$ for nonweak keys and independent behavior of $F(\boldsymbol{p}^i)$ and $F(\boldsymbol{p}^j)$ for $i \neq j$.

Further, there exist strongly biased linear relations for any number of rounds. More precisely, Leander et al. give the following results, where Proposition 10.1 is PRINTcipher-specific and follows from Proposition 10.2.

[1] $U + \boldsymbol{d}$ is a *coset* of the subspace U, but Leander et al. chose to dub this property as the invariant *subspace*, rather than *coset*, as it was expected that the arguably more correct term would be less understood by the community. [Lea12]

Proposition 10.1 (Corollary 2 in [LAAZ11]) In PRINTcipher, for a weak key and for any round $R \leq 48$ there exists at least one linear approximation for F with correlation at least $2^{-16} - 2^{-32}$.

Proposition 10.2 (Corollary 1 in [LAAZ11]) Consider a subspace U and a constant $\boldsymbol{d}$. If $F(U + \boldsymbol{d}) = U + \boldsymbol{d}$, there exists at least one linear approximation for F with correlation at least $\left|U^{\perp}\right|^{-1} - \left|U^{\perp}\right|^{-2}$.

In this chapter, we will try to say more about linear biases in the case of invariant subspaces. We will consider input and output masks $\boldsymbol{\alpha}, \boldsymbol{\beta} \in U^{\perp}$, i.e., in the case of PRINTcipher, linear approximations which involve only those plaintext and ciphertext bits that are fixed in the invariant subspace. As a preliminary outlook, note that

$$\mathbf{Pr}\left[\langle \boldsymbol{\alpha}, \boldsymbol{x} \rangle = \langle \boldsymbol{\beta}, F(\boldsymbol{x}) \rangle \mid \boldsymbol{x} \in U + \boldsymbol{d}\right] \in \{0, 1\},$$

and that if

$$\mathbf{Pr}\left[\langle \boldsymbol{\alpha}, \boldsymbol{x} \rangle = \langle \boldsymbol{\beta}, F(\boldsymbol{x}) \rangle \mid \boldsymbol{x} \notin U + \boldsymbol{d}\right] = \frac{1}{2},$$

then

$$\begin{aligned}
\mathbf{Pr}\left[\langle \boldsymbol{\alpha}, \boldsymbol{x} \rangle = \langle \boldsymbol{\beta}, F(\boldsymbol{x}) \rangle\right] =& \mathbf{Pr}\left[\langle \boldsymbol{\alpha}, \boldsymbol{x} \rangle = \langle \boldsymbol{\beta}, F(\boldsymbol{x}) \rangle \mid \boldsymbol{x} \in U + \boldsymbol{d}\right] \cdot \mathbf{Pr}\left[\boldsymbol{x} \in U + \boldsymbol{d}\right] \\
&+ \mathbf{Pr}\left[\langle \boldsymbol{\alpha}, \boldsymbol{x} \rangle = \langle \boldsymbol{\beta}, F(\boldsymbol{x}) \rangle \mid \boldsymbol{x} \notin U + \boldsymbol{d}\right] \cdot \mathbf{Pr}\left[\boldsymbol{x} \notin U + \boldsymbol{d}\right] \\
=& \left(\frac{1}{2} \pm \frac{1}{2}\right) \cdot \left|U^{\perp}\right|^{-1} + \frac{1}{2} \cdot \left(1 - \left|U^{\perp}\right|^{-1}\right) \\
=& \frac{\left|U^{\perp}\right|^{-1}}{2} \pm \frac{\left|U^{\perp}\right|^{-1}}{2} + \frac{1}{2} - \frac{\left|U^{\perp}\right|^{-1}}{2} \\
=& \frac{1}{2} \pm \frac{\left|U^{\perp}\right|^{-1}}{2}.
\end{aligned} \tag{10.2}$$

This can be loosely interpreted as »if F behaves well on all other plaintexts, then the correlation is approximately $\pm \left|U^{\perp}\right|^{-1}$ for all linear approximations with $\boldsymbol{\alpha}, \boldsymbol{\beta} \in U + \boldsymbol{d}$,« and should be compared to Proposition 10.2. This outlook gives a loose explanation for the large correlations based on the different behavior of different plaintexts.

This chapter will present work that goes beyond this simple analysis by looking into what happens in the linear hull, showing that a round function with an invariant subspace yields a submatrix B with a particular eigenvector with eigenvalue 1. This is crucial as this implies that this submatrix B, when taken to the Rth power, does not converge to the all-zero matrix. In particular, if this is the only eigenvector which is 1 in absolute terms, B^R converges to a nonzero constant (cf. Proposition 3.10). This means that with repeated use of the round function, or possibly variants of it, trails with all intermediate masks determined by the invariant subspace cluster significantly for any number of rounds. In particular, this follows from the invariant subspace property, not, e.g., the single-bit linear characteristics in the round function of PRINTcipher (cf. Chapter 9), making the analysis applicable to block ciphers that, e.g., have been designed according to the wide-trail strategy.

Interestingly, in a restricted sense to be discussed below, the reciprocal statement holds as well. That is, if the cipher does not exhibit invariant subspaces, then no submatrices (of a certain type) have eigenvectors (of a certain type) with eigenvalue 1. Thus by avoiding invariant subspaces, one also ensures that trail clustering for any number of rounds is less likely.

10.1 UNDERSTANDING THE LARGE CORRELATIONS

The following small example will be helpful to illustrate some details in this chapter.

Example 4.3 (Cont'd) Note that with F as on page 43,

$$F(\{(1,0,1),(1,1,0)\}) = \{(1,0,1),(1,1,0)\}.$$

That is, with

$$V = \{\boldsymbol{x} \in \mathbb{F}_2^3 : \ x_0 = 1, x_1 + x_2 = 1\} = \{(1,0,1),(1,1,0)\},$$

$F(V) = V$. To write V as a coset of a subspace of $\mathbb{F}_2^3$, consider

$$U = \{(0,0,0),(0,1,1)\}$$

and $\boldsymbol{d} = (1,0,1)$. Clearly, U is a subspace, and

$$U + \boldsymbol{d} = \{(1,0,1),(1,1,0)\} = V.$$

□

We will consider an R-round block cipher $F = F_{R-1} \circ F_{R-2} \circ \ldots \circ F_0$, constructed from round functions F_r. Observe that the round key is moved into the round function F_r.

Recall from Section 4.5 that the correlation matrix $C_r = (c_{F_r}(\boldsymbol{\alpha},\boldsymbol{\beta}))_{\boldsymbol{\alpha}\boldsymbol{\beta}}$ collects all correlation coefficients for the rth round function. For reasons that will soon be clear, we are interested in the submatrix $A_r = (a^r_{\boldsymbol{\alpha}\boldsymbol{\beta}})_{\boldsymbol{\alpha},\boldsymbol{\beta}\in U^\perp}$ constructed through $a^r_{\boldsymbol{\alpha}\boldsymbol{\beta}} = c_{F_r}(\boldsymbol{\alpha},\boldsymbol{\beta})$.

From Proposition 4.2, we know that $C_{R-1}C_{R-2}\ldots C_0$ is the correlation matrix of the entire cipher F. Similarly, the matrix $A_{R-1}A_{R-2}\ldots A_0$ describes the contribution to the linear hull from following trails with intermediate masks in $U^{\perp}$. More specifically, with

$$c_F^{\mathrm{i}}(\boldsymbol{\alpha},\boldsymbol{\beta}) = \sum_{\substack{\boldsymbol{\theta}\colon \boldsymbol{\theta}^0=\boldsymbol{\alpha},\boldsymbol{\theta}^R=\boldsymbol{\beta},\\ \boldsymbol{\theta}^r\in U^{\perp},\ \forall r}} (-1)^{\langle\boldsymbol{\theta},E(\boldsymbol{k})\rangle} C^{\mathbf{0}}_{\boldsymbol{\theta}},$$

and

$$c_F^{\mathrm{o}}(\boldsymbol{\alpha},\boldsymbol{\beta}) = \sum_{\substack{\boldsymbol{\theta}\colon \boldsymbol{\theta}^0=\boldsymbol{\alpha},\boldsymbol{\theta}^R=\boldsymbol{\beta},\\ \exists r\colon \boldsymbol{\theta}^r\notin U^{\perp}}} (-1)^{\langle\boldsymbol{\theta},E(\boldsymbol{k})\rangle} C^{\mathbf{0}}_{\boldsymbol{\theta}},$$

we can write Equation 4.7 as

$$c_F(\boldsymbol{\alpha},\boldsymbol{\beta}) = c_F^{\mathrm{i}}(\boldsymbol{\alpha},\boldsymbol{\beta}) + c_F^{\mathrm{o}}(\boldsymbol{\alpha},\boldsymbol{\beta}), \tag{10.3}$$

where $c_F^{\mathrm{i}}(\boldsymbol{\alpha},\boldsymbol{\beta})$ corresponds to element $(\boldsymbol{\alpha},\boldsymbol{\beta})$ of $A_{R-1}A_{R-2}\dots A_0$. That is, $c_F^{\mathrm{i}}(\boldsymbol{\alpha},\boldsymbol{\beta})$ is the contribution from all trails which stay »inside« $U^{\perp}$, while $c_F^{\mathrm{o}}(\boldsymbol{\alpha},\boldsymbol{\beta})$ is the contribution from all other trails, i.e., those that go »outside« for at least one round.

It is important to note that if all A_r are equal, e.g., because any round constants do not affect the particular elements of C_r extracted as A_r, then we can write $A_r = A$, $\forall r$, and $A_{R-1}A_{R-2}\dots A_0 = A^R$. Then something more can be said from Equation 10.3: if elements of A^R have a »large« magnitude, then the corresponding elements of C^R are approximately the same, unless the contributions from trails that go outside $U^{\perp}$ are also »large« and, e.g., cancel the contributions that go on the inside. For this reason, it is interesting to consider the asymptotic behavior of A^R. Note that by Proposition 3.7, we already know that the eigenvalues of A are on the unit disk.

Lemma 10.3 Consider an invertible vectorial Boolean function F, a subspace U, the orthogonal subspace $U^{\perp}$ and a vector $\boldsymbol{d}$. Define $A = (a_{\boldsymbol{\alpha}\boldsymbol{\beta}})_{\boldsymbol{\alpha},\boldsymbol{\beta}\in U^{\perp}}$, $a_{\boldsymbol{\alpha}\boldsymbol{\beta}} = c_F(\boldsymbol{\alpha},\boldsymbol{\beta})$ and $\boldsymbol{v} = (v_{\boldsymbol{\alpha}})_{\boldsymbol{\alpha}\in U^{\perp}}$, $v_{\boldsymbol{\alpha}} = (-1)^{\langle\boldsymbol{d},\boldsymbol{\alpha}\rangle}$. If $F(U+\boldsymbol{d}) = U+\boldsymbol{d}$, then $\boldsymbol{v}A = \boldsymbol{v}$.

Example 4.3 (Cont'd) On page 43, the correlation matrix for F was found to be

$$2^{-3}\begin{pmatrix} 8 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\ 0 & 0 & -4 & -4 & 0 & 0 & 4 & -4\\ 0 & 4 & -4 & 0 & 4 & 0 & 0 & 4\\ 0 & 4 & 0 & -4 & -4 & 0 & -4 & 0\\ 0 & 0 & 4 & -4 & 4 & 4 & 0 & 0\\ 0 & 0 & 0 & 0 & 4 & -4 & -4 & -4\\ 0 & 4 & 0 & 4 & 0 & 4 & 0 & -4\\ 0 & -4 & -4 & 0 & 0 & 4 & -4 & 0 \end{pmatrix}, \tag{10.4}$$

where rows and columns corresponding to

$$\boldsymbol{\alpha}, \boldsymbol{\beta} \in U^{\perp} = \{(0,0,0), (1,0,0), (0,1,1), (1,1,1)\}$$

have been grayed. Consider the submatrix

$$A = \frac{1}{2} \begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 \\ 0 & 1 & 0 & -1 \\ 0 & -1 & -1 & 0 \end{pmatrix},$$

and the vector

$$\begin{aligned} \boldsymbol{v} &= \left((-1)^{\langle \boldsymbol{d}, (0,0,0)\rangle}, (-1)^{\langle \boldsymbol{d}, (1,0,0)\rangle}, (-1)^{\langle \boldsymbol{d}, (0,1,1)\rangle}, (-1)^{\langle \boldsymbol{d}, (1,1,1)\rangle}\right) \\ &= (1, -1, -1, 1). \end{aligned}$$

Indeed,

$$(1, -1, -1, 1) \, \frac{1}{2} \begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 \\ 0 & 1 & 0 & -1 \\ 0 & -1 & -1 & 0 \end{pmatrix} = (1, -1, -1, 1).$$

□

Proof (Lemma 10.3). We want to show that for every $\boldsymbol{\alpha} \in U^{\perp}$, we have

$$(\boldsymbol{v}A)_{\boldsymbol{\alpha}} = v_{\boldsymbol{\alpha}}.$$

With scaling, we have

$$\begin{aligned} 2^n (\boldsymbol{v}A)_{\boldsymbol{\alpha}} &= 2^n \sum_{\boldsymbol{\beta} \in U^{\perp}} v_{\boldsymbol{\beta}} a_{\boldsymbol{\alpha}\boldsymbol{\beta}} = 2^n \sum_{\boldsymbol{\beta} \in U^{\perp}} v_{\boldsymbol{\beta}} 2^{-n} \widehat{F}(\boldsymbol{\alpha}, \boldsymbol{\beta}) \\ &= \sum_{\boldsymbol{\beta} \in U^{\perp}} (-1)^{\langle \boldsymbol{d}, \boldsymbol{\beta}\rangle} \sum_{\boldsymbol{x}} (-1)^{\langle \boldsymbol{\beta}, F(\boldsymbol{x})\rangle + \langle \boldsymbol{\alpha}, \boldsymbol{x}\rangle} \\ &= \sum_{\boldsymbol{x}} (-1)^{\langle \boldsymbol{\alpha}, \boldsymbol{x}\rangle} \sum_{\boldsymbol{\beta} \in U^{\perp}} (-1)^{\langle \boldsymbol{\beta}, F(\boldsymbol{x}) + \boldsymbol{d}\rangle}. \end{aligned}$$

Similar to Proposition 3.1,

$$\sum_{\boldsymbol{y} \in U^{\perp}} (-1)^{\langle \boldsymbol{a}, \boldsymbol{y}\rangle} = \begin{cases} |U^{\perp}|, & \boldsymbol{a} \in U, \\ 0, & \boldsymbol{a} \notin U. \end{cases}$$

Applying this to the above and using the invariant subspace property, we get

$$\begin{aligned}
2^n (\boldsymbol{v}A)_{\boldsymbol{\alpha}} &= \sum_{\boldsymbol{x}:\, F(\boldsymbol{x}) \in U+\boldsymbol{d}} (-1)^{\langle \boldsymbol{\alpha}, \boldsymbol{x} \rangle} \left| U^{\perp} \right| \\
&= \sum_{\boldsymbol{x} \in U+\boldsymbol{d}} (-1)^{\langle \boldsymbol{\alpha}, \boldsymbol{x} \rangle} \left| U^{\perp} \right| \\
&= \sum_{\boldsymbol{x} \in U} (-1)^{\langle \boldsymbol{\alpha}, \boldsymbol{x}+\boldsymbol{d} \rangle} \left| U^{\perp} \right| \\
&= (-1)^{\langle \boldsymbol{\alpha}, \boldsymbol{d} \rangle} \sum_{\boldsymbol{x} \in U} (-1)^{\langle \boldsymbol{\alpha}, \boldsymbol{x} \rangle} \left| U^{\perp} \right| .
\end{aligned}$$

Since $\boldsymbol{x} \in U$ and $\boldsymbol{\alpha} \in U^{\perp}$ it holds that $\langle \boldsymbol{\alpha}, \boldsymbol{x} \rangle = 0$, so

$$2^n (\boldsymbol{v}A)_{\boldsymbol{\alpha}} = \left| U^{\perp} \right| |U| \, (-1)^{\langle \boldsymbol{\alpha}, \boldsymbol{d} \rangle} = 2^n v_{\boldsymbol{\alpha}}.$$

Thus, $\boldsymbol{v}$ is an eigenvector of A with eigenvalue 1. ■

In any correlation matrix representing a permutation, the first row and column are all-zero except for $c(0,0) = 1$. Thus, the submatrix $B = (a_{\boldsymbol{\alpha}\boldsymbol{\beta}})_{\boldsymbol{\alpha},\boldsymbol{\beta} \in U_*^{\perp}}$ can be extracted without losing any information, where

$$U_*^{\perp} = U^{\perp} \backslash \{\mathbf{0}\}.$$

This is preferable as $U_*^{\perp}$ will be slightly more convenient to work with. Similarly, from $\boldsymbol{v}$, extract $\boldsymbol{u} = (v_{\boldsymbol{\alpha}})_{\boldsymbol{\alpha} \in U_*^{\perp}}$. From the block-diagonal structure of A with $c_F(\mathbf{0},\mathbf{0}) = 1$, it follows that $\boldsymbol{v}A = \boldsymbol{v}$ if and only if $\boldsymbol{u}B = \boldsymbol{u}$. It is also clear that $(\mathbf{0}||\boldsymbol{u})A = \mathbf{0}||\boldsymbol{u}$, and $(1,0,0,\ldots,0)A = (1,0,0,\ldots,0)$.

Now, in the case where there are only those two (linearly independent) eigenvectors to A with eigenvalue 1, the sequence A^R will converge (cf. Proposition 3.10). This motivates the following definition.

Definition 10.1
If the algebraic multiplicity of the eigenvalue 1 of A is two and A has no other eigenvalue of absolute value 1, we say that A (or the corresponding cipher) has a *stable symmetry*.

Theorem 10.4 If A has a stable symmetry then

$$B^R \to D = \frac{1}{|U^{\perp}| - 1} \boldsymbol{u}^{\mathrm{T}} \boldsymbol{u}, \quad R \to \infty,$$

and

$$A^R = \begin{pmatrix} 1 & \mathbf{0} \\ \mathbf{0} & B^R \end{pmatrix} \to \begin{pmatrix} 1 & \mathbf{0} \\ \mathbf{0} & D \end{pmatrix}, \quad R \to \infty.$$

Example 4.3 (Cont'd) The four eigenvectors of A can be chosen as

$$\begin{aligned} \boldsymbol{v}_0 &= (1,0,0,0)\,, \\ \boldsymbol{v}_1 &= (0,-1,-1,1)\,, \\ \boldsymbol{v}_2 &= (0,1,0,1)\,, \\ \boldsymbol{v}_3 &= (0,0,1,1)\,, \end{aligned}$$

with corresponding eigenvalues $\lambda_0 = \lambda_1 = 1$ and $\lambda_2 = \lambda_3 = -\frac{1}{2}$, so A has a stable symmetry. With $\boldsymbol{u} = (1,1,-1)$,

$$\boldsymbol{u}^{\mathrm{T}}\boldsymbol{u} = \begin{pmatrix} 1 & 1 & -1 \\ 1 & 1 & -1 \\ -1 & -1 & 1 \end{pmatrix}.$$

Further,

$$B = \frac{1}{2}\begin{pmatrix} 0 & 1 & -1 \\ 1 & 0 & -1 \\ -1 & -1 & 0 \end{pmatrix} = \frac{1}{2^1 \cdot 3}\begin{pmatrix} 0 & 3 & -3 \\ 3 & 0 & -3 \\ -3 & -3 & 0 \end{pmatrix},$$

and

$$B^2 = \frac{1}{4}\begin{pmatrix} 2 & 1 & -1 \\ 1 & 2 & -1 \\ -1 & -1 & 2 \end{pmatrix} = \frac{1}{2^2 \cdot 3}\begin{pmatrix} 6 & 3 & -3 \\ 3 & 6 & -3 \\ -3 & -3 & 6 \end{pmatrix}.$$

In fact, it is possible to show that

$$B^R = \frac{1}{2^R \cdot 3}\begin{pmatrix} d_R & e_R & -e_R \\ e_R & d_R & -e_R \\ -e_R & -e_R & d_R \end{pmatrix}, \quad R \geq 0,$$

with

$$d_R = 2^R + 2(-1)^{R \bmod 2},$$

and

$$e_R = 2^R - (-1)^{R \bmod 2}.$$

From this follows that

$$B^R \to \frac{1}{3}\begin{pmatrix} 1 & 1 & -1 \\ 1 & 1 & -1 \\ -1 & -1 & 1 \end{pmatrix}, \quad R \to \infty,$$

as predicted by Theorem 10.4. Thus,

$$A^R \to \frac{1}{3}\begin{pmatrix} 3 & 0 & 0 & 0 \\ 0 & 1 & 1 & -1 \\ 0 & 1 & 1 & -1 \\ 0 & -1 & -1 & 1 \end{pmatrix}, \quad R \to \infty.$$

□

Proof (Theorem 10.4). If A has a stable symmetry, then B has an eigenvector 1 with algebraic and geometric multiplicity 1, and there are no other eigenvalues with $|\lambda_i| = 1$. By Proposition 3.7, there are no eigenvalues with $|\lambda_i| > 1$. The eigenvector corresponding to the eigenvalue 1 is $\boldsymbol{u}$, and since

$$\boldsymbol{u}\boldsymbol{u}^{\mathrm{T}} = \sum_{\boldsymbol{\alpha}\in U_*^{\perp}} (-1)^{\langle \boldsymbol{d},\boldsymbol{\alpha}\rangle}(-1)^{\langle \boldsymbol{d},\boldsymbol{\alpha}\rangle} = \sum_{\boldsymbol{\alpha}\in U_*^{\perp}} 1 = \left|U_*^{\perp}\right| = \left|U^{\perp}\right| - 1,$$

it can be normalized as $l\boldsymbol{u}$, where $l = (\left|U^{\perp}\right| - 1)^{-1/2}$. By Proposition 3.10, $B^R \to (l\boldsymbol{u}^{\mathrm{T}})(l\boldsymbol{u}) = l^2\boldsymbol{u}^{\mathrm{T}}\boldsymbol{u}$, $R \to \infty$. ■

From Theorem 10.4 and the structure of $\boldsymbol{u}^{\mathrm{T}}\boldsymbol{u}$ follows, that for a cipher with a stable symmetry, with $\boldsymbol{\alpha}, \boldsymbol{\beta} \in U_*^{\perp}$, and for large enough values of R, $c_F^{\mathrm{i}}(\boldsymbol{\alpha},\boldsymbol{\beta}) \approx \pm\left|U^{\perp}\right|^{-1}$. Thus, if $c_F^{\mathrm{o}}(\boldsymbol{\alpha},\boldsymbol{\beta})$ is negligible, the correlation is $c_F(\boldsymbol{\alpha},\boldsymbol{\beta}) \approx \pm\left|U^{\perp}\right|^{-1}$ and the bias is $\varepsilon_F(\boldsymbol{\alpha},\boldsymbol{\beta}) \approx \pm\frac{\left|U^{\perp}\right|^{-1}}{2}$ (cf. Equations 10.2 and 10.3). In particular, if $\left|U^{\perp}\right|$ is »small,« e.g., $\left|U^{\perp}\right| < 2^{n/2}$, then the correlation is »large,« e.g., $|c_F(\boldsymbol{\alpha},\boldsymbol{\beta})| \gtrapprox 2^{-n/2}$.

Observe that for every permutation ϕ, there exists a $j > 0$ such that

$$\phi^j = \underbrace{\phi \circ \phi \circ \ldots \circ \phi}_{j \text{ times } \phi} \equiv \mathrm{id},$$

where id is the identity function which maps every $\boldsymbol{x}$ to $\boldsymbol{x}$ itself. By Proposition 4.1, the correlation matrix of the identify function is the identity matrix, so $C_{\phi^j} = C_\phi^j = \mathrm{I}$. Thus, even if there is a stable symmetry so that $c_{\phi^j}^{\mathrm{i}}(\boldsymbol{\alpha},\boldsymbol{\beta})$ are known asymptotically by Theorem 10.4, the correlation coefficients $c_{\phi^j}(\boldsymbol{\alpha},\boldsymbol{\beta})$ have very little similarity with $c_{\phi^j}^{\mathrm{i}}(\boldsymbol{\alpha},\boldsymbol{\beta})$, and thus the values $c_{\phi^j}^{\mathrm{o}}(\boldsymbol{\alpha},\boldsymbol{\beta})$ are not negligible. This shows that the asymptotic behavior of A^R does not *guarantee* large correlations, even with a stable symmetry.

Example 4.3 (Cont'd) $F \circ F \circ F \circ F \circ F \circ F = \mathrm{id}$, so $C^6 = \mathrm{I}$. From above,

$$A^6 = \frac{1}{64\cdot 3}\begin{pmatrix} 64\cdot 3 & 0 & 0 & 0 \\ 0 & 66 & 63 & -63 \\ 0 & 63 & 66 & -63 \\ 0 & -63 & -63 & 66 \end{pmatrix} \approx \frac{1}{3}\begin{pmatrix} 3 & 0 & 0 & 0 \\ 0 & 1 & 1 & -1 \\ 0 & 1 & 1 & -1 \\ 0 & -1 & -1 & 1 \end{pmatrix},$$

but

$$(C^6)_{\boldsymbol{\alpha},\boldsymbol{\beta}\in U^{\perp}} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

so for $R = 6$, the structure from A^R is not apparent in (a submatrix of) C^R. □

10.2 EQUIVALENCE BETWEEN EIGENVECTORS AND INVARIANT SUBSPACES

It is possible to strengthen Lemma 10.3 into an equivalence.

Theorem 10.5 Consider an invertible vectorial Boolean function F, a subspace U, the orthogonal subspace $U^\perp$ and a vector $\boldsymbol{d}$. Define $A = (a_{\boldsymbol{\alpha}\boldsymbol{\beta}})_{\boldsymbol{\alpha},\boldsymbol{\beta}\in U^\perp}$, $a_{\boldsymbol{\alpha}\boldsymbol{\beta}} = c_F(\boldsymbol{\alpha},\boldsymbol{\beta})$, and $\boldsymbol{v} = (v_{\boldsymbol{\alpha}})_{\boldsymbol{\alpha}\in U^\perp}$, $v_{\boldsymbol{\alpha}} = (-1)^{\langle \boldsymbol{d},\boldsymbol{\alpha}\rangle}$. Then $\boldsymbol{v}A = \boldsymbol{v}$ if and only if $F(U+\boldsymbol{d}) = U+\boldsymbol{d}$.

The »if« part follows from Lemma 10.3, so it only remains to prove the »only if« part. For this, we need the inverse transform,

$$(-1)^{\langle \boldsymbol{\beta},F(\boldsymbol{x})\rangle} = 2^{-n}\sum_{\boldsymbol{\alpha}}(-1)^{\langle \boldsymbol{\alpha},\boldsymbol{x}\rangle}\widehat{F}(\boldsymbol{\alpha},\boldsymbol{\beta}).$$

We first show a general lemma.

Lemma 10.6 Consider a Boolean function $f\colon \mathbb{F}_2^n \to \mathbb{F}_2$ and its Fourier transform $\widehat{f}$. Let U be a subspace of $\mathbb{F}_2^n$ and let $b \in \{0,1\}$. If, for $\boldsymbol{d} \in \mathbb{F}_2^n$,

$$2^{-n}\sum_{\boldsymbol{\alpha}\in U^\perp}(-1)^{\langle \boldsymbol{\omega},\boldsymbol{d}\rangle}\widehat{f}(\boldsymbol{\omega}) = (-1)^b,$$

or, equivalently,

$$\sum_{\boldsymbol{\omega}\in U^\perp}(-1)^{\langle \boldsymbol{\omega},\boldsymbol{d}\rangle}c_f(\boldsymbol{\omega}) = (-1)^b,$$

then $f(\boldsymbol{d}) = b$ and $f(\boldsymbol{x})$ is constant for $\boldsymbol{x} \in U+\boldsymbol{d}$, $f(U+\boldsymbol{d}) = \{b\}$.

Proof. By Proposition 3.5,

$$(-1)^b = \frac{|U^\perp|}{2^n}\sum_{\boldsymbol{x}\in U+\boldsymbol{d}}(-1)^{f(\boldsymbol{x})} = \frac{1}{|U|}\sum_{\boldsymbol{x}\in U+\boldsymbol{d}}(-1)^{f(\boldsymbol{x})}.$$

Since $(-1)^{f(\boldsymbol{x})} \in \{-1,+1\}$, $f(\boldsymbol{x})$ must be constant for $\boldsymbol{x} \in U+\boldsymbol{d}$. Further,

$$\sum_{\boldsymbol{x}\in U+\boldsymbol{d}}(-1)^{f(\boldsymbol{x})} = (-1)^{f(\boldsymbol{d})}\,|U|\,.$$

Thus, $(-1)^b = (-1)^{f(\boldsymbol{d})}$, so $f(\boldsymbol{d}) = b$. ■

What this lemma says is that if the calculation of $f(\boldsymbol{x})$ from $\{\widehat{f}(\boldsymbol{\alpha})\}_{\boldsymbol{\alpha}\in\mathbb{F}_2^n}$ appears to be possible using only a subset $\{\widehat{f}(\boldsymbol{\alpha})\}_{\boldsymbol{\alpha}\in U^\perp}$ of the Fourier coefficients, then it is (the subset is not just any subset but selected by a subspace).

Example 4.3 (Cont'd) For the component function $f(\boldsymbol{x}) = \langle (0,1,1), F(\boldsymbol{x}) \rangle$,

$$(f(0,0,0), f(1,0,0), \ldots, f(1,1,1)) = (0,1,0,1,0,1,1,0)$$

(cf. Table 4.1). By Equation 10.4, the Walsh spectrum is $(0,4,0,4,0,4,0,-4)$. Consider as before the subspace $U = \{(0,0,0),(0,1,1)\}$ and the orthogonal subspace $U^{\perp} = \{(0,0,0),(1,0,0),(0,1,1),(1,1,1)\}$. Note that $c_F(0,0,0) = 0$, $c_F(1,0,0) = \frac{1}{2}$, $c_F(0,1,1) = 0$, and $c_F(1,1,1) = -\frac{1}{2}$. For $\boldsymbol{x} = (1,0,1)$,

$$\begin{aligned}\sum_{\boldsymbol{\alpha} \in U^{\perp}} (-1)^{\langle \boldsymbol{\alpha}, \boldsymbol{x} \rangle} c_f(\boldsymbol{\alpha}) &= \left((-1)^0 \cdot 0 + (-1)^1 \cdot \frac{1}{2} + (-1)^1 \cdot 0 + (-1)^2 \cdot \left(-\frac{1}{2} \right) \right) \\ &= \left(-\frac{1}{2} - \frac{1}{2} \right) = (-1)^1,\end{aligned}$$

so $f(\boldsymbol{x}) = 1$ can be concluded from only those four Walsh coefficients. Further, $f(1,0,1) = f(1,1,0) = 1$, so, as predicted by Lemma 10.6, $f(\boldsymbol{y})$ is indeed constant for $\boldsymbol{y} \in U + \boldsymbol{x} = \{(1,0,1),(1,1,0)\}$.

On the other hand, for $\boldsymbol{x} = (0,1,1)$,

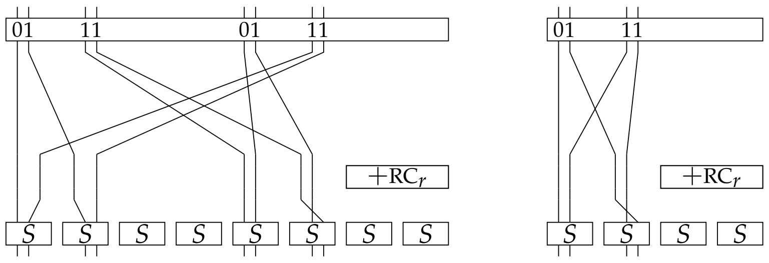
$$\begin{aligned}\sum_{\boldsymbol{\alpha} \in U^{\perp}} (-1)^{\langle \boldsymbol{\alpha}, \boldsymbol{x} \rangle} c_f(\boldsymbol{\alpha}) &= \left((-1)^0 \cdot 0 + (-1)^0 \cdot \frac{1}{2} + (-1)^0 \cdot 0 + (-1)^0 \cdot (-\frac{1}{2}) \right) \\ &= \left(\frac{1}{2} - \frac{1}{2} \right) = 0 \notin \{-1,1\},\end{aligned}$$

so the partial Walsh spectrum over $U^{\perp}$ is not sufficient to calculate $f(\boldsymbol{x})$. Further, $f(\boldsymbol{y})$ is not constant for $\boldsymbol{y} \in U + \boldsymbol{x} = \{(0,1,1),(0,0,0)\}$ as we have $f(0,1,1) \neq f(0,0,0)$. □

Proof (Theorem 10.5). As already noted, the »if« part follows from Lemma 10.3. Assume that $\boldsymbol{v}A = \boldsymbol{v}$. Consider $\boldsymbol{\alpha} \in U^{\perp}$ and write $\boldsymbol{a}_{\boldsymbol{\alpha}} = (c_F(\boldsymbol{\alpha}, \boldsymbol{\beta}))_{\boldsymbol{\beta} \in U^{\perp}}$ for the corresponding column of A. Note that for $\boldsymbol{x} \in U + \boldsymbol{d}$, we have $\langle \boldsymbol{\alpha}, \boldsymbol{x} \rangle = \langle \boldsymbol{\alpha}, \boldsymbol{d} \rangle$. Summing over $\boldsymbol{\beta} \in U^{\perp}$ yields

$$2^{-n} \sum_{\boldsymbol{\beta} \in U^{\perp}} (-1)^{\langle \boldsymbol{\beta}, \boldsymbol{d} \rangle} \widehat{F}(\boldsymbol{\alpha}, \boldsymbol{\beta}) = \sum_{\boldsymbol{\beta} \in U^{\perp}} v_{\boldsymbol{\beta}} c_F(\boldsymbol{\beta}, \boldsymbol{\alpha}) = \boldsymbol{v} \cdot \boldsymbol{a}_{\boldsymbol{\alpha}} = 1 \cdot v_{\boldsymbol{\alpha}} = (-1)^{\langle \boldsymbol{d}, \boldsymbol{\alpha} \rangle},$$

and by Lemma 10.6, we can conclude that $\langle \boldsymbol{\alpha}, F(\boldsymbol{x}) \rangle = \langle \boldsymbol{\alpha}, \boldsymbol{d} \rangle = \langle \boldsymbol{\alpha}, \boldsymbol{x} \rangle$ for all $\boldsymbol{x} \in U + \boldsymbol{d}$. Since this holds for all $\boldsymbol{\alpha} \in U^{\perp}$, $F(\boldsymbol{x}) \in U + \boldsymbol{d}$ for all $\boldsymbol{x} \in U + \boldsymbol{d}$. Since F is a permutation, $F(U + \boldsymbol{d}) = U + \boldsymbol{d}$. ■



(a) PRINTcipher-24. **(b)** PRINTcipher-12.

Figure 10.2: One round of PRINTcipher-24 resp. PRINTcipher-12 with partly fixed permutation key and xor key. Only the bits that matter for the invariant subspace property are shown.

10.3 EXPERIMENTAL RESULTS ON PRINTcipher

In this section, we investigate to what extent PRINTcipher behaves as expected from the above general results. Smaller-state versions of PRINTcipher are not formally defined but are easy to extrapolate from the specification of PRINTcipher, and are shown in Figure 10.2. We use the same round constants as in the first rounds of PRINTcipher-48. When using a fixed weak permutation key and sometimes also a fixed weak xor key, they are always chosen as the keys with the lowest Hamming weight. This should not be understood as the all-zero key, but the unique key in the class of weak keys where all variable key bits are chosen as zero.

10.3.1 THE BIAS DISTIBUTION OVER XOR KEYS

PRINTcipher-24 is shown in Figure 10.2a. We study the class of weak keys consisting of keys such that

$$\boldsymbol{k}^+ \in \{(\texttt{01* *11 *** *** 01* *11 *** ***})\},$$
$$\boldsymbol{k}^\pi \in \{(\texttt{10 01 ** ** 0* 11 ** **})\}.$$

Fix the permutation key and consider the characteristic from the leftmost plaintext bit to the leftmost ciphertext bit,

$$\boldsymbol{\alpha} = \boldsymbol{\beta} = (100\,000\,000\,000\,000\,000\,000\,000) \in U^\perp,$$

(cf. Chapter 9). As we run through all xor keys (weak and nonweak), we can derive the distribution of the correlation $c_F(\boldsymbol{\alpha}, \boldsymbol{\beta})$ of this characteristic. That is,

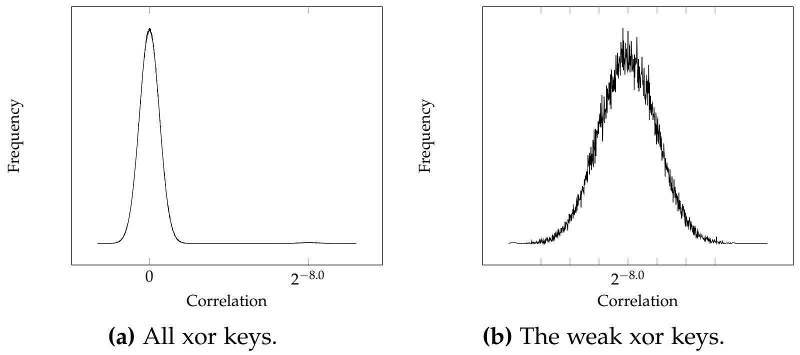


Figure 10.3: The distribution of PRINTcipher-24 biases for a fixed permutation key. In Figure 10.3b, the experimentally observed mean m is indicated. The standard deviation σ is approximately $2^{-12.0}$. Ticks have been placed at $m + j\sigma$, $j \in \{-3, \ldots, 3\}$.

for each xor key, we derive the correlation $c_F(\boldsymbol{\alpha}, \boldsymbol{\beta})$ explicitly by going through all 2^{24} plaintext–ciphertext pairs.

Figure 10.3a shows the correlation distribution. At first glance, it appears to consist of a single bell-shape nicely centered around 0, but there is also a small bump to the right. This is another bell-shape, representing precisely the weak keys, and in Figure 10.3b, we provide a magnified view of this part of the distribution. This experiment shows that all nonweak keys are distributed around approximately zero, while all weak keys are distributed around approximately 2^{-8}. From Chapter 9, we might expect 2^{-24} rather than 0, but the amount of data available is not enough to distinguish between these two values.

10.3.2 EXPERIMENTAL RESULTS ON PRINTcipher-48

We have implemented PRINTcipher-48 for a fixed key from the class of weak keys given by Equation 10.1. By exhaustively going through all plaintext–ciphertext pairs for all $\boldsymbol{\alpha}, \boldsymbol{\beta} \in U^{\perp}$, we could then derive A, which allowed us to verify that $\boldsymbol{v}A = \boldsymbol{v}$. We have also derived the correlations for 16 characteristics with $\boldsymbol{\alpha}, \boldsymbol{\beta} \in U^{\perp}$, see Table 10.1. We expect correlations

$$(-1)^{\langle \boldsymbol{d}, \boldsymbol{\alpha} \rangle} (-1)^{\langle \boldsymbol{d}, \boldsymbol{\beta} \rangle} \left(2^{16} - 1\right)^{-1} = (-1)^{\langle \boldsymbol{d}, \boldsymbol{\alpha} + \boldsymbol{\beta} \rangle} \left(2^{16} - 1\right)^{-1},$$

and all 16 correlations are indeed very close to $\pm 2^{-16}$ with the correct sign. Certainly, these are only 2^4 of almost 2^{32} correlation values, but this gives

Table 10.1: Input and output masks and the correlations experimentally observed over all plaintext–ciphertext pairs for a fixed weak key of PRINTCIPHER-48.

$\boldsymbol{\alpha}$	$\boldsymbol{\beta}$	Expected sign	$c_F(\boldsymbol{\alpha}, \boldsymbol{\beta})$
800000000000	800000000000	$+$	$+2^{-16.000}$
400000000000	000800000000	$+$	$+2^{-15.996}$
040000000000	000000000040	$+$	$+2^{-16.004}$
000080000000	000000000800	$-$	$-2^{-15.996}$
000440000000	000040080000	$-$	$-2^{-15.997}$
800040800000	080000000800	$-$	$-2^{-16.007}$
040000400000	000000000400	$+$	$+2^{-15.997}$
000000880000	400080000040	$+$	$+2^{-15.994}$
000080040000	040000400000	$-$	$-2^{-16.003}$
800000840000	000040880000	$-$	$-2^{-15.990}$
000000080800	800400000000	$-$	$-2^{-16.005}$
080800000400	000800000000	$-$	$-2^{-16.011}$
000000000080	000040040000	$-$	$-2^{-16.013}$
000840000080	400800800000	$-$	$-2^{-15.987}$
400000000040	0c0000800000	$-$	$-2^{-15.996}$
4c0480440c40	cc0480440c80	$-$	$-2^{-16.002}$

some circumstantial support to the idea that PRINTCIPHER-48 has a stable symmetry, that $B^{48} \approx D$, and that $c_F^{\mathrm{i}}(\boldsymbol{\alpha}, \boldsymbol{\beta})$ is the main contribution to $c_F(\boldsymbol{\alpha}, \boldsymbol{\beta})$.

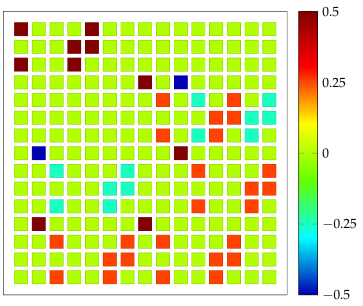
10.3.3 EXPERIMENTAL RESULTS ON PRINTCIPHER-12

On PRINTCIPHER-12, for a fixed key, we can derive B analogously to above. The class of weak keys considered consists of keys such that

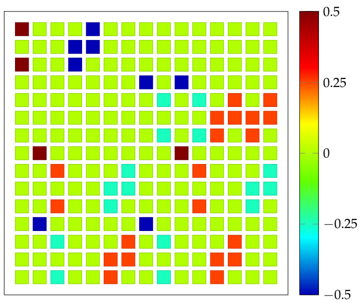
$$\boldsymbol{k}^{+} \in \{(\texttt{01* *11 *** ***})\},$$
$$\boldsymbol{k}^{\pi} \in \{(\texttt{0* 11 ** **})\}.$$

Here the stable symmetry can then be confirmed by deriving the eigenvalues numerically for all possible matrices B. Also, the convergence can be observed experimentally. Figure 10.4a shows B_{n} for a fixed nonweak key

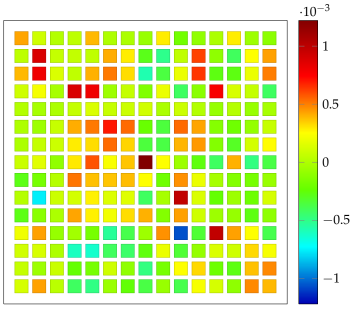
$$\boldsymbol{k}_{\mathrm{n}}^{+} = (000\ 000\ 000\ 000),$$
$$\boldsymbol{k}_{\mathrm{n}}^{\pi} = (00\ 11\ 00\ 00),$$



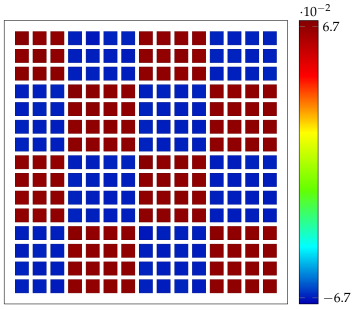
(a) B_n



(b) B_w



(c) B_n^{12}



(d) B_w^{12}

Figure 10.4: The matrices B and B^{12} for two different keys.

while Figure 10.4b shows B_w for a fixed weak key

$$\begin{aligned} \boldsymbol{k}_\text{w}^{+} &= (\texttt{010 011 000 000}), \\ \boldsymbol{k}_\text{w}^{\pi} &= (\texttt{00 11 00 00}). \end{aligned}$$

In particular, $\boldsymbol{k}_\text{w}^{\pi} = \boldsymbol{k}_\text{n}^{\pi}$, so the matrices B_n and B_w only differ by the signs of some nonzero elements. Figure 10.4c shows B_n^{12} and Figure 10.4d shows B_w^{12}. The matrices clearly differ both in terms of magnitude and structure.

10.4 A TECHNICAL, PRINTCIPHER-SPECIFIC PROOF OF LEMMA 10.3

We consider PRINTCIPHER with a weak key from the class given by Equation 10.1 and want to show the eigenvector property, $\boldsymbol{v}A = \boldsymbol{v}$. If one only wants to understand why this property arises in the general case, the proof given on page 160 is sufficient and simpler. However, in this section, we will

Table 10.2: LAT for the Sbox in PRINTcipher. All nontrivial nonzero values have the same magnitude.

$4 \cdot c_F(\boldsymbol{\alpha}, \boldsymbol{\beta})$		Input mask, $(\boldsymbol{\alpha})$							
		0	1	2	3	4	5	6	7
Output mask, $(\boldsymbol{\beta})$	0	4	0	0	0	0	0	0	0
	1	0	2	0	2	0	2	0	-2
	2	0	0	2	2	0	0	-2	2
	3	0	2	2	0	0	-2	2	0
	4	0	0	0	0	-2	2	2	2
	5	0	-2	0	2	2	0	2	0
	6	0	0	2	-2	2	2	0	0
	7	0	2	-2	0	2	0	0	2

take a careful look at what happens inside PRINTcipher, how the individual bits interact with each other and how everything adds up. We present things from the bottom up, first looking at individual bits and Sboxes, and then gradually moving to larger structures until we reach the entire matrix A. To simplify, we fix the permutation key as in Figure 10.1.

Table 10.2 presents the scaled correlation matrix of the PRINTcipher Sbox from Equation 9.1. This will be referred to as the linear approximation table (LAT). As all interesting nonzero elements are the same in absolute terms, we will from now on give elements simply as »+« or »-.« It will be clear below that this is possible without loss of information when we know how many these nonzero elements are, or their absolute value.

Throughout this section, $\boldsymbol{\alpha}, \boldsymbol{\beta} \in U^{\perp}$ will be assumed. Denote the ith Sbox as S_i, $0 \leq i < 16$, but include in it the partial permutation key and partial xor key. The elements of A depend only on the eight Sboxes indexed by

$$\mathcal{S} = \{0, 1, 4, 5, 8, 9, 12, 13\},$$

which can be partitioned into $\mathcal{S}_{\text{even}} = \{0, 4, 8, 12\}$ and $\mathcal{S}_{\text{odd}} = \{1, 5, 9, 13\}$. We further write $\mathcal{S}_{\text{even}}^{\text{inner}} = \{4, 8\}$ and $\mathcal{S}_{\text{even}}^{\text{outer}} = \{0, 12\}$. By collecting the indices of nonzero bits of $\boldsymbol{d}$ in $\mathcal{B}_{\boldsymbol{d}} = \{j : d_j = 1\} = \{4, 16, 28, 40\}$,

$$v_{\boldsymbol{\alpha}} = (-1)^{\langle \boldsymbol{d}, \boldsymbol{\alpha} \rangle} = \prod_{j \in \mathcal{B}_{\boldsymbol{d}}} (-1)^{\alpha_j}.$$

Denote by $\boldsymbol{a}_{\boldsymbol{\alpha}} = (c_F(\boldsymbol{\alpha}, \boldsymbol{\beta}))_{\boldsymbol{\beta} \in U^{\perp}}$ a column of A.

We note that we care about precisely two input and output bits for each Sbox. It turns out that among the eight Sboxes, there are only three distinct Sboxes. Three different partial LATs are given in Table 10.3. The following is straightforward to derive from Figure 10.1 and Table 10.2.

Table 10.3: Partial LATs for the different Sboxes appearing in the invariant subspace in PRINTcipher.

(a) Sboxes indexed by $\mathcal{S}_{\text{odd}}$.

		$(\boldsymbol{\alpha})$		
		2	4	6
$(\boldsymbol{\beta})$	2	-		-
	4		+	+
	6	-	-	

(b) Sboxes indexed by $\mathcal{S}_{\text{even}}^{\text{outer}}$.

		$(\boldsymbol{\alpha})$		
		1	2	3
$(\boldsymbol{\beta})$	1	+		-
	2		-	-
	3	+	-	

(c) Sboxes indexed by $\mathcal{S}_{\text{even}}^{\text{inner}}$.

		$(\boldsymbol{\alpha})$		
		1	2	3
$(\boldsymbol{\beta})$	1	-		-
	2		+	-
	3	-	+	

Proposition 10.7 With a weak (partial) permutation key and xor key, the Sboxes indexed by $\mathcal{S}_{\text{odd}}$ use the partial LAT in Table 10.3a. The Sboxes indexed by $\mathcal{S}_{\text{even}}^{\text{outer}}$ use the partial LAT in Table 10.3b, while those indexed by $\mathcal{S}_{\text{even}}^{\text{inner}}$ use the partial LAT in Table 10.3c.

An *active* Sbox is an Sbox with a nonzero input mask. For any input mask, the Sboxes that contribute to the biased output masks are precisely the active Sboxes.

Lemma 10.8 $\sum_{\boldsymbol{\beta}} |a_{\boldsymbol{\alpha},\boldsymbol{\beta}}| = 1$. More precisely, there are 2^j nonzero elements of $\boldsymbol{a}_{\boldsymbol{\alpha}}$, each being $\pm 2^{-j}$, where j is the number of Sboxes that are activated by the input mask $\boldsymbol{\alpha}$.

Proof. This is certainly true for $j = 0$. Further, it can be seen from the partial LATs that each nonzero input mask yields precisely two different output masks with nonzero correlation. Thus, there will arise precisely 2^j output masks with nonzero $c_F(\boldsymbol{\alpha}, \boldsymbol{\beta})$. In particular, with $j = 1$, there are 2 nonzero correlations, and they are $\pm 2^{-1}$. For $j > 1$, $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ can be written as $\boldsymbol{\alpha} = \boldsymbol{\alpha}^0 + \boldsymbol{\alpha}^1 + \ldots + \boldsymbol{\alpha}^{j-1}$ and $\boldsymbol{\beta} = \boldsymbol{\beta}^0 + \boldsymbol{\beta}^1 + \ldots + \boldsymbol{\beta}^{j-1}$, where each $\boldsymbol{\alpha}^l$ activates precisely one Sbox, and $\boldsymbol{\beta}^l$ is the corresponding part of the output mask. By Proposition 4.4 it then follows that each nonzero $c_F(\boldsymbol{\alpha}, \boldsymbol{\beta})$ is $\pm 2^{-j}$. ∎

Proving $\boldsymbol{v}A = \boldsymbol{v}$ amounts to showing that $\boldsymbol{v}\boldsymbol{a}_{\boldsymbol{\alpha}} = v_{\boldsymbol{\alpha}}$, $\forall \boldsymbol{\alpha}$. It is clear that $\boldsymbol{v}\boldsymbol{a}_{\mathbf{0}} = 1 = v_{\mathbf{0}}$, so we focus on nonzero $\boldsymbol{\alpha}$. Recall that $\boldsymbol{v}\boldsymbol{a}_{\boldsymbol{\alpha}} = \sum_{\boldsymbol{\beta} \in U^{\perp}} v_{\boldsymbol{\beta}} c_F(\boldsymbol{\alpha}, \boldsymbol{\beta})$.

We know from Lemma 10.8 that $\boldsymbol{a}_{\boldsymbol{\alpha}}$ sums absolutely to 1. Since $v_{\boldsymbol{\alpha}} \in \{-1, 1\}$, $\boldsymbol{v}\boldsymbol{a}_{\boldsymbol{\alpha}}$ is the sum of the (nonzero) elements of $\boldsymbol{a}_{\boldsymbol{\alpha}}$ with or without a change of sign. Thus, to prove $\boldsymbol{v}A = \boldsymbol{v}$, we need to show that $\boldsymbol{v}\boldsymbol{a}_{\boldsymbol{\alpha}} \in \{-1, 1\}$, i.e., the elements of $\boldsymbol{a}_{\boldsymbol{\alpha}}$ are always added constructively, and that the sum is $v_{\boldsymbol{\alpha}}$ (as opposed to $-v_{\boldsymbol{\alpha}}$).

Lemma 10.9 Let S_i, $i \in \mathcal{S}$, be the only active Sbox for the input mask $\boldsymbol{\alpha}$. Then $\boldsymbol{v}\boldsymbol{a}_{\boldsymbol{\alpha}} = v_{\boldsymbol{\alpha}}$.

Proof. We know from Lemma 10.8 that there are two nonzero $c_F(\boldsymbol{\alpha}, \boldsymbol{\beta})$. Further, there are at most two nonzero bits α_{a_0} and α_{a_1} where the precise values of a_0 and a_1 are known (and depend on i). Thus, we can write $\boldsymbol{\alpha} = (\alpha_{a_0}, \alpha_{a_1})$, or $\boldsymbol{\alpha} = \alpha_{a_0}\alpha_{a_1}$, for short. Similarly, $v_{\beta_{b_0}\beta_{b_1}}$ is the $v_{\boldsymbol{\beta}}$ where $\boldsymbol{\beta}$ is nonzero only in one or two of the bits indexed by b_0 and b_1. We have

$$\boldsymbol{v}\boldsymbol{a}_{\boldsymbol{\alpha}} = \sum_{\boldsymbol{\beta} \in U^{\perp}} v_{\boldsymbol{\beta}} c_F(\boldsymbol{\alpha}, \boldsymbol{\beta}) = v_{10} c_F(\boldsymbol{\alpha}, 10) + v_{01} c_F(\boldsymbol{\alpha}, 01) + v_{11} c_F(\boldsymbol{\alpha}, 11).$$

From the characterization of the Sboxes, it is sufficient to show the lemma for each of the three types of Sboxes. Since their LATs all have their nonzero elements in the same pattern, we have

$$\boldsymbol{v}\boldsymbol{a}_{\boldsymbol{\alpha}} = \begin{cases} v_{10} c_F(10, 10) + v_{11} c_F(10, 11), & \boldsymbol{\alpha} = 10, \\ v_{01} c_F(01, 01) + v_{11} c_F(01, 11), & \boldsymbol{\alpha} = 01, \\ v_{10} c_F(11, 10) + v_{01} c_F(11, 01), & \boldsymbol{\alpha} = 11, \end{cases}$$

for all $i \in \mathcal{S}$. In the following, it might be useful to refer to Figure 10.5, where bits indexed by $\mathcal{B}_d$ (one per Sbox) have been drawn thicker. Recall that they affect the signs of the elements $v_{\boldsymbol{\alpha}}$ and $v_{\boldsymbol{\beta}}$.

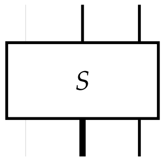
First, consider $i \in \mathcal{S}_{\text{odd}}$:

$$\boldsymbol{v}\boldsymbol{a}_{\boldsymbol{\alpha}} = \begin{cases} v_{10} c_F(10, 10) + v_{11} c_F(10, 11) = (- \cdot -) + (- \cdot -) = +, & \boldsymbol{\alpha} = 10, \\ v_{01} c_F(01, 01) + v_{11} c_F(01, 11) = (+ \cdot +) + (- \cdot -) = +, & \boldsymbol{\alpha} = 01, \\ v_{10} c_F(11, 10) + v_{01} c_F(11, 01) = (- \cdot -) + (+ \cdot +) = +, & \boldsymbol{\alpha} = 11. \end{cases}$$

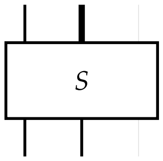
We see that the two nonzero elements add up constructively and that we always get $\boldsymbol{v}\boldsymbol{a}_{\boldsymbol{\alpha}} = +1$. But we also have $v_{\boldsymbol{\alpha}} = (-1)^0 = 1$.

Second, consider $i \in \mathcal{S}_{\text{even}}^{\text{inner}}$:

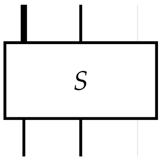
$$\boldsymbol{v}\boldsymbol{a}_{\boldsymbol{\alpha}} = \begin{cases} v_{10} c_F(10, 10) + v_{11} c_F(10, 11) = (+ \cdot -) + (+ \cdot -) = -, & \boldsymbol{\alpha} = 10, \\ v_{01} c_F(01, 01) + v_{11} c_F(01, 11) = (+ \cdot +) + (+ \cdot +) = +, & \boldsymbol{\alpha} = 01, \\ v_{10} c_F(11, 10) + v_{01} c_F(11, 01) = (+ \cdot -) + (+ \cdot -) = -, & \boldsymbol{\alpha} = 11. \end{cases}$$



(a) Sboxes indexed by $\mathcal{S}_{\text{odd}}$.



(b) Sboxes indexed by $\mathcal{S}_{\text{even}}^{\text{outer}}$.



(c) Sboxes indexed by $\mathcal{S}_{\text{even}}^{\text{inner}}$.

Figure 10.5: A schematic overview of the different Sboxes appearing in the invariant subspace in PRINTcipher. Marked bits are those considered in the partial LATs. Thicker bits are those that are indexed by $\mathcal{B}_d$, i.e., that affect $v_{\boldsymbol{\alpha}}$ or $v_{\boldsymbol{\beta}}$.

We see that the two nonzero elements add up constructively and that we always get $\boldsymbol{v}\boldsymbol{a}_{\boldsymbol{\alpha}} = (-1)^{\alpha_{a_0}}$. But we also have $v_{\boldsymbol{\alpha}} = (-1)^{\alpha_{a_0}}$.

Third, consider $i \in \mathcal{S}_{\text{even}}^{\text{outer}}$:

$$\boldsymbol{v}\boldsymbol{a}_{\boldsymbol{\alpha}} = \begin{cases} v_{10}c_F(10,10) + v_{11}c_F(10,11) = (+\cdot+) + (+\cdot+) = +, & \boldsymbol{\alpha} = 10, \\ v_{01}c_F(01,01) + v_{11}c_F(01,11) = (+\cdot-) + (+\cdot-) = -, & \boldsymbol{\alpha} = 01, \\ v_{10}c_F(11,10) + v_{01}c_F(11,01) = (+\cdot-) + (+\cdot-) = -, & \boldsymbol{\alpha} = 11. \end{cases}$$

We see that the two nonzero elements add up constructively and that we always get $\boldsymbol{v}\boldsymbol{a}_{\boldsymbol{\alpha}} = (-1)^{\alpha_{a_1}}$. But we also have $v_{\boldsymbol{\alpha}} = (-1)^{\alpha_{a_1}}$. ■

Thus, on our way to proving that $\boldsymbol{v}A = \boldsymbol{v}$, i.e., $\boldsymbol{v}\boldsymbol{a}_{\boldsymbol{\alpha}} = v_{\boldsymbol{\alpha}}$ for $\boldsymbol{\alpha} \in U^{\perp}$, we have seen that $\boldsymbol{v}\boldsymbol{a}_{\boldsymbol{\alpha}} = v_{\boldsymbol{\alpha}}$ when $\boldsymbol{\alpha}$ only activates one Sbox. It remains to show that $\boldsymbol{v}\boldsymbol{a}_{\boldsymbol{\alpha}} = v_{\boldsymbol{\alpha}}$ when more than one Sbox of $\mathcal{S}$ is activated. This follows from the following by induction:

Lemma 10.10 Assume that with $\boldsymbol{\alpha}^0$, $\boldsymbol{v}\boldsymbol{a}_{\boldsymbol{\alpha}^0} = v_{\boldsymbol{\alpha}^0}$, where $\boldsymbol{\alpha}^0$ activates $j \geq 1$ Sboxes. From $\boldsymbol{\alpha}^1$, activating precisely one Sbox S_i not activated by $\boldsymbol{\alpha}^0$, define $\boldsymbol{\alpha} = \boldsymbol{\alpha}^0 + \boldsymbol{\alpha}^1$. Then $\boldsymbol{v}\boldsymbol{a}_{\boldsymbol{\alpha}} = v_{\boldsymbol{\alpha}}$.

Proof sketch. There are two nonzero elements in $\boldsymbol{a}_{\boldsymbol{\alpha}^1}$. Write them as $c_F(\boldsymbol{\alpha}^1, \boldsymbol{\beta}_0^1)$ and $c_F(\boldsymbol{\alpha}^1, \boldsymbol{\beta}_1^1)$, where $\boldsymbol{\beta}_0^1 \neq \boldsymbol{\beta}_1^1$.

Each nonzero element (»old element«) in $\boldsymbol{a}_{\boldsymbol{\alpha}^0}$ yields two nonzero elements (»new elements«) in $\boldsymbol{a}_{\boldsymbol{\alpha}}$. By Proposition 4.4, the sign of such a new element differs from that of the old precisely when $c_F(\boldsymbol{\alpha}^1, \boldsymbol{\beta}_b^1) < 0$; we say that *the sign changes*. There are essentially two cases to consider: $i \in \mathcal{S}_{\text{odd}}$ and $i \in \mathcal{S}_{\text{even}}$.

Assume that $i \in \mathcal{S}_{\text{odd}}$. Then the sign changes precisely when $\boldsymbol{\beta}_b^1 \in \{2, 6\}$. This corresponds to a bit of $\boldsymbol{\beta}$ used in calculating $v_{\boldsymbol{\beta}}$. This means that the new element changes sign precisely when the corresponding element in $\boldsymbol{\alpha}$ changes sign.

Assume that $i \in \mathcal{S}_{\text{even}}$. Then the sign changes for some certain values of $\boldsymbol{\alpha}$, depending on i. The sign change corresponds to a bit of $\boldsymbol{\alpha}$ used in calculating $v_{\boldsymbol{\alpha}}$. This means that the new element changes sign precisely when the corresponding element in $\boldsymbol{v}$ changes sign.

Thus, the sign changes always cancel ($i \in \mathcal{S}_{\text{odd}}$) or match ($i \in \mathcal{S}_{\text{even}}$), so we have $\boldsymbol{v}\boldsymbol{a}_{\boldsymbol{\alpha}} = v_{\boldsymbol{\alpha}}$. ∎

We offer the following recapitulation and interpretation: The LAT of the entire PRINTcipher round is constructed from several smaller LATs for the different Sboxes. There are in fact only three distinct partial LATs, when we account for the partial permutation and xor keys. At this stage, we can see how each column of the partial correlation matrix adds up to one, absolutely.

We depend on each *row* of Table 10.3a containing a constant sign, which »matches« a particular bit of the input mask. Similarly, each *column* of Tables 10.3b and 10.3c contain a constant sign: two columns contain minuses and one, precisely the one corresponding to the bit used in the rule for $\boldsymbol{v}$, contains pluses.

For the resulting big LAT (A), all columns can be added up to ± 1 by using a very simple rule ($\boldsymbol{v}$) for choosing the signs. In particular, $\boldsymbol{v}$ is an eigenvector.

10.5 CONCLUSION

We have seen how the large biases that follow from the invariant subspace property in a block cipher can be explained in terms of an eigenvector with eigenvalue 1 which arises in the correlation matrix of the round function. A loose summary could be that »linear approximations with input and output masks chosen according to the invariant subspace will cause trail clustering inside the invariant subspace—assuming that the linear hull behaves well outside the invariant subspace, the linear approximations will have large biases.«

Further, by Theorem 10.5 the invariant subspace is a necessary and sufficient condition for this kind of trail clustering. However, it should be noted that the theorem does not make any predictions about the eigenvalue 1, other than that it *will* show up when there is an invariant subspace. Also, Theorem 10.5 does not rule out that the rest of the linear hull cancels the effect of the trail clustering, yielding small correlations $c_F(\boldsymbol{\alpha}, \boldsymbol{\beta}) = c_F^{\mathrm{i}}(\boldsymbol{\alpha}, \boldsymbol{\beta}) + c_F^{\mathrm{o}}(\boldsymbol{\alpha}, \boldsymbol{\beta})$.

More to the point, Theorem 10.5 does not guarantee »good behavior« in the absence of invariant subspaces. It does however make »bad behavior« slightly less likely in a certain sense. Choosing a function F at random, there is some *a priori* probability distribution for eigenvectors and -values. If one can show that there are no invariant subspaces, the *a posteriori* probabilities are known to be 0 for the eigenvectors considered in Theorem 10.5 with eigenvalues 1.

It is worth noting that the eigenvalue 1 by itself is not a sign of trail clustering: if the eigenvector $\boldsymbol{v}$ is »irregular« (possibly containing complex-valued elements), the limit $\boldsymbol{v}^{\mathrm{T}}\boldsymbol{v}$ can certainly have (possibly complex-valued) elements that are all close to 0. The eigenvectors considered in Theorem 10.5, however, have a certain structure, which carries over into the limit matrix and yields elements which are plus or minus a relatively large value.

It would be interesting to see future work try to identify (a) tighter link(s) between »bad behavior« of the function F from a randomness perspective (here: an invariant subspace) and »bad behavior« of the linear hull (here: trail clustering which may or may not be canceled by the rest of the linear hull).

Concluding Remarks

In this dissertation, some new designs in the area of symmetric lightweight cryptography have been proposed, and various cryptanalytic results have been presented. A new class of hardware-attractive universal hash functions has been proposed. Grain-128a, the newest member of the Grain family of stream ciphers has been presented. Various cryptanalytic results have been proposed on the stream cipher BEAN and the block ciphers KTANTAN and PRINTcipher. The relation between linear correlations and invariant subspaces, first observed in PRINTcipher, has been analyzed.

While most of the attacks in this dissertation require at least one of a reduced number of rounds, an impractical time complexity, and an unrealistic attack setting, they have advanced the understanding of the specific primitives cryptanalyzed. Hopefully, they have also increased the knowledge about the design strategies used in these primitives regarding, e.g., FCSR combiners in keystream generators or key schedules in block ciphers.

It remains to be seen how far the lightweight trend can go as measured in, e.g., the number of gates used for a hardware implementation. If and when a successor or alternative to AES will be standardized, it would be interesting to see to what extent the cryptanalysis of lightweight designs, which often balance on the edge of insecurity, has advanced the knowledge of what can be included in, and excluded from, a robust, long-term secure primitive.

Acronyms

ε-AXU	ε-almost xor universal
AES	Advanced Encryption Standard
ANF	algebraic normal form
BSC	binary symmetric channel
CBC	cipher block chaining
CTR	counter mode
DES	Digital Encryption Standard
ECB	electronic codebook
FCSR	feedback with carry shift register
GCM	Galois counter mode
iid	independent and identically distributed
IV	initialization vector
LAT	linear approximation table
LDPC	low-density parity-check
LFSR	linear feedback shift register
LTE	Long Term Evolution
MAC	message authentication code
NESSIE	New European Schemes for Signatures, Integrity and Encryption
NFSR	nonlinear feedback shift register
NIST	National Institute of Standards and Technology
OTP	one-time pad
PRNG	pseudorandom number generator
SPN	substitution–permutation network
UMTS	Universal Mobile Telecommunications System

References

[AÅBL12] M. A. Abdelraheem, M. Ågren, P. Beelen, and G. Leander, »On the distribution of linear biases: Three instructive examples,« in *Advances in Cryptology—CRYPTO 2012*, ser. Lecture Notes in Computer Science, R. Safavi-Naini and R. Canetti, Eds., vol. 7417. Springer-Verlag, 2012, pp. 50–67.

[AB04] F. Arnault and T. Berger, »Design of new pseudo random generators based on filtered FCSR automaton,« October 2004, the State of the Art of Stream Ciphers, Workshop Record, SASC 2004, Brugge, Belgium.

[ABL06] F. Arnault, T. Berger, and C. Lauradoux, »Update on F-FCSR stream cipher,« eSTREAM, ECRYPT Stream Cipher Project, Report 2006/025, 2006, http://www.ecrypt.eu.org/stream.

[ABM08] F. Arnault, T. Berger, and M. Minier, »Some results on FCSR automata with applications to the security of FCSR-based pseudorandom generators,« *IEEE Transactions on Information Theory*, vol. 54, no. 2, pp. 836–840, February 2008.

[ADH$^+$09] J.-P. Aumasson, I. Dinur, L. Henzen, W. Meier, and A. Shamir, »Efficient FPGA implementations of high-dimensional cube testers on the stream cipher Grain-128,« in *Workshop on Special Purpose Hardware for Attacking Cryptographic Systems (SHARCS'09)*, 2009.

[AGHP90] N. Alon, O. Goldreich, J. Håstad, and R. Peralta, »Simple construction of almost k-wise independent random variables,« *Annual IEEE Symposium on Foundations of Computer Science*, pp. 544–553, 1990.

[ÅHJM11] M. Ågren, M. Hell, T. Johansson, and W. Meier, »Grain-128a: A new version of Grain-128 with optional authentication,« *International Journal of Wireless and Mobile Computing*, vol. 5, no. 1, pp. 48–59, 2011.

[ALZ11] M. A. Abdelraheem, G. Leander, and E. Zenner, »Differential cryptanalysis of round-reduced PRINTcipher: Computing roots of permutation,« in *Fast Software Encryption—FSE 2011*, ser. Lecture Notes in Computer Science, A. Joux, Ed., vol. 6733. Springer-Verlag, 2011, pp. 1–17.

[BBS06] E. Barkan, E. Biham, and A. Shamir, »Rigorous bounds on cryptanalytic time/memory tradeoffs,« in *Advances in Cryptology—CRYPTO 2006*, ser. Lecture Notes in Computer Science, C. Dwork, Ed., vol. 4117. Springer-Verlag, 2006, pp. 1–21.

[BCK] M. Bellare, R. Canetti, and H. Krawczyk, »Keying hash functions for message authentication,« in *Advances in Cryptology—CRYPTO'96*, ser. Lecture Notes in Computer Science, N. Koblitz, Ed., vol. 1109. Springer-Verlag, pp. 1–15.

[BGJ08] C. Berbain, H. Gilbert, and A. Joux, »Algebraic and correlation attacks against linearly filtered non linear feedback shift registers,« in *Selected Areas in Cryptography—SAC 2008*, ser. Lecture Notes in Computer Science, R. Avanzi, L. Keliher, and F. Sica, Eds., vol. 5381. Springer-Verlag, 2008, pp. 184–198.

[BGM06] C. Berbain, H. Gilbert, and A. Maximov, »Cryptanalysis of Grain,« in *Fast Software Encryption—FSE 2006*, ser. Lecture Notes in Computer Science, M. Robshaw, Ed., vol. 4047. Springer-Verlag, 2006, pp. 15–29.

[Bih94] E. Biham, »New types of cryptanalytic attacks using related keys,« *Journal of Cryptology*, vol. 7, no. 4, pp. 229–246, 1994.

[BJV04] T. Baignères, P. Junod, and S. Vaudenay, »How far can we go beyond linear cryptanalysis?« in *Advances in Cryptology—ASIACRYPT 2004*, ser. Lecture Notes in Computer Science, P. J. Lee, Ed., vol. 3329. Springer-Verlag, 2004, pp. 432–450.

[BKL$^+$07] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe, »PRESENT: An ultra-lightweight block cipher,« in *Cryptographic Hardware and Embedded Systems—CHES 2007*, ser. Lecture Notes in Computer Science, P. Paillier and I. Verbauwhede, Eds., vol. 4727. Springer-Verlag, 2007, pp. 450–466.

[BKR11] A. Bogdanov, D. Khovratovich, and C. Rechberger, »Biclique cryptanalysis of the full AES,« in *Advances in Cryptology—ASIACRYPT 2011*, ser. Lecture Notes in Computer Science, D. H. Lee and X. Wang, Eds., vol. 7073. Springer-Verlag, 2011, pp. 344–371.

[BL05] A. Braeken and J. Lano, »On the (im)possibility of practical and secure nonlinear filters and combiners.« in *Selected Areas in Cryptography—SAC 2005*, ser. Lecture Notes in Computer Science, B. Preneel and S. Tavares, Eds., vol. 3897. Springer-Verlag, 2005, pp. 159–174.

[BLP08] D. J. Bernstein, T. Lange, and C. Peters, »Attacking and defending the McEliece cryptosystem,« in *Post-Quantum Cryptography—PQCrypto 2008*, ser. Lecture Notes in Computer Science, J. Buchmann and J. Ding, Eds., vol. 5299. Springer-Verlag, 2008, pp. 31–46.

[BR10] A. Bogdanov and C. Rechberger, »A 3-subset meet-in-the-middle attack: cryptanalysis of the lightweight block cipher KTANTAN,« in *Selected Areas in Cryptography—SAC 2010*, ser. Lecture Notes in Computer Science, A. Biryukov, G. Gong, and D. R. Stinson, Eds., vol. 6544. Springer-Verlag, 2010, pp. 229–240.

[BR11] A. Bogdanov and V. Rijmen, »Linear hulls with correlation zero and linear cryptanalysis of block ciphers,« Cryptology ePrint Archive, Report 2011/123, 2011, http://eprint.iacr.org/2011/123.

[BS93] E. Biham and A. Shamir, *Differential Cryptanalysis of the Data Encryption Standard*. Springer-Verlag, 1993.

[BS00] A. Biryukov and A. Shamir, »Cryptanalytic time/memory/data tradeoffs for stream ciphers,« in *Advances in Cryptology—ASIACRYPT 2000*, ser. Lecture Notes in Computer Science, T. Okamoto, Ed., vol. 1976. Springer-Verlag, 2000, pp. 1–13.

[BW99] A. Biryukov and D. Wagner, »Slide attacks,« in *Fast Software Encryption—FSE'99*, ser. Lecture Notes in Computer Science, L. R. Knudsen, Ed., vol. 1636. Springer-Verlag, 1999, pp. 245–259.

[Car10a] C. Carlet, »Boolean functions for cryptography and error-correcting codes,« in *Boolean Models and Methods in Mathematics, Computer Science, and Engineering*, Y. Crama and P. L. Hammer, Eds. Cambridge University Press, 2010, pp. 257–397.

[Car10b] C. Carlet, »Vectorial boolean functions for cryptography,« in *Boolean Models and Methods in Mathematics, Computer Science, and Engineering*, Y. Crama and P. L. Hammer, Eds. Cambridge University Press, 2010, pp. 398–469.

[CC98] A. Canteaut and F. Chabaud, »A new algorithm for finding minimum-weight words in a linear code: Application to McEliece's cryptosystem and to narrow-sense BCH codes of length 511,« *IEEE Transactions on Information Theory*, vol. 44, no. 1, pp. 367–378, January 1998.

[CHJ02] D. Coppersmith, S. Halevi, and C. Jutla, »Cryptanalysis of stream ciphers with linear masking,« in *Advances in Cryptology—CRYPTO 2002*, ser. Lecture Notes in Computer Science, M. Yung, Ed., vol. 2442. Springer-Verlag, 2002, pp. 515–532.

[CJM02] P. Chose, A. Joux, and M. Mitton, »Fast correlation attacks: An algorithmic point of view,« in *Advances in Cryptology—EUROCRYPT 2002*, ser. Lecture Notes in Computer Science, L. Knudsen, Ed., vol. 2332. Springer-Verlag, 2002, pp. 209–221.

[CJS00] V. Chepyzhov, T. Johansson, and B. Smeets, »A simple algorithm for fast correlation attacks on stream ciphers,« in *Fast Software Encryption—FSE 2000*, ser. Lecture Notes in Computer Science, B. Schneier, Ed., vol. 1978. Springer-Verlag, 2000, pp. 181–195.

[CM03] N. Courtois and W. Meier, »Algebraic attacks on stream ciphers with linear feedback,« in *Advances in Cryptology—EUROCRYPT 2003*, ser. Lecture Notes in Computer Science, E. Biham, Ed., vol. 2656. Springer-Verlag, 2003, pp. 345–359.

[CT00] A. Canteaut and M. Trabbia, »Improved fast correlation attacks using parity-check equations of weight 4 and 5,« in *Advances in Cryptology—EUROCRYPT 2000*, ser. Lecture Notes in Computer Science, B. Preneel, Ed., vol. 1807. Springer-Verlag, 2000, pp. 573–588.

[DDK09] C. De Cannière, O. Dunkelman, and M. Knežević, »KATAN and KTANTAN — a family of small and efficient hardware-oriented block ciphers,« in *Cryptographic Hardware and Embedded Systems—CHES 2009*, ser. Lecture Notes in Computer Science, C. Clavier and K. Gaj, Eds., vol. 5747. Springer-Verlag, 2009, pp. 272–288.

[DGP$^+$11] I. Dinur, T. Güneysu, C. Paar, A. Shamir, and R. Zimmermann, »An experimentally verified attack on full Grain-128 using dedicated reconfigurable hardware,« Cryptology ePrint Archive, Report 2011/282, 2011, http://eprint.iacr.org/2011/282.

[DGV95] J. Daemen, R. Govaerts, and J. Vandewalle, »Correlation matrices,« in *Fast Software Encryption—FSE'94*, ser. Lecture Notes in Computer Science, B. Preneel, Ed., vol. 1008. Springer-Verlag, 1995, pp. 275–285.

[DH76] W. Diffie and M. E. Hellman, »New directions in cryptography,« *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, November 1976.

[DH77] W. Diffie and M. E. Hellman, »Exhaustive cryptanalysis of the NBS data encryption standard,« *Computer*, vol. 10, no. 6, pp. 74–84, 1977.

[DKP08] C. De Cannière, Ö. Küçük, and B. Preneel, »Analysis of Grain's initialization algorithm,« in *Progress in Cryptology—AFRICACRYPT 2008*, ser. Lecture Notes in Computer Science, S. Vaudenay, Ed., vol. 5023. Springer-Verlag, 2008, pp. 276–289.

[DP08] C. De Cannière and B. Preneel, »Trivium,« in *New Stream Cipher Designs*, ser. Lecture Notes in Computer Science, M. Robshaw and O. Billet, Eds., vol. 4986. Springer-Verlag, 2008, pp. 244–266.

[DR01] J. Daemen and V. Rijmen, »The wide trail design strategy,« in *Cryptography and Coding 2001*, ser. Lecture Notes in Computer Science, B. Honary, Ed., vol. 2260. Springer-Verlag, 2001, pp. 222–238.

[DR02] J. Daemen and V. Rijmen, *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer-Verlag, 2002.

[DS11] I. Dinur and A. Shamir, »Breaking Grain-128 with dynamic cube attacks,« in *Fast Software Encryption—FSE 2011*, ser. Lecture Notes in Computer Science, A. Joux, Ed., vol. 6733. Springer-Verlag, 2011, pp. 167–187.

[EHJ07] H. Englund, M. Hell, and T. Johansson, »Two general attacks on Pomaranch-like keystream generators,« in *Fast Software Encryption—FSE 2007*, ser. Lecture Notes in Computer Science, A. Biryukov, Ed., vol. 4593. Springer-Verlag, 2007, pp. 274–289.

[EJ05] H. Englund and T. Johansson, »A new simple technique to attack filter generators and related ciphers,« in *Selected Areas in Cryptography—SAC 2004*, ser. Lecture Notes in Computer Science, H. Handschuh and M. A. Hasan, Eds., vol. 3357. Springer-Verlag, 2005, pp. 39–53.

[ETS09] ETSI/SAGE, »Specification of the 3GPP confidentiality and integrity algorithms UEA2 & UIA2. Document 1: UEA2 and UIA2 specification,« 2009.

[ETS11a] ETSI/SAGE, »Specification of the 3GPP confidentiality and integrity algorithms 128-EEA3 & 128-EIA3. Document 1: 128-EEA3 and 128-EIA3 specification,« 2011.

[ETS11b] ETSI/SAGE, »Specification of the 3GPP confidentiality and integrity algorithms 128-EEA3 & 128-EIA3. Document 2: ZUC specification,« 2011.

[FGKV07] W. Fischer, B. M. Gammel, O. Kniffler, and J. Velten, »Differential power analysis of stream ciphers,« in *Topics in Cryptology—CT-RSA 2007*, ser. Lecture Notes in Computer Science, M. Abe, Ed., vol. 4377. Springer-Verlag, 2007, pp. 257–270.

[FGRV10] T. Fuhr, H. Gilbert, J.-R. Reinhard, and M. Videau, »A forgery attack on the candidate LTE integrity algorithm 128-EIA3 (updated version),« Cryptology ePrint Archive, Report 2010/618, 2010, http://eprint.iacr.org/2010/618.

[FGRV12] T. Fuhr, H. Gilbert, J.-R. Reinhard, and M. Videau, »Analysis of the initial and modified versions of the candidate 3GPP integrity algorithm 128-EIA3,« in *Selected Areas in Cryptography—SAC 2011*, ser. Lecture Notes in Computer Science, A. Miri and S. Vaudenay, Eds., vol. 7118. Springer-Verlag, 2012, pp. 230–242.

[FI92] S. H. Friedberg and A. J. Insel, »Convergence of matrix powers,« *International Journal of Mathematical Education in Science and Technology*, vol. 23, no. 5, pp. 765–769, 1992.

[FMS08] S. Fischer, W. Meier, and D. Stegemann, »Equivalent representations of the F-FCSR keystream generator,« February 2008, the State of the Art of Stream Ciphers, Workshop Record, SASC 2008, Lausanne, Switzerland.

[GMS74] E. N. Gilbert, F. J. MacWilliams, and N. J. A. Sloane, »Codes which detect deception,« *Bell Systems Technical Journal*, vol. 53, no. 3, pp. 405–424, 1974.

[Gol94] J. D. Golić, »Intrinsic statistical weakness of keystream generators,« in *Advances in Cryptology—ASIACRYPT'94*, ser. Lecture Notes in Computer Science, J. Pieprzyk and R. Safavi-Naini, Eds., vol. 917. Springer-Verlag, 1994, pp. 91–103.

[Gol96] J. D. Golić, »Computation of low-weight parity-check polynomials,« *Electronic Letters*, vol. 32, no. 21, pp. 1981–1982, October 1996.

[Hel80] M. E. Hellman, »A cryptanalytic time–memory trade-off,« *IEEE Transactions on Information Theory*, vol. 26, no. 4, pp. 401–406, July 1980.

[HJ07] M. Hell and T. Johansson, »On the problem of finding linear approximations and cryptanalysis of Pomaranch version 2,« in *Selected Areas in Cryptography—SAC 2006*, ser. Lecture Notes in Computer Science, E. Biham, Ed., vol. 4356. Springer-Verlag, 2007, pp. 220–234.

[HJ08] M. Hell and T. Johansson, »Breaking the F-FCSR-H stream cipher in real time,« in *Advances in Cryptology—ASIACRYPT 2008*, ser. Lecture Notes in Computer Science, J. Pieprzyk, Ed., vol. 5350. Springer-Verlag, 2008, pp. 557–569.

[HJ11] M. Hell and T. Johansson, »Linear attacks on stream ciphers,« in *Advanced Linear Cryptanalysis of Block and Stream Ciphers*, P. Junod and A. Canteaut, Eds. IOS Press, 2011, pp. 55–85.

[HJB08] M. Hell, T. Johansson, and L. Brynielsson, »An overview of distinguishing attacks on stream ciphers,« *Cryptography and Communications*, vol. 1, no. 1, pp. 71–94, 2008.

[HJM06] M. Hell, T. Johansson, and W. Meier, »Grain: A stream cipher for constrained environments.« *International Journal of Wireless and Mobile Computing*, vol. 2, no. 1, pp. 86–93, 2006.

[HJMM06] M. Hell, T. Johansson, A. Maximov, and W. Meier, »A stream cipher proposal: Grain-128,« in *International Symposium on Information Theory—ISIT 2006*. IEEE, 2006.

[HP08] H. Handschuh and B. Preneel, »Key-recovery attacks on universal hash function based MAC algorithms,« in *Advances in Cryptology—CRYPTO 2008*, ser. Lecture Notes in Computer Science, D. Wagner, Ed., vol. 5157. Springer-Verlag, 2008, pp. 144–161.

[HS04] J. Hoch and A. Shamir, »Fault analysis of stream ciphers.« in *Cryptographic Hardware and Embedded Systems—CHES 2004*, ser. Lecture Notes in Computer Science, M. Joye and J.-J. Quisquater, Eds., vol. 3156. Springer-Verlag, 2004, pp. 240–253.

[Int99] International Organization for Standardization (ISO), »ISO/IEC 9797-1, information technology – security techniques – message authentication codes (MACs) – part 1: Mechanisms using a block cipher,« 1999.

[Int08] International Electrotechnical Commission (IEC), »IEC 80000-13:2008, quantities and units – part 13: Information science and technology,« 2008.

[JHK05] C. Jansen, T. Helleseth, and A. Kholosha, »Cascade jump controlled sequence generator (CJCSG),« eSTREAM, ECRYPT Stream Cipher Project, Report 2005/022, 2005, http://www.ecrypt.eu.org/stream.

[JJ99] T. Johansson and F. Jönsson, »Fast correlation attacks based on turbo code techniques,« in *Advances in Cryptology—CRYPTO'99*, ser. Lecture Notes in Computer Science, M. Wiener, Ed., vol. 1666. Springer-Verlag, 1999, pp. 181–197.

[JJ00] T. Johansson and F. Jönsson, »Fast correlation attacks through reconstruction of linear polynomials,« in *Advances in Cryptology—CRYPTO 2000*, ser. Lecture Notes in Computer Science, M. Bellare, Ed., vol. 1880. Springer-Verlag, 2000, pp. 300–315.

[JKS94] T. Johansson, G. Kabatianskii, and B. Smeets, »On the relation between A-codes and codes correcting independent errors,« in *Advances in Cryptology—EUROCRYPT'93*, ser. Lecture Notes in Computer Science, T. Helleseth, Ed., vol. 765. Springer-Verlag, 1994, pp. 1–11.

[JL11] T. Johansson and C. Löndahl, »An improvement of Stern's algorithm,« Department of Electrical and Information Technology, Lund University, Tech. Rep., 2011, https://lup.lub.lu.se/record/2204753.

[KDH12] F. Karakoç, H. Demirci, and A. E. Harmanci, »Combined differential and linear cryptanalysis of reduced-round PRINTcipher,« in *Selected Areas in Cryptography—SAC 2011*, ser. Lecture Notes in Computer Science, A. Miri and S. Vaudenay, Eds., vol. 7118. Springer-Verlag, 2012, pp. 169–184.

[KG94] A. Klapper and M. Goresky, »2-adic shift registers,« in *Fast Software Encryption—FSE'93*, ser. Lecture Notes in Computer Science, R. Anderson, Ed., vol. 809. Springer-Verlag, 1994, pp. 174–178.

[KG97] A. Klapper and M. Goresky, »Feedback shift registers, 2-adic span, and combiners with memory,« *Journal of Cryptology*, vol. 10, no. 2, pp. 111–147, 1997.

[KJJ99] P. Kocher, J. Jaffe, and B. Jun, »Differential power analysis,« in *Advances in Cryptology—CRYPTO'99*, ser. Lecture Notes in Computer Science, M. Wiener, Ed., vol. 1666. Springer-Verlag, 1999, pp. 388–397.

[KLPR10] L. Knudsen, G. Leander, A. Poschmann, and M. Robshaw, »PRINTcipher: A block cipher for IC-printing,« in *Cryptographic Hardware and Embedded Systems—CHES 2010*, ser. Lecture Notes in Computer Science, S. Mangard and F.-X. Standaert, Eds., vol. 6225. Springer-Verlag, 2010, pp. 16–32.

[KM93] E. Kushilevitz and Y. Mansour, »Learning decision trees using the fourier spectrum,« *SIAM Journal on Computing*, vol. 22, no. 6, pp. 1331–1348, 1993.

[Knu93] L. R. Knudsen, »Cryptanalysis of LOKI 91,« in *Advances in Cryptology—AUSCRYPT'92*, ser. Lecture Notes in Computer Science, J. Seberry and Z. Y, Eds., vol. 718. Springer-Verlag, 1993, pp. 196–208.

[Knu95] L. R. Knudsen, »Truncated and higher order differentials,« in *Fast Software Encryption—FSE'94*, ser. Lecture Notes in Computer Science, B. Preneel, Ed., vol. 1008. Springer-Verlag, 1995, pp. 196–211.

[Knu98] L. R. Knudsen, »DEAL – a 128-bit block cipher,« Department of Informatics, University of Bergen, Tech. Rep. 151, 1998, http://www2.mat.dtu.dk/people/Lars.R.Knudsen/papers/deal.pdf.

[KOJL09] N. Kumar, S. Ojha, K. Jain, and S. Lal, »BEAN: A lightweight stream cipher,« in *Proceedings of the 2nd International Conference on Security of Information and Networks, SIN 2009*. ACM, 2009, pp. 168–171.

[KR94] B. S. Kaliski and M. J. B. Robshaw, »Linear cryptanalysis using multiple approximations,« in *Advances in Cryptology—CRYPTO'94*, ser. Lecture Notes in Computer Science, Y. Desmedt, Ed., vol. 839. Springer-Verlag, 1994, pp. 26–39.

[KR07] L. R. Knudsen and V. Rijmen, »Known-key distinguishers for some block ciphers,« in *Advances in Cryptology—ASIACRYPT 2007*, ser. Lecture Notes in Computer Science, K. Kurosawa, Ed., vol. 4833. Springer-Verlag, 2007, pp. 315–324.

[Kra94] H. Krawczyk, »LFSR-based hashing and authentication,« in *Advances in Cryptology—CRYPTO'94*, ser. Lecture Notes in Computer Science, Y. Desmedt, Ed., vol. 839. Springer-Verlag, 1994, pp. 129–139.

[Kra95] H. Krawczyk, »New hash functions for message authentication,« in *Advances in Cryptology—EUROCRYPT'95*, ser. Lecture Notes in Computer Science, L. C. Guillou and J.-J. Quisquater, Eds., vol. 921. Springer-Verlag, 1995, pp. 301–310.

[Küç06] Ö. Küçük, »Slide resynchronization attack on the initialization of Grain 1.0,« eSTREAM, ECRYPT Stream Cipher Project, Report 2006/044, 2006, http://www.ecrypt.eu.org/stream.

[LAAZ11] G. Leander, M. A. Abdelraheem, H. AlKhzaimi, and E. Zenner, »A cryptanalysis of PRINTcipher: The invariant subspace attack,« in *Advances in Cryptology—CRYPTO 2011*, ser. Lecture Notes in Computer Science, P. Rogaway, Ed., vol. 6841. Springer-Verlag, 2011, pp. 206–221.

[Lea11] G. Leander, »On linear hulls, statistical saturation attacks, PRESENT and a cryptanalysis of PUFFIN,« in *Advances in Cryptology—EUROCRYPT 2011*, ser. Lecture Notes in Computer Science, K. G. Paterson, Ed., vol. 6632. Springer-Verlag, 2011, pp. 303–322.

[Lea12] G. Leander, personal communication, 2012.

[LJSH08] Y. Lee, K. Jeong, J. Sung, and S. Hong, »Related-key chosen IV attacks on Grain-v1 and Grain-128,« in *13th Australasian Conference on Information Security and Privacy—ACISP 2008*, ser. Lecture Notes in Computer Science, Y. Mu, W. Susilo, and J. Seberry, Eds., vol. 5107. Springer-Verlag, 2008, pp. 321–335.

[Mat94a] M. Matsui, »The first experimental cryptanalysis of the Data Encryption Standard,« in *Advances in Cryptology—CRYPTO'94*, ser. Lecture Notes in Computer Science, Y. Desmedt, Ed., vol. 839. Springer-Verlag, 1994, pp. 1–11.

[Mat94b] M. Matsui, »Linear cryptanalysis method for DES cipher,« in *Advances in Cryptology—EUROCRYPT'93*, ser. Lecture Notes in Computer Science, T. Helleseth, Ed., vol. 765. Springer-Verlag, 1994, pp. 386–397.

[Max06] A. Maximov, »Cryptanalysis of the "Grain" family of stream ciphers,« in *ACM Symposium on Information, Computer and Communications Security (ASIACCS'06)*, 2006, pp. 283–288.

[MD12] S. S. Mansouri and E. Dubrova, »An architectural countermeasure against power analysis attacks for FSR-based stream ciphers,« in *Constructive Side-Channel Analysis and Secure Design—COSADE 2012*, ser. Lecture Notes in Computer Science, W. Schindler and S. A. Huss, Eds., vol. 7275. Springer-Verlag, 2012, pp. 54–68.

[MFI02] M. Mihaljević, M. Fossorier, and H. Imai, »Fast correlation attack algorithm with list decoding and an application,« in *Fast Software Encryption—FSE 2001*, ser. Lecture Notes in Computer Science, M. Matsui, Ed., vol. 2355. Springer-Verlag, 2002, pp. 196–210.

[MMT11] A. May, A. Meurer, and E. Tomae, »Decoding random linear codes in $\tilde{\mathcal{O}}\left(2^{0.054n}\right)$,« in *Advances in Cryptology—ASIACRYPT 2011*, ser. Lecture Notes in Computer Science, D. H. Lee and X. Wang, Eds., vol. 7073. Springer-Verlag, 2011, pp. 107–124.

[MS89] W. Meier and O. Staffelbach, »Fast correlation attacks on certain stream ciphers,« *Journal of Cryptology*, vol. 1, no. 3, pp. 159–176, 1989.

[MV04] D. A. McGrew and J. Viega, »The security and performance of the Galois/Counter Mode (GCM) of operation,« in *Progress in Cryptology—INDOCRYPT 2004*, ser. Lecture Notes in Computer Science, A. Canteaut and K. Viswanathan, Eds., vol. 3348. Springer-Verlag, 2004, pp. 343–355.

[NIS10] NIST, »A statistical test suite for random and pseudorandom number generators for cryptographic applications,« NIST Special Publication 800-22b, 2010.

[NIS12] NIST, »Recommendation for the Triple Data Encryption Algorithm (TDEA) block cipher (revised January 2012),« NIST Special Publication 800-67 Revision 1, 2012.

[NN93] J. Naor and M. Naor, »Small-bias probability spaces: Efficient constructions and applications,« *SIAM Journal on Computing*, vol. 22, no. 4, pp. 838–856, 1993.

[Nyb95] K. Nyberg, »Linear approximation of block ciphers,« in *Advances in Cryptology—EUROCRYPT'94*, ser. Lecture Notes in Computer Science, A. D. Santis, Ed., vol. 950. Springer-Verlag, 1995, pp. 439–444.

[Oec03] P. Oechslin, »Making a faster cryptanalytic time-memory tradeoff,« in *Advances in Cryptology—CRYPTO 2003*, ser. Lecture Notes in Computer Science, D. Boneh, Ed. Springer-Verlag, 2003, vol. 2729, pp. 617–630.

[Old40] R. Oldenburger, »Infinite powers of matrices and characteristic roots,« *Duke Mathematical Journal*, vol. 6, no. 2, pp. 357–361, 1940.

[PK95] W. Penzhorn and G. Kühn, »Computation of low-weight parity checks for correlation attacks on stream ciphers,« in *Cryptography and Coding – 5th IMA Conference*, ser. Lecture Notes in Computer Science, C. Boyd, Ed., vol. 1025. Springer-Verlag, 1995, pp. 74–83.

[PL11] N. R. Pillai and Y. K. Lather, »Algebraic attack on BEAN a lightweight stream cipher,« 2011, unpublished manuscript.

[RH02] G. Rose and P. Hawkes, »On the applicability of distinguishing attacks against stream ciphers,« Cryptology ePrint Archive, Report 2002/142, 2002, http://eprint.iacr.org/2002/142.

[Rij10] V. Rijmen, »Practical-titled attack on AES-128 using chosen-text relations,« Cryptology ePrint Archive, Report 2010/337, 2010, http://eprint.iacr.org/2010/337.

[Sar08] P. Sarkar, »A new universal hash function and other cryptographic algorithms suitable for resource constrained devices,« Cryptology ePrint Archive, Report 2008/216, 2008, http://eprint.iacr.org/2008/216.

[SDGM01] L. R. Simpson, E. Dawson, J. D. Golić, and W. L. Millan, »LILI keystream generator,« in *Selected Areas in Cryptography—SAC 2000*, ser. Lecture Notes in Computer Science, D. R. Stinson and S. Tavares, Eds., vol. 2012. Springer-Verlag, 2001, pp. 248–261.

[Sim70] G. J. Simmons, »The number of irreducible polynomials of degree n over GF(p),« *The American Mathematical Monthly*, vol. 77, no. 7, pp. 743–745, 1970.

[Sim92] G. J. Simmons, »A survey of information authentication,« in *Contemporary Cryptology, The Science of Information Integrity*, G. J. Simmons, Ed. IEEE Press, 1992, pp. 379–419.

[Sta10] P. Stankovski, »Greedy distinguishers and nonrandomness detectors,« in *Progress in Cryptology—INDOCRYPT 2010*, ser. Lecture Notes in Computer Science, G. Gong and K. C. Gupta, Eds., vol. 6498. Springer-Verlag, 2010, pp. 210–226.

[Ste89] J. Stern, »A method for finding codewords of small weight,« in *Coding Theory and Applications 1988*, ser. Lecture Notes in Computer Science, G. Cohen and J. Wolfmann, Eds., vol. 388. Springer-Verlag, 1989, pp. 106–113.

[Sti92] D. R. Stinson, »Universal hashing and authentication codes,« in *Advances in Cryptology—CRYPTO'91*, ser. Lecture Notes in Computer Science, J. Feigenbaum, Ed., vol. 576. Springer-Verlag, 1992, pp. 74–85.

[TCG92] A. Tardy-Corfdir and H. Gilbert, »A known-plaintext attack of FEAL-4 and FEAL-6,« in *Advances in Cryptology—CRYPTO'91*, ser. Lecture Notes in Computer Science, J. Feigenbaum, Ed., vol. 576. Springer-Verlag, 1992, pp. 172–182.

[Var97] A. Vardy, »The intractability of computing the minimum distance of a code,« *IEEE Transactions on Information Theory*, vol. 43, no. 6, pp. 1757–1766, November 1997.

[Wag02] D. Wagner, »A generalized birthday problem,« in *Advances in Cryptology—CRYPTO 2002*, ser. Lecture Notes in Computer Science, M. Yung, Ed., vol. 2442. Springer-Verlag, 2002, pp. 288–303.

[WC81] M. N. Wegman and J. L. Carter, »New hash functions and their use in authentication and set equality,« *Journal of Computer and System Sciences*, vol. 22, pp. 265–279, 1981.

[WHJÅ12] H. Wang, M. Hell, T. Johansson, and M. Ågren, »Improved key recovery attack on the BEAN stream cipher,« 2012, submitted.

[ZW09] H. Zhang and X. Wang, »Cryptanalysis of stream cipher Grain family,« Cryptology ePrint Archive, Report 2009/109, 2009, http://eprint.iacr.org/2009/109.