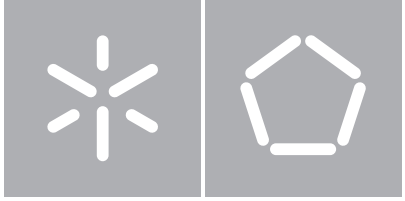
Universidade do Minho
Escola de Engenharia

Óscar Francisco Godinho Pereira

Towards a Fully Algebrisable Symmetric Cryptosystem

Setembro 2015



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Óscar Francisco Godinho Pereira

Towards a Fully Algebrisable Symmetric Cryptosystem

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho realizado sob orientação de

Professor Doutor José Manuel Valença

Setembro 2015

I dedicate this work to my parents, Pascoal and Teresa Pereira, the giants on whose shoulders I have stood. For everything, and so much more.

ACKNOWLEDGEMENTS

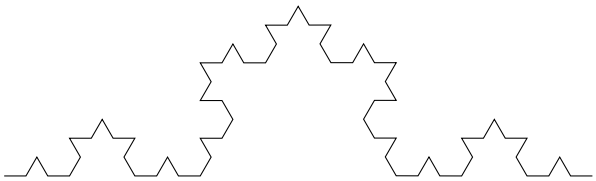
I would like to express my foremost gratitude to my supervisor, José Manuel Valença, for his guidance and availability, throughout the development of this work. His insights were very valuable, not only to help understanding topics that besides complex, were at times novel to me, but also to overcome the dead-ends that inevitably appear in research.

I would also like to thank Manuel Barbosa, for the help in procuring the generous grants that have funded this work.

I am also very grateful to my friends, who besides putting up with me—by no means a small feat—were also a frequent reminder that there is life beyond academia.

To my lab colleagues, not only for the insights, knowledge and humour q.b., but also for crafting an awesome work environment.

Last—but most assuredly *not* least—to my family, specially to my parents, to whom this work is dedicated. For all the reasons this meagre sheet of paper is too narrow to contain, thank you.



This work was supported by the following two grants.

From the SmartGrids project, through grant NORTE-01-0124-FEDER-000056, included on ON.2 IC&DT Programa Integrado “BEST CASE – Better Science Through Cooperative Advanced Synergetic Efforts”; Ref. BI-2014_BestCase_RL5.1_UMINHO.

From INESC TEC LA - HASLAB, through grant AE2015-0110.

ABSTRACT

Using components with simple algebraic descriptions, a round function for an iterated symmetric cipher is constructed, that seems to have some desirable properties. Two iterations of the round seem enough to obtain “full diffusion”, in the sense of having even one input bit change to affect on average half of the output bits. Furthermore the round also appears to have very high resistance against differential cryptanalysis.

That is the core result. In addition to that, a review of the state of the art is conducted, justifying why ciphers with simple algebraic descriptions are an area of research worth pursuing. Also, the components used in the round construction are of two separate kinds: linear and nonlinear. Each of those kinds is analysed in a dedicated chapter, exploring the properties that justify the way those components are used in the round function.

RESUMO

É descrita a construção de uma função de *round*, para uma cifra simétrica iterativa, utilizando componentes com uma descrição algébrica simples. Esta função tem propriedades que são desejáveis, nomeadamente iterá-la duas vezes parece ser suficiente para obter "difusão completa", neste sentido: mudar um bit no input, faz com que em média, mudem metade dos bits do output. Para além disto, a função aparenta ter uma muita alta resistência à criptanálise diferencial.

Esse é o resultado central. Adicionalmente, é também analisado o estado da arte, para justificar o porquê de cifras com descrições algébricas simples ser uma área de investigação com interesse. Finalmente, os componentes utilizados na construção da função de *round* enquadram-se em duas categorias: lineares e não-lineares. Cada um destes tipos é analisado num capítulo dedicado, explorando as propriedades que justificam o modo como esses componentes são utilizados na função de *round*.

CONTENTS

1 INTRODUCTION 1
1.1 Context . . . 1
1.2 Objectives and Motivation . . . 3
1.3 Outline of this document . . . 3
1.4 License Information . . . 4
2 PRELIMINARIES 6
2.1 Rings and Fields . . . 6
2.2 Galois fields . . . 7
2.3 Polynomials and bitstrings . . . 8
2.4 Linear Feedback Shift Registers . . . 8
2.5 Chinese Remainder Theorem . . . 9
2.6 Boolean functions and Cryptography . . . 10
2.7 Security Requirements . . . 11
3 STATE OF THE ART 12
3.1 Defining security, attempt #1 . . . 12
3.2 Defining security, attempt #2 . . . 15
3.3 Complexity as a future-proof strategy . . . 16
3.4 The Rijndael cipher . . . 17
3.5 Substitution-Permutation networks . . . 19
4 CIPHER DEVELOPMENT 21
4.1 Building Blocks . . . 21
4.1.1 Exclusive-OR . . . 21
4.1.2 CRT and Modulus . . . 22
4.1.3 Multiplication in R . . . 22
4.1.4 APNL functions . . . 22
4.1.5 Split/Join . . . 22
4.2 Round one . . . 23
4.3 Round two: dropping the *mod* . . . 25
4.4 Round three: the strange case of the "even-sized" APNL functions . . . 26
4.5 Further changes . . . 27
4.6 Final round . . . 27
4.7 Some remarks on implementation . . . 30
4.7.1 On the size of S-boxes . . . 31

4.8 Key schedule and number of rounds . . . 33
4.9 Conclusions . . . 34
5 LINEARITY 35
5.1 The Chinese Remainder Theorem . . . 35
5.2 Polynomial Ring Isomorphism . . . 37
5.2.1 The parity bit . . . 39
5.3 Diffusion analysis of the CRT matrix . . . 39
5.4 Constants in R and "full diffusion" . . . 41
5.5 Conclusions . . . 42
6 NON-LINEARITY 43
6.1 Differential Cryptanalysis . . . 43
6.2 The S-boxes . . . 44
6.2.1 The simplest case . . . 46
6.2.2 The *iterated* simplest case . . . 49
6.2.3 Differential Cryptanalysis on a full SP-network . . . 51
6.3 Applications to the cipher . . . 54
6.4 Conclusions . . . 56
7 OTHER ATTACKS 58
7.1 Linear Cryptanalysis . . . 58
7.2 Algebraic Cryptanalysis and SAT . . . 61
7.3 Conclusions . . . 62
8 CONCLUSIONS AND FUTURE WORK 63
8.1 Conclusions . . . 63
8.2 Prospect for future work . . . 64
A SAGE MATHEMATICS 65
A.1 Terminology and Python facts . . . 65
A.2 Basics . . . 65
A.3 The CRT matrix . . . 66
A.4 Simple differential cryptanalysis . . . 68
A.5 Simple (partial) linear cryptanalysis . . . 69
B CODES, MATRICES AND DIFFUSION 72
B.1 Coding theory . . . 72
B.2 Matrices and diffusion . . . 74
B.3 A basis for the $(x, f(x))$ codewords . . . 75
B.4 MDS matrices in GF(2) . . . 76
Bibliography 78
Acronyms 84

LIST OF FIGURES

Figure 1.1 The ROT13 alphabet permutation. 2
Figure 2.1 Example LFSR. 9
Figure 4.1 First sketch of round design. 24
Figure 4.2 Final round design. 28
Figure 6.1 Simple cipher. 47
Figure 6.2 Two round simple cipher. 49
Figure 6.3 Three round simple cipher. 51
Figure 6.4 Full SP-network. The blank rectangles represent diffusion. 53
Figure 7.1 Example target of linear cryptanalysis. 58
Figure 7.2 Two round example of linear cryptanalysis. 60

LIST OF TABLES

Table 1 Addition in $\mathbb{F}_2$. 8
Table 2 Multiplication in $\mathbb{F}_2$. 8
Table 3 The S-Box of Mini-AES. 44
Table 4 o-XOR distribution for fixed i-XOR 1011. 45
Table 5 Almost linear function, for i-XOR a. 48
Table 6 "Almost linear function", for i-XOR a. 49
Table 7 o-XOR distribution for fixed i-XOR 0010. 52
Table 8 o-XOR distribution for fixed i-XOR 1100. 53
Table 9 o-XOR distribution for fixed i-XOR 1110. 54

LIST OF CODE SNIPPETS

4.1 Code for the S-Boxes. 29
4.2 Encryption round. 30
4.3 S-Box sizes, with primes 17, 19, and 23. 32

5.1 Example of using SAGE to solve a simple CRT example. 36

A.1 Using SAGE to show the small moduli are irreducible (it outputs `True` 17 times). 67
A.2 Using SAGE to construct the CRT matrix. File `crtmatrix_test.sage`. 67
A.3 Using SAGE to test the CRT matrix. File `crtmatrix_test.sage`. 68
A.4 Using SAGE to do differential cryptanalysis. 70
A.5 Using SAGE to do compute linear expressions used for linear cryptanalysis. . . 71

1

INTRODUCTION

The whole point of cryptography is to solve problems. (Actually, that's the whole point of computers—something many people tend to forget.)

Bruce Schneier

IRAV IVQV IVPV

GAIVS IVLIVS CÆSAR

1.1 CONTEXT

This work and the resulting thesis concern an area of knowledge called *cryptology*. It is a vast field, that lies at a somewhat fuzzy intersection of mathematics, computer science, and (mainly software) engineering. But in spite of that vastness, it is surprisingly unknown, even to practitioners of some of the previously mentioned disciplines! It only gets worse when one considers the general public as a target audience[1]. Nevertheless, it is the author's opinion that that same general public deserves at least an overview in broad strokes—but in terms that it can be expected to understand—of how the resources that it trusts to the academic enterprise are being put to use. For while usually hidden behind a more or less thick layer of bureaucracy, it is the general public that is the ultimate patron of the majority of academic work. Such is then, the aim of these first few paragraphs.

Cryptology can be (roughly) divided into two subareas: *cryptography*, and *cryptanalysis*. Borrowing from (Anderson, 2008, §5.1)[2] *«cryptography refers to the science and art of designing ciphers; cryptanalysis to the science and art of breaking them; while cryptology, often shortened to just crypto, is the study of both»*. But what is a cipher? To a broad and diverse

1 On this last point the author wishes to offer the following piece of anecdotal evidence: in social settings, when he is asked about his field of study, if he answers with "cryptography", the reply is often something along the lines of "I'm sorry, did you say *cartography*?".

2 This is also the source of (the idea for) the second epigraph above, the relevance of which will become obvious momentarily.

(i.e. non-specialist) audience, a formal answer would be next to useless. An example however, will be far more illuminating.

After achieving a swift victory at the Battle of Zela (47 BC), Gaius Julius Cæsar is rumoured to have said *«Veni, vidi, vici»*, alluding to the short duration of the conflict: "I came, I saw, I conquered" (Wikipedia, e). If Cæsar wanted to relay this information back to Rome, but prevent his enemies from being able to read it (for example, by capturing the messenger), how should he proceed? One possible strategy would be to agree with the intended recipient a pre-specified *permutation* of the alphabet. For example, to write a message the enemy cannot read, you could replace the `A`'s with `N`'s, the `B`'s with `O`'s, and so on, according to a scheme known as `ROT13` (Wikipedia (c), Figure 1.1) which is a particular variant of a more general scheme known precisely as *Caeser cipher* (for simplicity it is assumed that all messages to be transmitted are written in capital letters, and without punctuation marks). Then the **plaintext** message "VENI VIDI VICI" is transformed into the **ciphertext** message "IRAV IVQV IVPV", and it is this last message that is transmitted. After receiving it, the recipient simply reverses the process to obtain the original message.

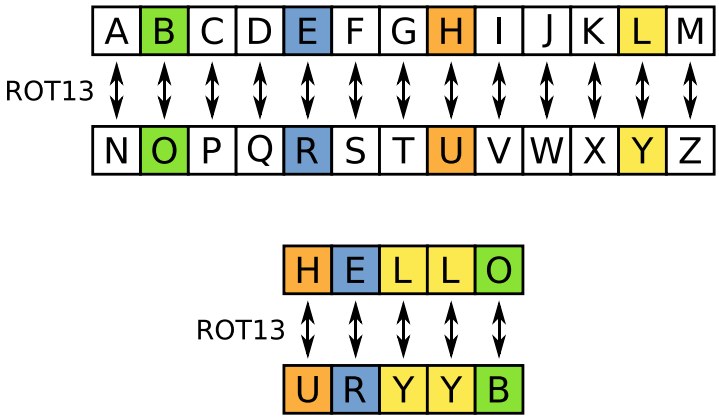


Figure 1.1.: The ROT13 alphabet permutation.

Thus a cipher is a set of instructions to produce messages that are unreadable to all, except those who are in possession of a crucial piece of information, called the *key*. In the case of the Cæser cipher, the key is the concrete alphabet permutation to be used. Also note that the *same* key is used by both the sender and the receiver of the secret message—these are called *symmetric* ciphers. Today's symmetric ciphers, while using far more advanced mathematics than simple permutations, still work in essentially the same framework (from the point of view of the cipher's *user*).

The goal of this work is to lay the first steps towards a cipher of this type (symmetric)—one that the author hopes will yield a stronger security guarantee than its predecessors.

1.2 OBJECTIVES AND MOTIVATION

The objective of this work is the specification and analysis of a symmetric block cipher that follows a simple algebraic model. The *hidden* objective is that that algebraic simplicity can be translated into a more formal analysis of security—yielding correspondingly stronger conclusions.

This work takes as a starting point the *Rijndael* cryptosystem (Daemen and Rijmen, 2002)[3]. Its design philosophy already reflects the idea of *simplicity* (*ibid.*, §5.2). In particular, its authors divide it into two types: simplicity of *specification*, and of *analysis*. The former means that only *«a limited number of operations»* are used, and that *«the operations by themselves can be easily explained»*. The latter means that we are able *«to demonstrate and understand in what way the cipher offers protection against known types of cryptanalysis»*. This provides the motivation: it is hoped that the envisaged algebraic simplicity will lead to both these requirements being fulfilled, with a particular emphasis on simplicity of analysis—where we hope to leverage the underlying mathematical structure (and theory) to attempt to prove better bounds for the different streaks of cryptanalytic techniques to be considered.

In more detail, symmetric ciphers are usually constructed combining (in an appropriate way) components that provide *confusion* and *diffusion* (rigorous definitions of these concepts, and of those that follow, will be given in the following chapters). The components to be used for confusion are a particular type of Boolean functions, called *«Almost Perfect NonLinear»* functions (APNL), while diffusion will be done using the *«Chinese Remainder Theorem»* (CRT). The purpose is then to study these components, and see if they are suitable building blocks for this class of ciphers—in particular, if they can yield better security guarantees than their predecessors.

A final remark is in order, concerning something that, while being of fundamental importance in the broader picture of block cipher development, will *not* be a prime goal of this work: efficiency. The reason is that studying it is not deemed feasible in the available time. And given that worrying about efficiency without ensuring security first is moot, it is the former that must be, for the time being, postponed.

1.3 OUTLINE OF THIS DOCUMENT

We start by rigorously stating the problem we attempt to solve, and giving some preliminary notions which are needed to understand the rest of the work (§2). Next, an overview of the state of the art is given (§3). This will pave the way for the complete description of our cipher (§4). As will be detailed therein, all modern symmetric ciphers are built combining

3 Pronounced "rhine dahl". This algorithm was chosen by the NIST in October of 2000, as the *Advanced Encryption Standard* (AES). For details of the selection process, see e.g. (*ibid.*, §1).

components that provide one of two properties: *diffusion* and *confusion*. The former are implemented through linear algebraic constructions, which will be addressed in §5, and the latter are implemented with nonlinear functions, explained in §6. In this latter chapter the round's resistance to differential cryptanalysis will also be studied. The following chapter (§7) describes two other powerful attacks against block cipher, viz. linear cryptanalysis and algebraic cryptanalysis—which constitute one of the main directions for future work. This future work, along with some concluding observations, are finally discussed in chapter 8.

REMARKS

The terms AES and Rijndael are, except when otherwise noted, used interchangeably. The same is true for "block cipher" and "symmetric cipher" (when there is no risk of confusion, both forms may be shortened to just "cipher"). Whenever there is ambiguity on whether a certain word should hyphenated or not (e.g. "plaintext" versus "plain-text"), the adopted convention will be to drop the hyphen(s)[4].

Throughout this document several excerpts of source code are shown. Often however, due to space and/or layout considerations, these are only uncommented snippets of a larger file. When such is the case, the caption of those snippets indicates the relevant filename (in `typewritten face`), under which the full content is available at `http://erroneousthoughts.org/mei`. Still about the code, when referring names of methods, the convention is to always end them in `()`, even though they might have mandatory arguments. This is done to aid the reader in distinguish those from other names (e.g. variable names, etc.).

The author wishes to apologise in advance to the more academically conservative a reader, for whom the liberal use of Wikipedia as a source, as done in this text, might be anathema. In his defense, the author points out that such use is done only either on non-technical matters (e.g. the introductory section), or when the information therein provided is corroborated by other sources. This latter scenario, however, begs the question of why bother with Wikipedia at all. The reason is that often it presents the information organised in such a fashion that makes it very suitable to introduce a novel idea or concept—thus giving a basis from where further (expository) writing can then be done. See for example, §B.2.

1.4 LICENSE INFORMATION

- Figure 1.1 is in the Public Domain:
  Wikipedia. Rot13. `http://en.wikipedia.org/wiki/ROT13`, c. Accessed: 2015-01-22

4 To give credit where it is due, this decision is largely due to Knuth's argument here: `http://www-cs-faculty.stanford.edu/~uno/email.html`.

1.4. License Information

- Figure 2.1 is licensed under a Creative Commons Attribution-Share Alike 3.0 Unported: https://commons.wikimedia.org/wiki/File:LFSR_Fibonacci_8_bits.png, Accessed: 2015-08-31

2

PRELIMINARIES

All of a sudden I had firm pegs on which I could hang other knowledge

Donald E. Knuth

This chapter describes the main algebraic structures used for this work, as well as the main notions from cryptography that are required. For further details about the algebraic notions, the reader is referred to Nicholson (2007).

A brief introduction of the relevant cryptographic concepts is also given in the final sections.

2.1 RINGS AND FIELDS

To define those structures, one must first define binary operations and groups. A *binary operation* $*$ on a set M is a mapping that to each ordered pair of elements of M, (a, b) assigns another element $a * b$ of M. Hereinafter, binary operations are assumed to be *closed*: i.e. *for all* $a, b \in M$, $a * b \in M$.

Definition 2.1.1 (Group)**.** *A set G together with a binary operation $*$ is said to be a* group *if the following four properties:*

1. *$*$ is associative: $a * (b * c) = (a * b) * c$, $\forall\, a, b, c \in G$.*
2. *G has an identity element of $*$. I.e. there is e such that $\forall\, g \in G$ we have $e * g = g$.*
3. *Every element in G has an inverse. I.e. $\forall\, g$ there exists $g^{-1} \in G$ such that $g^{-1}g = e$, where e is the identity.*

G is called an abelian group *if it verifies the following additional condition:*

4. *The operation $*$ is commutative.*

With *two* binary operations, we can define a *ring*. This construction is a generalisation of the structure of the integers, in which we have addition and multiplication.

Definition 2.1.2 (Ring)**.** *A set R together with two binary operations, $+$ and $\times$, is said to be a* ring *if the following properties hold:*

1. $(R, +)$ *is an abelian group.*
2. *The operation* $\times$ *is associative.*
3. *The operation* $\times$ *has an identity element (which belongs to* R*).*
4. *The distributive law holds:* $a \times (b + c) = a \times b + a \times c$ *and* $(b + c) \times a = b \times a + c \times a$*.*

R *is called an* abelian ring *if it verifies the following additional condition:*

5. *The operation* $\times$ *is commutative.*

When no ambiguity will result, the $\times$ can usually be omitted. The operations $+$ and $\times$ are usually called addition and multiplication respectively (although they can be very different operations from the usual addition and multiplication). For non-trivial rings, i.e. rings with more than one element, the identities of the two operations are different elements of R. By convention, the identity of $+$ is denoted by 0, and the identity of $\times$ by 1.

Definition 2.1.3 (Field)**.** *A ring* R *is said to be a* field *if all elements in* $R \setminus \{0\}$ *have multiplicative inverse, and if the operation* $\times$ *is commutative.*

This section ends with a note on notation: the operations, although usually referred to with nomenclature similar to integer addition and multiplication can, as already mentioned, be very different from their integer counterparts. Additionally, in the mathematical literature, the symbols used also vary wildly. Furthermore, the elements of the set to which those operations apply may even not be integers at all! One example of this latter case shall be seen shortly, where the elements of the ring are *polynomials*.

2.2 GALOIS FIELDS

When the usual integer addition and multiplication are performed modulo a prime p, those operations, together with the set $\{0, \ldots, p-1\}$ form an algebraic structure called the *Galois field* of size p, usually denoted by $\mathbb{F}_p$. One such field of particular importance for computer applications is the Galois field of dimension 2 (abbreviated to $\mathbb{F}_2$). It consists of a set with two elements, $\{0, 1\}$, and two binary operations, $+$ and $\times$. If the two elements are taken to coincide with the two eponymous binary digits, then the former operation coincides with binary addition without carry (or equivalently, the logical operation *exclusive-or*, or XOR), and the latter with the logical AND. For convenience we present their definitions below.

Beyond fields of prime size, there also exist Galois fields of size p^n, with p prime and n nonnegative integer. The elements of these latter fields however, are not integers. Such fields can be constructed, for example, with *polynomials*—see the next section. Before that however, we mention one result from modular arithmetic that will be required in chapter 4.

Theorem 2.2.1 (Fermat's (little) theorem)**.** *If* p *is a prime, then* $a^p \equiv a \pmod{p}$*, for all integers* a*. If* a *is relatively prime to* p*, then that can be simplified to* $a^{p-1} \equiv 1 \pmod{p}$*.*

+	0	1
0	0	1
1	1	0

Table 1.: Addition in $\mathbb{F}_2$.

×	0	1
0	0	0
1	0	1

Table 2.: Multiplication in $\mathbb{F}_2$.

2.3 POLYNOMIALS AND BITSTRINGS

A **polynomial of degree m over a field $\mathbb{F}$** is defined as an expression of the form:

$$p_0 + p_1 x + \times p_{m-1} x^{m-1} + p_m x^m \tag{1}$$

where $p_m \neq 0$, each $p_i \in \mathbb{F}$, and x is the indeterminate[1]. Note that a polynomial of degree m has $m+1$ coefficients (including the zeros). The set of all such polynomials constitutes a ring, denoted $\mathbb{F}[x]$. Because of this, addition and multiplication are also defined for polynomials. A polynomial is said to be *irreducible* if it can only be factorised as a product of itself and 1 (multiplicative identity). The integer division algorithm also applies to polynomials: for any $p(x)$ and $d(x)$, the exist $q(x)$ and $r(x)$ such that $p(x) = d(x)q(x) + r(x)$ (with $deg(r) < deg(d)$). This means *modular arithmetic* can also be done on polynomials; in fact, for a given $p(x)$, the set of all polynomials modulo $p(x)$, denoted $\mathbb{F}[x]/\langle p(x) \rangle$ is a ring; if $p(x)$ is irreducible it is a field.

In $\mathbb{F}_2$, a polynomial can be naturally identified with a bitstring. Following the way SAGE outputs an array of polynomial coefficients (independent term at index 0, i.e. "leftmost", like done above), we use the following convention when thinking of polynomials as bitstrings: the independent term is *the **leftmost** bit*; and conversely, the coefficient of highest the degree monomial is *the **rightmost** bit*. If the bitstring (or array) is depicted vertically, the independent term is on top.

2.4 LINEAR FEEDBACK SHIFT REGISTERS

Consider the construction depicted in Figure 2.1. That is an LFSR: an array of bits, where on each iteration, all bits are right-shifted by 1, and the leftmost bit becomes the XOR of the bits in pre-specified positions of the original array (i.e. before the right-shift). They are a particular case of shift registers, called "linear" because the XOR operation is linear. They have been used not only in cryptography, but also in coding theory. For a more thorough description (and further references) see (Schneier, 1996, §16.2).

1 The reader may be used to thinking of x as the "variable", and indeed it can be thought of as one (and we shall do so at times; cf. for instance the ending of §5.2). But that is not necessary: polynomials can be constructed as mathematical objects which are never evaluated (i.e. x is never "given a value").

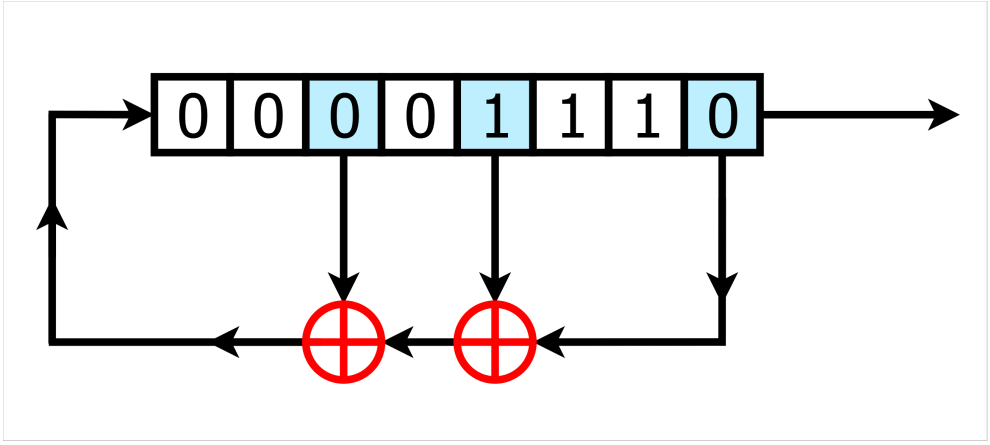


Figure 2.1.: Example LFSR.

2.5 CHINESE REMAINDER THEOREM

The Chinese Remainder Theorem (CRT) will be described in greater detail in chapter 5. Here we shall outline the essential to be able to understand the cipher's round description, given in chapter 4.

Consider the polynomial ring $\mathbb{F}_2[x]$, and the polynomial $x^{257}+1$. In that ring it factorises as a product of sixteen polynomials of degree 16 and one polynomial of degree 1; see Equation 4 in §5.1. Thus we can define a "big ring" $R[x]$ (or just R, when meaning is clear from context) as $\mathbb{F}_2[x]/\langle x^{257}+1\rangle$, and seventeen "small rings" $r_i[x]$ (or just r_i, idem.), $0 \geq i \geq 16$, as $\mathbb{F}_2[x]/\langle m_i\rangle$, where $m_0 = x+1$, $m_1 = x^{16}+x^{12}+x^{11}+x^8+x^5+x^4+1$, and so on and so forth, following the ordering suggested by Equation 4. Thus r_0 is the ring that contains only the two polynomials of degree 0 of $\mathbb{F}_2$, viz. 0 and 1; the other r_i contain all polynomials of degree smaller than 16. With the CRT we can construct an isomorphism between the big ring R, and the direct product ring $r[x]$ (or just r, idem.) defined by $\prod_{i=0,\ldots,16} r_i$. This means that to *any* polynomial in R there corresponds one (and only one) tuple of seventeen polynomials in the different r_i, and vice-versa.

Polynomials in the big ring R (of degree at most 256) will be designated by $P_1, P_2, \ldots$. The corresponding element in r of P_1 is the 17-tuple $(p_0^1, p_1^1, \ldots, p_{16}^1)$. Basically the index of the big ring "becomes a superscript" in the small rings. When considering just *one* element of R, the subscript may be dropped, i.e. the element will just be denoted by P. The corresponding element in r is $(p_0, p_1, \ldots, p_{16})$.

Coefficients of monomials will be referred to by a second subscript, e.g. $P_{2,0}$ refers to the independent term of polynomial P_2, and $p_{10,5}^1$ refers to the coefficient of the monomial of degree 5 of the polynomial in small ring r_{10}, of the tuple in product ring r that corresponds to element P_1 in R. If the element of the big ring is denoted without subscripts, then we add them to refer to the coefficients of that element; it will be clear from context what the subscript is

being used for—i.e. if P_1 is one element of R or the coefficient of the monomial of degree 1 of $P \in R$. For the corresponding element in r of polynomial $P \in R$ we just add a second subscript, but without any superscripts (e.g. $p_{1,0}$ is the independent term of the polynomial in r_1.

As per the notational convention stated above, a given polynomial in R can be seen as a 257 bitstring; but concatenating the bitstrings of the polynomials of r yields another bitstring (again of length 257). When doing so, the polynomials in the small rings will again be identified with bitstrings having as leftmost bit the independent term, and furthermore those strings will be concatenated from left to right, following the ordering of the tuple at the beginning of the paragraph: first the bitstring of the polynomial in r_0, then the one of in r_1, and so on until r_{16}.

Particular bits in a bitstring can be identified by a coefficient of its corresponding polynomial; for example, the *rightmost* bit of a 257 bitstring may be referred to as P_{256}.

2.6 BOOLEAN FUNCTIONS AND CRYPTOGRAPHY

Let $X = \{0,1\}^n$ and $Y = \{0,1\}^m$, i.e. the sets of all binary strings of length n and m respectively. A boolean function is a function from X to Y, i.e. $f : X \to Y$. Consider the set of *all* such functions; that set has size $|Y|^{|X|}$. If f is such that it cannot be "efficiently" distinguished from a function chosen at random from $|Y|^{|X|}$, then we say that f is a *Pseudorandom function*. If $n = m$, then it is a *Pseudorandom permutation*. An algorithm is said to be "efficient" if it is a practical algorithm; this is sometimes stated by saying the algorithm runs in polynomial time. These notions can be defined more rigorously, but for the purposes of the current work, the intuition of the concept that this paragraph tries to convey is sufficient.

A block cipher is a *keyed* pseudorandom permutation; "keyed" means that it is actually a set of pseudorandom permutations, and each choice of the key determines one such permutation. Because block ciphers often needed to encrypt a variable amount data, different from n bits, they are often used together with a *mode of operation*. These are basically ways to repeatedly use the cipher to encrypt the required data, while maintaining the cipher's security properties. These modes range from simply using the cipher in parallel until all the data has been enciphered (*Electronic Code Book)*, to chaining the previous ciphered block the current block (*Cipher Block Chaining*). For more details, see (Katz and Lindell, 2007, §3.6.4).

Block ciphers are a particular case of *symmetric cryptosystems*, because the same key is used for encryption and decryption. They are called "block" ciphers because both processes are done "block-wise". In contrast with, for example, stream ciphers, where it is done bit-wise (i.e. with blocks of size 1).

There several several types of attacks against block ciphers. The most (conceptually!) simple one is a *ciphertext only* attack: here the cryptanalyst is given only ciphertext messages, and has to deduce either their corresponding plaintexts, or the encryption key. Two other types of attacks are *known plaintext* and *chosen plaintext*:

- In known plaintext, the cryptanalyst is given pairs of (plaintext, ciphertext) messages, and has to deduce the key. An example is linear cryptanalysis.
- In chosen plaintext, the attacker can *request* pairs (plaintext, ciphertext), and his choice of the next pair might be influenced by computations he made on earlier requested pairs. An example of this is differential cryptanalysis.

There are other attacks, but these are the most relevant for this work. For more information on those other attacks, see e.g. (Schneier, 1996, §1.1).

2.7 Security Requirements

Block ciphers are primarily designed to provide *secrecy*. That is, to transform a given plaintext message in such a way that it its original structure—and thus its meaning—is removed, and furthermore cannot be recovered without knowledge of the encryption key. However, block ciphers can also be used as building blocks in other constructions, and these might have security goals other than secrecy. A non-exhaustive list of examples is the following:

- *Authentication*: here the purpose is enable the receiver of a message to verify its origin. In other words, the sender should not be able to send a message as someone else.
- *Integrity*: the receiver should be able to verify that the message has not been modified since it was produced or sent. Or if such modifications have occurred, the receiver must be able to detect that that was the case.
- *Non-repudiation*: A sender must not be able to send a given message, and later deny having done so.

3

STATE OF THE ART

> *There is a remarkably close parallel between the problems of the physicist and those of the cryptographer. The system on which a message is enciphered corresponds to the laws of the universe, the intercepted messages to the evidence available, the keys for a day or a message to important constants which have to be determined. The correspondence is very close, but the subject matter of cryptography is very easily dealt with by discrete machinery, physics not so easily.*
>
> Alan M. Turing

> *I conclude that there are two ways of constructing a software design: One way is to make it so simple that there are* obviously *no deficiencies and the other way is to make it so complicated that there are no* obvious *deficiencies. The first method is far more difficult. It demands the same skill, devotion, insight, and even inspiration as the discovery of the simple physical laws which underlie the complex phenomena of nature.*
>
> C. A. R. Hoare

For a project of this nature, researching the state of the art encompasses several topics. The first one is security: what does it mean for a block cipher to be *secure?*—this is the topic of sections 3.1 and 3.2. Next, as mentioned previously, this work's starting point is the design of Rijndael, so a brief description of this cipher is also in order. After this, a brief description is given of how the primitives of confusion and diffusion will be lay out in the cipher to be developed.

3.1 defining security, attempt #1

Alan Turing may have been correct when he quipped that computers are naturally more suited to cryptography than to physics, but despite that suitability, all the "discrete machinery" in the world will not help us in providing a sensible definition of what it means for a cipher to

be secure. The main problem is that security is *relative*: what is secure in one usage scenario might be totally insecure in another one. Furthermore, "security" can have different meanings depending on the context: it can mean privacy/secrecy, or integrity, or authentication, or non-repudiation, etc. Providing one of these can hinder the effort to provide another. For example secrecy requires integrity (see below), but combining the two securely leads to some non-obvious concerns (Bellare and Namprempre, 2000), that simply do not exist with integrity alone. Another example is combining non-repudiation with anonymity, and so on. This means that one has to think about security in terms of **trade-offs**. To make matters even worse, *«[b]lock ciphers are very useful building blocks. They tend to get abused in every imaginable way.»* (Ferguson and Schneier, 2003, §4.2). All of this makes attempting to provide a "one size fits all" definition a very challenging task.

Nevertheless, the goal of a block cipher, when seen as a cryptographic primitive of its own right (as opposed to when seen as a building block of a larger construction), is to conceal a given message, i.e. to provide **secrecy**; so let us put the focus there. The first attempt to study the security of block ciphers stemmed from Claude Shannon's groundbreaking work in Information Theory, which he then applied to problem of secret communication (Shannon, 1949). He started by carefully delimiting the problem, by noting that while secrecy systems can generally be classified as being of one of three types, his concern would be only one of those types. More concretely, secrecy systems can be *concealment systems*, that hide the existence of the message (think "invisible ink"), or what Shannon dubbed *privacy systems*, *«for example speech inversion, in which special equipment is required to recover the message»* (*ibid.*); or, lastly, they can be *«"true" secrecy systems where the meaning of the message is concealed by cipher, code, etc., although its existence is not hidden, and the enemy is assumed to have any special equipment necessary to intercept and record the transmitted signal»* (*ibid.*). Shannon's focus was on systems of the third type, which are what we today call symmetric cryptosystems[1].

He then proceeded to determine the requirements for a secrecy system that could not be broken even by an adversary with *unlimited* resources. That is, if given a (ciphertext) message enciphered with one such system, not even such a powerful adversary would be able to recover the original (plaintext) message—or *any* part of the original message, no matter how small. Shannon dubbed this *perfect secrecy*, and it essentially means that an adversary—*any* adversary, no matter how powerful—observing the ciphertext will not be able to extract *any* information whatsoever from it. Another way to phrase this is to say the ciphertext does not *leak* any information about the plaintext. This means the adversary is left with only what information he already knew, *a priori*, about the plaintext. It is possible to construct cryptosystems which possess this property; however Shannon also proved that

1 The reader may have noticed that the third definition also encompasses **a**symmetric cryptosystems. But these had not yet been developed at the time of Shannon's work; and furthermore, they are not this work's focus—we just state this for completeness.

they inherently require keys that are (at least) as long as the plaintext messages (*ibid.*, §10), which makes perfect secrecy an inadequate definition for most real-world scenarios: indeed if one has the capability of transmitting securely keys that are as large as the plaintext, one would usually be justified in ditching the keys altogether and transmitting only the plaintext messages!

It should be stressed, if only in passing, that even though for the purposes of the present discussion we have restricted the meaning of information security to information secrecy, it is well settled that when ***using*** block ciphers, it is impossible to guarantee secrecy if the usage mode does not also provide *integrity*. For a general description of what can go wrong see for example Bellovin (1996); padding oracle attacks (Vaudenay, 2002) are another example. This is true *even if the block cipher in question provides perfect secrecy*: for example, with the *One-Time Pad*[2], although it is perfectly secret, not only is the ciphertext *malleable*, but also one can always claim the key had any value that makes the ciphertext decipher to any plaintext one wants (Anderson, 2008, §5.2.2). Alas there is perfect secrecy, but no message integrity. Security is *relative*!

So total, "absolute" security does not seem possible, and even perfect secrecy seems impractical. This latter problem can be ameliorated by relaxing the definitions. First we stop considering adversaries with unlimited power, and restrict the analysis to adversaries with a computational power that is "realistic in practice". Next, we weaken the definition of perfect secrecy, so that we no longer demand the ciphertext leaks *no information* about the plaintext, but rather that the information that is leaked is not easily retrievable by the weaker adversary we are now considering. The more formal expression of these ideas is actually posterior to Shannon's work, and thus is not needed here. Nevertheless they encompass what he meant by "practical secrecy"[3], to which we devote the rest of this section.

For any general symmetric cipher, after enough ciphertext has been intercept, it is possible—at least in theory—to uniquely solve (i.e. to determine an unique key). This happens after reaching the *unicity distance*. However, even after this point has been reached, the amount of work required to determine this solution for different ciphers can vary greatly. The two most oft-cited principles Shannon put forth to maximise this amount of work are *confusion* and *diffusion* (we will get to these shortly), but there exists a third, very important one: that all (or the majority) of the bits of the key should be used to encrypt all of the message. This prevents attacks that break the keyspace into smaller chunks (thus reducing the overall effort to determine the key).

To make the cryptanalyst's job as hard as possible, Shannon suggested two approaches, which he called *diffusion* and *confusion* (Shannon, 1949, §23)[4]. Diffusion consists of having

2 In Shannon (1949), a special form of the One-Time Pad, that applies to letters of the alphabet (instead of bits) is designated by the "Vernam system".

3 Shannon, *op. cit.*, §21 onwards.

4 This is not the *whole* truth: he also mentions that one could resort to *ideal secrecy systems* (*loc. cit.*), although these are seldom practical (*ibid.*, §18).

the *«statistical structure of M which leads to its redundancy [is] "dissipated" into long range statistics»*. In other words, the patterns present in the plaintext (due to redundancy of the message) should be as scattered as possible, in order to increase the amount of traffic that has to be intercepted before said patterns become noticeable. In modern parlance this means that changing one bit of the input should change as many bits of the output as possible.

Confusion consists in *«mak[ing] the relation between the simple statistics of E and the simple description of K a very complex and involved one»*. Shannon gives one simple example where this is *not* the case: the (simple) substitution cipher. Indeed, here the key consists in a particular permutation of the alphabet, and with a few ciphertexts one can already (by frequency count) severely constrain the set of possible keys. The current way to think about confusion is as trying to obscure the relationship between the plaintext, the ciphertext, and the key. This makes attacks that exploit the relationship between them (e.g. differential and linear cryptanalysis) much harder to carry out.

3.2 Defining security, attempt #2

In the current state of the art, the most general definition of block cipher security is that which is based on the notion of computational indistinguishability (Katz and Lindell, 2007, §3.2.1). But this is more helpful when one tries to give a *security reduction*[5] for some construction that uses the block cipher, rather than when one is tasked with designing it. We can build a "provably secure" block cipher, taking as a starting point for example, a "secure" stream cipher (Anderson and Biham, 1996). But then the problem remains of how to define what security means when applied to stream ciphers… And furthermore, this "reductionist-style" reasoning is not useful when the cipher is to be built from lower level components, as is the case in this work. Lastly, the authors of Rijndael point out that the proofs of security that have been presented for block ciphers rely *«on (often implicit) assumptions that make [the proof] irrelevant in the real world»*—although they don't give any references or examples (Daemen and Rijmen, 2002, §5.5.3).

So provable security will not do. What other alternatives are there? Daemen and Rijman offer two security criteria—*K-secure* and *hermetic*—that are so abstract that they themselves admit that *«we cannot prove that Rijndael satisfies [them]»*, even before stating the respective definitions (*ibid.*, §5.5.1). These seem to be a sort *a posteriori* criteria—given a particular block cipher design, you study it and see if the definitions are verified. But again, this is not useful as a guide for design choices.

The strategy that remains is to guide the design by increasing resistance to attacks. What do we mean by "attack"? *Grosso modo*, it means a way of obtaining some piece of information that the attacker should not be able to access. For example, all ciphers where the size of

5 Also known as *proof of security*.

the key is much smaller than the size of the plaintext are vulnerable to an attack called *brute force*. This consists of trying all the possible keys. If the attacker has a (plaintext, ciphertext) pair, then (at least) one key is going to match and, with very high probability it will be unique—which means it is the correct key. But even if the attacker only has ciphertext messages, only very few keys will yield intelligible plaintext messages—which is already a substantial reduction of the number of possible keys. The only way to thwart this attack (other than using keys as large as the plaintext) is to use a key size that, while smaller than the plaintext size, is still large enough to make trying all possible keys infeasible. Then the cipher will be secure against attacks by brute force.

More generally, if a new way is discovered of obtaining some information, about the key and/or the plaintext, that takes less "effort" than brute force, then that will constitute a new attack (such is the case, for example, with differential cryptanalysis). Thus, we say that a cipher is *secure* when either no such attacks exist, or when they are, for the cipher in question, infeasible to carry out.

This will be the course of action to be pursued. But it should be noted that, while here only resistance against cryptanalytic attacks will be tested, this is, in general, not sufficient. To give an example, in (Ferguson and Schneier, 2003, §4.2) it is argued that if a cipher that is vulnerable to related key attacks is used in the Davies-Meyer construction, the resulting hash function will be insecure (also cf. the ending of the first paragraph above).

3.3 Complexity as a future-proof strategy

Developing ciphers is a long process. After being proposed it has to be attacked, and it will only be trusted if it successfully resists those attacks (or if it is only vulnerable to attacks that are not practical, by a *large* margin). Thus, when developing a cipher, the designers want to be as confident as possible that it will resist any novel attack that might appear in the future. This is often used as a justification to introduce additional complexity. Moreover, the lack of such complexity motivated the following criticism of AES: *«It is possible to write an AES encryption as a relatively simple closed algebraic formula [...] if anyone can ever solve [it], then AES will be broken»*, followed by an anecdote of how a hypothetical mathematics graduate student might break AES when searching for something to do in an afternoon of boredom (*ibid.*, §4.5.2).

Besides being essentially a defense of security by (design) obscurity, such a criticism misses an important point: with a simple design, a cipher that withstands years of attack will become trusted, and everyone will understand the rationale behind that trust. With a complicated one, it will take more time to reach the same level of trust, but there will still exist a greater possibility that some weakness has escaped the designers' eyes, precisely because it was hidden in the complexity of the design! Furthermore, such weaknesses, if they do exist, will require

more time for being rooted out. And lastly, ease of implementation matters. As Whitfield Diffie wrote in the foreword to Schneier (1996): *«all the great cryptographic papers in the world do not protect a single bit of traffic»*. The more complicated the design, the more likely it is that implementation mistakes will occur—and then, no amount of theoretical results will be of any help in achieving security.

Although not (directly) related to cipher design, we describe, as but one example of what can go wrong with the complexity mindset, Donald Knuth's attempt to create *«a fantastically good [pseudorandom number] generator»*, using an algorithm, dubbed *Algorithm K*, so complicated *«that a person reading a listing of it without explanatory comments wouldn't know what the program was doing.»* (Knuth, 1997, §3.1). The result, however, was failure (emphasis on the original):

> Considering all the contortions of Algorithm K, doesn't it seem plausible that it should produce almost an infinite supply of unbelievably random numbers? No! In fact, when this algorithm was first put onto a computer, it almost immediately converged to the 10-digit value 6065038420, which—by extraordinary coincidence—is transformed into itself by the algorithm (see Table 1). With another starting number, the sequence began to repeat after 7401 values, in a cyclic period of length 3178.
>
> The moral of this story is that *random numbers should not be generated with a method chosen at random.* Some theory should be used.

Considering the critical dependency of cryptographic primitives on the underlying source of pseudorandom numbers[6], it seems reasonable that the above cautionary tale should apply to the development of said primitives as well.

3.4 The Rijndael cipher

As previously mentioned, Rijndael strives for simplicity. It consists of a key-independent round transformation, applied iteratively, with intermediate key additions in between. Its pseudocode description is given below. For further details the reader is referred to (Daemen and Rijmen, 2002, §3).

The key to understand how it works is to understand its *State*. Its 16 bytes are arranged top to bottom, left to right, 4 bytes per columns. Thus byte 0 is on the top left corner, and byte 15 on the bottom right corner. The key is arranged similarly (although with a possibly different number of columns). These form the `State` and `CipherKey` variables. The intermediate keys are just XOR'ed with the intermediate round output.

The number of rounds varies according to the size of the key, so as to maximise resistance to cryptanalysis (the more rounds, the more resistant—but also the more slower—the cipher

6 See, e.g. (Goldreich, 2006, §1.1.2).

becomes). Each round is composed by three operations, described in the next three paragraphs.

SubBytes is the operation that introduces non-linearity. It is an SBox that is applied to all the bytes of the state (note that it is the *same* SBox for all bytes). The non-linear function over $GF(2^8)$ implemented by the SBox is $f : x \to y = x^{-1}$ (0 maps onto itself). To prevent interpolation attacks however, the SBox actually implements f followed by an affine transformation (which does not affect its non-linearity properties).

ShiftRows is a byte transposition that acts on the *rows* of the state (remember that the state is in essence a byte matrix with 4 rows and a variable number of columns). In the version of Rijndael that was standardised as the AES, with a fixed block length of 16 bytes, the first row shifts 0 bytes to the left, the first 1 byte, the second 2; and the third 3.

MixColumns is a permutation that acts on the state column by column. It interprets each column as a polynomial over $GF(2^8)$, and multiplies it with a fixed polynomial (modulo an irreducible polynomial over the same field).

```
Rijndael(State, CipherKey)
{
        KeyExpansion(CipherKey, ExpandedKey);
        AddRoundKey(State, ExpandedKey[0]);
        for(i=1; i < N_r; i++) {
                Round(State);
                AddRoundKey(State, ExpandedKey[i]);
        }
        FinalRound(State);
        AddRoundKey(State, ExpandedKey[i]);
}

Round(State, ExpandedKey[i])
{
        SubBytes(State);
        ShiftRows(State);
        MixColumns(State);
}

FinalRound(State, ExpandedKey[i])
{
        SubBytes(State);
        ShiftRows(State);
}
```

3.5 SUBSTITUTION-PERMUTATION NETWORKS

In Shannon's aforementioned article, before discussing (among other things) the principles of confusion and diffusion, he stated that *«[i]t is difficult to define the pertinent ideas involved [in making cryptanalysis as hard as possible] with sufficient precision to obtain results in the form of mathematical theorems, but it is believed that the conclusions, in the form of general principles, are correct»* (Shannon, 1949, §21). Thus, there is more than one way of designing a cipher that accommodates both principles. Rijndael's structure is one such way, but the most common using what is known as *Substitution-Permutation network* (or SP-network, for short). This basically consists of layer providing confusion (usually through S-Boxes), followed by another layer providing diffusion, usually through permutation boxes (or P-boxes, for short). This is the scheme that we shall follow, although diffusion will be provided by a

construction different than P-boxes. The details are given in §4; also, see Figure 6.4 for a schematic of a typical SP-network design.

4

CIPHER DEVELOPMENT

> *Il [le chiffre] faut qu'il n'exige pas le secret, et qu'il puisse sans inconvénient tomber entre les mains de l'ennemi.*
>
> Auguste Kerckhoffs

One of the motivations of this work is to study the cryptographic properties of the CRT isomorphism. They fell short of what was expected, but nevertheless led to a design that seems promising. This chapter describes the design iterations that resulted in the current round function—which is then described. Finally other aspects of cipher development are described, namely the key schedule and the number of rounds, which in the present work were only briefly touched.

4.1 BUILDING BLOCKS

The goal is to construct a symmetric cipher with a simple algebraic description, for which we had available the following repertoire of constructions: polynomial modular arithmetic, exclusive-OR, multiplication in the big ring, CRT, APNL and split/join. Each of these is described in the following subsections.

4.1.1 *Exclusive-OR*

This is one of the most used operations in cryptography, it corresponds to addition modulo 2 or, equivalently, addition in $\mathbb{F}_2$.

The epigraph is the original French version of what is today known as *Kerckhoffs' principle*: *«The design of a system should not require secrecy, and compromise of the system should not inconvenience the correspondents»*. It is usually paraphrased as "the security of the system must rest *solely* in the key (and not in the algorithm's design)".

4.1.2 *CRT and Modulus*

As mentioned, there exists an isomorphism between the polynomial ring $R[x]/\langle x^{257}+1\rangle$ (denoted by R), and the product ring of the 17 rings that comprise the factorisation of the modulus (r_i) (Eq 4). The CRT operation corresponds to taking a 17-tuple of polynomials in the r_i, and writing it as polynomial in R. The modulus (*mod*) operation corresponds to the inverse, i.e. taking the polynomial in R and writing it as a 17-tuple. The reason it is called "modulus" is because dividing the polynomial in R by the modulus of the different r_i yields the different elements of the 17-tuple; see chapter 5 for details. Additional details on the underlying mathematics of the CRT can be found in any standard Abstract Algebra textbook, e.g. Nicholson (2007).

4.1.3 *Multiplication in R*

Let P be a polynomial in R, and consider the vector of its binary coefficients, $P_0P_1\cdots P_{256}$. Multiplying P by x means moving all bits in the vector one position to the right (right-shift), and P_{256} to the left-most position. More generally, to multiply P with x^m one has to right-shift all bits m positions, with round-robin (i.e. "looping around"). If the polynomial has more than one monomial, the multiplication can be done monomial by monomial.

Note that the described operation can be implemented efficiently (specially in hardware) using an LFSR.

4.1.4 *APNL functions*

Almost Perfect Nonlinear Functions (APNL functions) are designed to make differential cryptanalysis as hard as possible. These are the Boolean functions, with the same number of input and output bits, used for S-Boxes. There exists a very advanced theory about them; see Pott and Edel (2009); Weng et al. (2013); Edel and Pott (2009); Berger and Canteaut (2006). In addition, they are further described in chapter 6.

4.1.5 *Split/Join*

The *join* operator takes a 17-tuple of polynomials in the small rings, and returns the polynomial in the big ring, that corresponds to the concatenation of those polynomials. It is defined as the inner product of that tuple (seen as a vector) with

$$J = (x^0, x(x^{16})^0, x(x^{16})^1, x(x^{16})^2, \ldots, x(x^{16})^{15})$$

Thus we have

$$\begin{aligned}&(p_{0,0}, p_{1,0} + p_{1,1}x + \cdots + p_{1,15}x^{15}, \ldots, p_{16,0} + p_{16,1}x + \cdots + p_{16,15}x^{15})\\&\cdot (x^0, x(x^{16})^0, x(x^{16})^1, x(x^{16})^2, \ldots, x(x^{16})^{15})\\&= P_0 + P_1x + \cdots + P_{16}x^{16} + \cdots + P_{241}x^{241} + \cdots + P_{256}x^{256}\end{aligned}$$

Conversely, the *split* operator takes a polynomial in the big ring, and returns the elements of the 17-tuple, from left to right. Algorithm:

Let $P = P_0 + P_1x + \cdots + P_{16}x^{16} + \cdots + P_{241}x^{241} + \cdots + P_{256}x^{256}$ and $D = x$.

1. Polynomial division: let Q and R be such that $P = Q \times D + R$
2. Push R to the leftmost available position in the 17-tuple.
3. Let $P = Q$ and $D = x^{16}$.
4. Go to step 1 and repeat until P has no nonzero monomials left.
5. Return the 17-tuple.

4.2 ROUND ONE

The first round function that was sketched used mainly the CRT and APNL constructions. It is depicted in Figure 4.1. The idea was to treat the received plaintext block as a 17-tuple of the polynomials in the small rings r_i, put that through 16 S-Boxes (each implementing an APNL function from 16 to 16 bits), take the result (also a 17-tuple of polynomials in the small rings) and run it through the CRT matrix, and multiply the resulting polynomial in R with the round key—seen here also as a polynomial in the big ring R. The S-Boxes would implement APNL functions, at the time yet to be chosen.

Note that what was just described is a round function that maps 257 input to 257 output bits. This was something to avoid if possible—using an input length that is not a power of of 2 would, at the very least, make it harder to write efficient implementations. The most obvious strategy seemed to be tweaking the cipher to the nearest power of 2 (256), by somehow "getting rid" of the $p_{0,0}$ bit—which is why its arrow has a question mark. But the question mark is there because of another issue: if getting rid of that extra bit turned out not to be possible, than what should be done with it? An APNL function for one bit only is overkill, but it clearly cannot go directly to the CRT either, sidestepping the confusion layer.

Another problem that this structure has is that R is a *ring*, but not a field, which means that there exist elements for which there is no multiplicative inverse, which constrains the choice of round keys. But this is not as problematic as it seems. The reason is that R is isomorphic to the 17-tuple, and the property of having an inverse is preserved by isomorphisms. Now

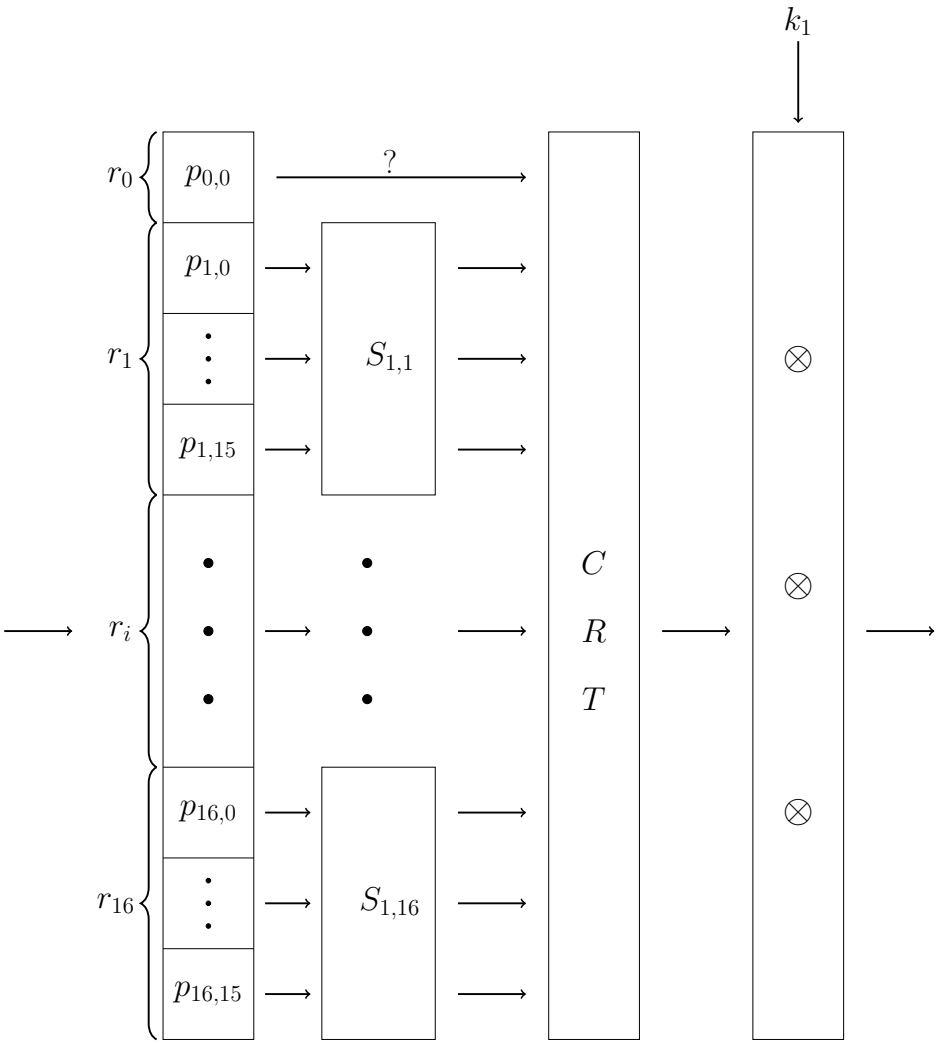


Figure 4.1.: First sketch of round design.

each of the small rings r_i is actually a field (the moduli are all irreducible, cf. A.1), and fields have the property that the *only* element without multiplicative inverse is 0 (i.e. the additive identity). This means we can calculate the proportion of elements with inverse (possible keys) to the total number of elements (note that it must be $p_{0,0} = 1$, because the only other element of r_0 is 0):

$$\frac{(2^{16}-1)^{16}}{2^{257}} \approx 0.4999$$

This seems really bad news—the keyspace just got reduced to *half*!—but even so, there is still enough entropy to make the cipher secure: $\log((2^{16}-1)^{16})/\log(2) = 255.9996$ i.e. almost 256 bits of entropy.[1]

The most serious flaw, however, is related with iterating this round structure. For the second round, we need to take the output of the first (a polynomial in R), and convert it to a 17-tuple in the r_i, through the *mod* operation. Which is problematic because *mod* is the inverse operation of the CRT: applying one after the other, with only a linear operation in between (the multiplication by the round key), is essentially the same thing as applying only the linear transformation.

4.3 ROUND TWO: DROPPING THE *mod*

The solution to the last objection of the previous section was to drop the *mod* operation, and even though the output of the first round is a polynomial in R, the second round would treat the corresponding bitstring as a 17-tuple, and put it through the S-Boxes, etc. Note that the construction thus obtained is *reversible*: during decryption, when going from round $j+1$ to round j, we do the opposite i.e. we take the bitstring corresponding to the 17-tuple and treat it as representing a polynomial in R. Strictly speaking, we did not "drop the *mod*" operator; it is still required for *decryption*, to reverse the CRT operation. But for encryption it is no longer necessary.

The "change of perspective" from seeing a bitstring as representing an element of R to an element in the product ring can be described more formally; indeed it corresponds to the *split* operator. The inverse operation corresponds to the *join* operator.

This seemed a more promising design, but there was still the matter of the element of r_0. There were a few ideas. One was to make that bit a "parity bit". By this it is not meant that the input should be 256 bits, and the algorithm then computes the parity of that bitstring, and feeds it to the round—this is an increase in redundancy, which in some scenarios can actually make cryptanalysis easier (Bellovin, 1996). Rather, the CRT isomorphism has the property that given a 17-tuple $(p_0, \ldots, p_{16})$, and its corresponding polynomial in R, P_1, then p_0 is the

1 For a slightly amusing, if very informal, justification of why that entropy more than enough, see this discussion thread: https://security.stackexchange.com/questions/14068/why-most-people-use-256-bit-encryption-instead-of-128-bit.

parity of the bits of P_1 (see §5.2.1). Another option was to "shift by one" the 16 S-Boxes, i.e. apply them first "leaving out" $P_{1,0}$ (which would go unchanged to round two), then apply them "leaving out" $P_{1,256}$, and so on. However while these options were being studied, a far more serious problem surfaced, concerning the APNL functions. Interestingly, the solution for *that* problem also took care of the "extra bit"—which was not extra anymore.

4.4 Round three: the strange case of the "even-sized" APNL functions

When the alternative designs began to be coded in SAGE, the APNL function chosen was the Gold function, defined by: $y = x^{2^i+1}$, with $x, y \in \mathbb{F}_{2^n}$, and $\gcd(i, n) = 1$. This choice was motivated by the simplicity of its definition (compare with alternative choices for example in (Pott and Edel, 2009, Table 1, p. 3)). In our scenario $n = 16$, thus the smallest possible value for i was chosen: 3. So the APNL function is $y = x^9$. The next step is to write down the formula for its inverse. The simplest approach is to use Theorem 2.2.1 to compute d such that $(x^9)^d = x \bmod 2^{16}$, or equivalently $9d = 1 \pmod{2^{16} - 1}$. The problem with this approach is that the latter modulus is composite: $2^{16} - 1 = 3 \times 5 \times 17 \times 257$. I.e. $\gcd(9, 2^{16} - 1) \neq 1$, which means the congruence has no solution. Worse: if n is even, a power function which is APNL will not be bijective, and thus will not be invertible (Carlet, 2009, Prop. 17)[2]. From the same source however, also came a formula for the inverse, *when n is odd* (*ibid.*, §3.1.6). The inverse of x^{2^i+1} is in fact of the form $y = x^d$, where

$$d = \sum_{k=0}^{\frac{n-1}{2}} 2^{2ik} \pmod{2^n - 1} \tag{2}$$

This suggested that the scheme of Figure 4.1 could be modified, with a different layout of S-Boxes. That layout would be based on the different ways 257 can be written as a linear combination of primes. Two observations: first, we are now considering S-Box layouts that will not allow the CRT isomorphism to be used (at least not directly), and so it does not make sense to stick only to 256 bits—hence the full 257 length was used. And second, to overcome the limitation of an explicit inverse, we needed only to make sure n was odd, not prime. However, requiring primeness seemed the more conservative choice, and because it came with no additional drawbacks, that was the path followed.

The bigger the S-Box, the more memory requires its implementation, but the smaller the S-Boxes used in a round, the stronger its diffusion has to be (otherwise some correlations between input and output might exist). Thus the option chosen was: $19 \times 1 + 17 \times 14$—see §4.7.1 for further details.

2 Also cf. file `gold_odd.sage` for a test of injectivity on $\mathbb{F}_{2^8}$, where the formula for the inverse (Eq. 2) also fails.

4.5 FURTHER CHANGES

However those were not the only changes. Because the CRT can no longer be used directly, experiences were conducted with an alternate mechanism for diffusion, namely using some carefully chosen *constants* in R, which are then multiplied by the output of the S-Boxes. Note that the idea is to have one (different) round constant *per round*; see §5.4 for further details.

This in turn, prompted yet another change. Given that constants were already being multiplied in the big ring, following that by the multiplication of the round key might, in some cases, "undo" part of the diffusion created by the multiplication of the constants. Constants were chosen so as to maximise diffusion, but round keys are random. To avoid this, the way round keys are applied was changed from multiplication in R to XORing.

Lastly, for symmetry reasons, the fourteen 17-bit S-Boxes were applied as depicted in Figure 4.2, and the latter half implement the inverse function of the former half. This final design is described in the next section.

4.6 FINAL ROUND

Figure 4.2 might at first seem a bit daunting, so the first task is to explain it. The round structure depicted takes as input a 257 bitstring (indexed with the coefficients for a generic polynomial of R, as explained in chapter 2). This input bitstring is put through a layer of fifteen S-Boxes: $S_{1,1}$ through $S_{1,7}$ implement the chosen Gold function (x^9) in $\mathbb{F}_{2^{17}}$; and $S_{1,9}$ through $S_{1,15}$ implement the inverse Gold function (x^d with d as in Eq. 2). Note that all these S-Boxes have input and output of length 17. $S_{1,8}$ also implements the Gold function, but on $\mathbb{F}_{2^{19}}$, which means its input and output length is 19. The resulting bitstring is then multiplied by the round constant, and to the result of this operation the round key is XORed.

Figure 4.2 depicts one *encryption* round; for decryption the order of the operations is reversed. This does *not* mean the order of S-Boxes is changed; they remain in the same positions, but now the "bottom half" implement the Gold function, while the "top half" (plus $S_{1,8}$) implement its inverse, on the respective fields. Still concerning the S-Boxes, note that there is no penalty in terms of space requirements, in using both the Gold function and its inverse in the round: even if this was *not* the case, the table for the inverse S-Box would still have to be stored, for it is required for decryption. More on the size of the S-Boxes will be said in section 4.7.1.

We end this section by giving a quick walk-through of this round's implementation in SAGE.[3] The full code is available in the script `round_F2_17_19_constants.sage`.

3 An overview of SAGE is given in Appendix A.

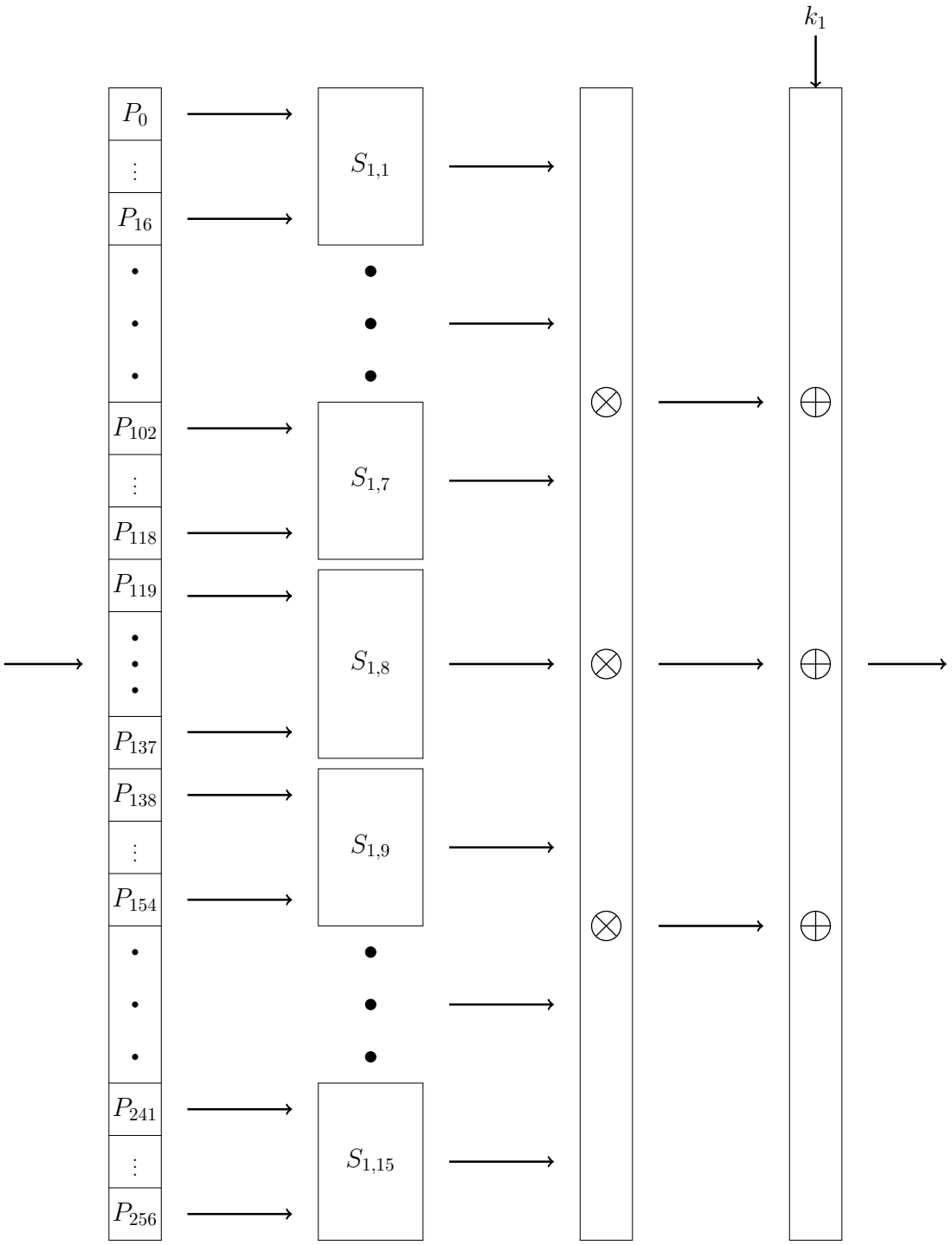


Figure 4.2.: Final round design.

```
GF2=GF(2)
P_GF2.<X> = PolynomialRing(GF2)

mod17 = X^17+X^3+1
mod19 = X^19 + X^18 + X^13 + X + 1
GF2_17.<m> = P_GF2.quotient_ring(mod17)
GF2_19.<n> = P_GF2.quotient_ring(mod19)

def gold_f9(p_xval):
  p_xval = p_xval**9
  return p_xval.list()

# i = 3
# sage: reduce(lambda a, b: a+b, [2**(6*k) for k in range(0, 9)]) % (2**17-1)
# 29127
def gold_inv17(p_xval):
  p_xval = p_xval**29127
  return p_xval.list()

# i = 3
# sage: reduce(lambda a, b: a+b, [2**(6*k) for k in range(0, 10)]) % (2**19-1)
# 466033
def gold_inv19(p_xval):
  p_xval = p_xval**466033
  return p_xval.list()
```

Code Snippet 4.1: Code for the S-Boxes.

The three S-Box Boolean functions are implemented as shown in Code Snippet 4.1. Functions `gold_inv17` and `gold_inv19` receive a polynomial object with `m` or `n` as the indeterminate, respectively, and compute its inverse using an exponent calculated according to Eq. 2. `gold_f9` receives a polynomial object with either indeterminate, and returns the exponentiation in the proper Galois field.

The encryption round is shown in Code Snippet 4.2. It has a straightforward correspondence with Figure 4.2: the input is put through the S-Box layer, then the round constant is multiplied, and finally the round key is XORed, and the output returned. Note that going from the output of the S-Boxes (which is in the small rings) to the multiplication of the constant (which is in the big ring) the (implicit) *join* operator is used.

Decryption is similar, but with one nuance: the round constants must be inverted. In a real implementation both the constants and their inverses would be stored, but here inversion is done with the SAGE one-liner:

```
  inv_const = P_GF2(list(const_i)).inverse_mod(big_modulus)
```

Lastly, the mentioned SAGE script also contains tests that iterate the use of the round function. When this is done, the structure used is that of an SP-network, which means that before the first round, an initial key is XORed to the input (cf. §5.4). This is implicitly stated in both Figures 4.1 and 4.2, where the round key is denoted by k_1 (instead of k_0), precisely because k_0 is the key that is XORed to the plaintext, prior to the first round.

```
def round_enc(mi, ki, const_i):
  aux =  gold_f9(GF2_17 (mi[17*0 :17*1]))
  aux += gold_f9(GF2_17 (mi[17*1 :17*2]))
  aux += gold_f9(GF2_17 (mi[17*2 :17*3]))
  aux += gold_f9(GF2_17 (mi[17*3 :17*4]))
  aux += gold_f9(GF2_17 (mi[17*4 :17*5]))
  aux += gold_f9(GF2_17 (mi[17*5 :17*6]))
  aux += gold_f9(GF2_17 (mi[17*6 :17*7]))

  # 17*7+19 = 138
  aux += gold_f9(GF2_19 (mi[17*7:17*7+19]))

  aux += gold_inv17(GF2_17 (mi[138+17*0:138+17*1]))
  aux += gold_inv17(GF2_17 (mi[138+17*1:138+17*2]))
  aux += gold_inv17(GF2_17 (mi[138+17*2:138+17*3]))
  aux += gold_inv17(GF2_17 (mi[138+17*3:138+17*4]))
  aux += gold_inv17(GF2_17 (mi[138+17*4:138+17*5]))
  aux += gold_inv17(GF2_17 (mi[138+17*5:138+17*6]))
  aux += gold_inv17(GF2_17 (mi[138+17*6:138+17*7]))

  res = fill257((P_GF2(list(aux)) *
                 P_GF2(list(const_i))) % big_modulus)

  return map(lambda a, b: a+b, res, ki)
```

Code Snippet 4.2: Encryption round.

4.7 SOME REMARKS ON IMPLEMENTATION

Although implementation concerns were not the current work's focus, there are some aspects of a more practical nature that cannot be avoided. Chiefly among those is the *size* of the S-Boxes, about which there is enough to say to dedicate a separate subsection to it, §4.7.1.

Besides S-Boxes, the proposed round structure uses polynomial multiplications in the big ring R, and bitwise XOR for the round keys. About polynomial multiplication, it can be a computationally costly operation when considered in its fullest generality (i.e. with arbitrary algebraic structures). However because in this work all the rings and fields have characteristic 2, polynomial multiplication in R of the round constants can be efficiently implemented using an LFSR. Note that some properties that are usually desirable for an LFSR to have—maximal

period, for example—are not necessary here: of interest is only the fact that they offer a fast and simple way to implement the diffusion mechanism.

On what concerns the XORing of round keys, it is also a fast and simple operation, but one which has, in our scenario, a caveat, namely the fact that the bitstrings are of length **257**. As mentioned at an earlier section, this was something to be avoided, if possible. As it was not, what is left to do is to analyse its impact on the cipher's operations, as well as possible mitigation strategies—for further discussion see chapter 8.

4.7.1 *On the size of S-boxes*

There are two different "sizes" to consider: one is the size of the S-Boxes per se (which affect the choice of the layout), and the other is the size of the *implementation* of the chosen S-Box layout. We will address first the former, then the latter.

When seen as a Boolean function that maps n input bits to m output bits, an S-Box can be implemented as a lookup table. The size of such a lookup table is $2^n \cdot m$ bits: we just store the possible outputs, each of m bits; and there must be one output for each of the 2^n possible inputs[4]. When $n = m$, as is the case with the S-Boxes used in this work, it simplifies to $2^n \cdot n$. Furthermore, for decryption it is also necessary to store the inverse S-Box; in the simplified case we just multiply the size by 2: $2^{n+1} \cdot n$. Note that size of an S-Box depends *solely* on the length of its input and output; in particular it does *not* depend on the specific Boolean function it implements.

As mentioned previously, there are several choices for constructing an S-Box layout that encompasses the round's 257 input bits, based on the different ways of writing 257 as a linear combination of primes (the fact the layout will give the round's 257 output bits is guaranteed by only considering S-Boxes with $n = m$). It seemed prudent not to depart too much from the original choice, of 16 S-Boxes of 16 bits each. For this reason the focus was on ways of writing 257 as linear combinations of *two* primes; adding a third would mean the primes would have to be smaller. For example, we have $257 = 10 \times 13 + 10 \times 11 + 1 \times 17$, but with two primes we have $8 \times 13 + 9 \times 17$. The first attempt was to search a layout based on primes 17, 19, and 23, using the code shown in Snippet 4.3. This was the output:

```
{'19': 1, '17': 14}       sbox size in kbytes = 2976
{'17': 7, '23': 6}        sbox size in kbytes = 47648
```

The difference in size is because the S-Box of 23 input bits is considerably bigger than the others:

4 This ignores extra bits that might be required in an implementation, e.g. for padding. But because implementation details are not the focus of the present work, the S-Boxes are here compared only with respect to their "theoretical size".

```
1  pp = MixedIntegerLinearProgram()
2  v = pp.new_variable(integer=True, nonnegative=True)
3  x,y,z = v['17'],v['19'],v['23']
4
5  C = [ x*17 + y*19 , x*17 + z*23 , y*19 + z*23 ]
6
7  def sbox_size(n):
8    return round(2*n*2**n/(8.0*1024),0)
9
10 Sboxes = {'17' : sbox_size(17) , '19' : sbox_size(19) , '23' : sbox_size(23) }
11 keys = Sboxes.keys()
12
13 for c in C:
14   pp.set_objective(c)
15   pp.add_constraint(c <= 257)
16   if pp.solve() == 257 :
17     res = {} ; size = 0
18     for k in keys:
19       res[k] = int(pp.get_values(v[k]))
20       if res[k] != 0: size += Sboxes[k]
21     print str({k:v for (k,v) in res.iteritems() if v != 0}).ljust(25, ' '), \
22           'sbox size in kbytes =', size
```

Code Snippet 4.3: S-Box sizes, with primes 17, 19, and 23.

```
sage: Sboxes
{'17': 544, '19': 2432, '23': 47104}
```

For this reason, and because this attempt produced only two choices, the next idea was to replace the biggest S-Box (23) with one of 13 bits. This is done replacing lines in Snippet 4.3 like so:

```
1  # replaces line 3
2  x,y,z = v['13'],v['17'],v['19']
3
4  # replaces line 5
5  C = [ x*13 + y*17 , x*13 + z*19 , y*17 + z*19 ]
6
7  # replaces line 10
8  Sboxes = { '13' : sbox_size(13) , '17' : sbox_size(17) , '19' : sbox_size(19) }
```

The result is shown below. The sizes are all much smaller, and because the size difference between the last two choices is small, it was decided to go the last one viz. $19 \times 1 + 17 \times 14$.

```
{'13': 8, '17': 9}        sbox size in kbytes = 570
{'19': 6, '13': 11}       sbox size in kbytes = 2458
{'19': 1, '17': 14}       sbox size in kbytes = 2976
```

With the size of the layout decided, the focus is the size of its implementation. As already mentioned, implementation is not the focus of this work, so this is mentioned for completeness. The previous script calculates size assuming that each function (and its inverse) only needed to be implemented *once*. This is effectively a *lower bound* on the size requirement of the layout. But this may not be feasible, for example if all 17 bit S-Boxes implement different Boolean functions (see Figure 4.2). Remember from section 4.6 that $S_{1,1}$ through $S_{1,7}$ implement x^9, and $S_{1,9}$ through $S_{1,15}$ implement its inverse. This needed not be the case: $S_{1,1}$ through $S_{1,7}$ could still implement the Gold function, but with different exponents (with the other 7 S-Boxes implementing the corresponding inverses). Or all fourteen 17 S-Boxes could be different functions. Or even if those fourteen S-Boxes implement the same function, the requirement could be imposed of them processing the input in parallel, instead of sequentially re-using the same implementation. This leads to an *upper bound*, in which the size of an S-Box must be multiplied by the number of such S-Boxes present in a layout. Modifying the code in Snippet 4.3 to calculate these upper bounds is straightforward: the only change needed is to replace line 20 with:

```
1       size += res[k]*Sboxes[k]
```

Then the size requirements for the last three layout choices become as shown below. This further strengths our choice, because its upper bound is actually *smaller* than for the layout using 13 bit S-Boxes.

```
{'13': 8, '17': 9}        sbox size in kbytes = 5104
{'19': 6, '13': 11}       sbox size in kbytes = 14878
{'19': 1, '17': 14}       sbox size in kbytes = 10048
```

4.8 Key schedule and number of rounds

As will be shown in chapter 5, two rounds with the described structure already provide "full diffusion", in the sense of (Daemen and Rijmen, 2002, §3.5). Nevertheless, the security margin of the cipher is very dependent on the number of rounds[5], so one cannot be too careful. And if anything, cryptographic history shows that one is often justified in erring in the side of caution. Consider the case of the FEAL cipher, designed to have a strong round function, so as to need fewer rounds; its four round variant, FEAL-4, succumbed to differential cryptanalysis with only 20 chosen plaintexts Murphy (1990)—and this was not even the worst attack on FEAL-4 (Schneier, 1996, §13.4). The author would be *very* surprised if the proposed round

5 For instance in (Anderson et al., 1998, §2): *«16-round Serpent would be as secure as triple-DES, and twice as fast as DES. However, AES may persist for 25 years as a standard and a further 25 years in legacy systems, and will have to withstand advances in both engineering and cryptanalysis during that time. We therefore propose 32 rounds to put the algorithm's security beyond question»*.

structure could be made into a secure cipher with less than 10 rounds, the minimum for Rijndael (Daemen and Rijmen, 2002, §3.5).

And continuing on the subject of turning the round into a fully-fledged cipher, one thing that has not been described it the key schedule. This will be the most pressing item in the list of future work. But at least for completeness' sake, if for nothing else, the author feels that, as a minimum, the criteria that a good key schedule should meet must be given some mention. Unfortunately, this is easier said then done (Daemen and Rijmen, 2002, §5.8):

> There is no consensus on the criteria that a key schedule must satisfy. In some design approaches, the key schedule must generate round keys in such a way that they appear to be *mutually independent* and can be considered *random* [...] [F]or some ciphers, the key schedule is so strong that [it] appears to make use of components that can be considered as cryptographic primitives in their own right.

For Rijndael, its authors list three criteria that guided the design of its key schedule(*ibid.*). The second one is resistance to something called *related-key* attacks, and the third is resistance to certain attacks in scenarios where the cryptanalyst knows or can chose parts of the key (usually when the block cipher is used as a primitive in some larger construction). For the purposes of the current work, the most relevant seems to be their first criteria: the key schedule must introduce *asymmetry*. Quoting again:

> The first one is the introduction of asymmetry. Asymmetry in the key schedule prevents symmetry in the round transformation and between the rounds leading to weaknesses or allows attacks. Examples of such weaknesses are the complementation property of the DES or weak keys such as in the DES. Examples of attacks that exploit symmetry are slide attacks.

Although the authors of Rijndael employ a somewhat narrow definition of symmetry (*ibid.*, §5.3), it at least partially applies to the round structure this chapter describes—and from here stems the importance of asymmetry for the key schedule to be developed.

4.9 Conclusions

This chapter describes the "end product", if it so may be called, of the work that has been done. The coming chapters describe specific parts of the overall result, until the final chapter on conclusions. For this reason, the concluding remarks that could justifiably be placed here are postponed until chapter 8.

5

LINEARITY

In general terms, linearity referrers to mechanism used to provide *diffusion*. The initial idea was that said mechanism would be based on the ring isomorphism that stems from the *Chinese Remainder Theorem* (CRT). This turned out not to be possible, so another alternative was taught up. However, the CRT is still used somewhat implicitly, which is why this chapter begins with its description. Then it describes the alternative construction for diffusion, and analyses its strength.

5.1 THE CHINESE REMAINDER THEOREM

In its original form, the *Chinese Remainder Theorem* (CRT)[1] consists of a method for solving multiple congruences. To take an example from Wikipedia (a): *«what is the lowest number n that when divided by 3 leaves a remainder of 2, when divided by 5 leaves a remainder of 3, and when divided by 7 leaves a remainder of 2?»*. More formally we have[2]:

Definition 5.1.1 (Chinese Remainder Problem)**.** *Let* $N = q_1 q_2 \cdots q_n$*, with* $gcd(q_i, q_j) = 1$ *for* $i \neq j$*. Given* n *elements* $a_i \in \mathbb{Z}_{q_i}$*, determine* $a \in \mathbb{Z}_N$ *such that* $a \equiv a_i \pmod{q_i}$ *for all* $i = 1, \ldots, n$*.*

The way to solve this problem is to calculate the CRT basis, which in the context of Definition 5.1.1, is a sequence of integers $e = \langle e_1, \cdots, e_n \rangle$, that guarantees that a, computed like so:

$$a = \left(\sum_{i=1}^{n} a_i \, e_i \right) \mod N \tag{3}$$

has the required property, viz. that $a \equiv a_i \pmod{q_i}$ for all $i = 1, \ldots, n$. In order to construct the CRT basis one can use the SAGE function `crt_basis`[3]. We can use it for instance, to solve the example mentioned in the first paragraph; see Code Snippet 5.1.

1 It is so named because the version of this result relating to integer congruences was already known to the Chinese in the first century AD.

2 This brief exposition is based on Professor Valença's lecture notes.

3 An overview of the SAGE background required is provided in Appendix A.

```
sage: crt_basis([3, 5, 7])
[-35, 21, 15]
sage: -35*2+21*3+15*2
23
sage: 23%3
2
sage: 23%5
3
sage: 23%7
2
```

Code Snippet 5.1: Example of using SAGE to solve a simple CRT example.

But the reason the CRT is interesting to us is that, as already mentioned, it can be used to provide diffusion. This is possible because we can generalise it from the integer rings $\mathbb{Z}_{q_i}$ and $\mathbb{Z}_N$ to rings built on more general structures—and then choose one suitable to our purposes.

To use the CRT to manipulate bits, we shall apply it to over $\mathbb{F}_2[x]$. In this ring, the modulus factorises in $\mathbb{F}_2$ as the product of one polynomial of degree 1 and sixteen polynomials of degree 16:

$$\begin{aligned}
x^{257}+1 = (x+1) & \\
&\cdot (x^{16}+x^{12}+x^{11}+x^8+x^5+x^4+1) \\
&\cdot (x^{16}+x^{13}+x^8+x^3+1) \\
&\cdot (x^{16}+x^{13}+x^{12}+x^{10}+x^8+x^6+x^4+x^3+1) \\
&\cdot (x^{16}+x^{14}+x^{12}+x^{11}+x^8+x^5+x^4+x^2+1) \\
&\cdot (x^{16}+x^{14}+x^{13}+x^{11}+x^{10}+x^9+x^8+x^7+x^6+x^5+x^3+x^2+1) \\
&\cdot (x^{16}+x^{14}+x^{13}+x^{12}+x^{10}+x^8+x^6+x^4+x^3+x^2+1) \\
&\cdot (x^{16}+x^{14}+x^{13}+x^{12}+x^{11}+x^9+x^8+x^7+x^5+x^4+x^3+x^2+1) \\
&\cdot (x^{16}+x^{15}+x^8+x+1) \\
&\cdot (x^{16}+x^{15}+x^{13}+x^9+x^8+x^7+x^3+x+1) \\
&\cdot (x^{16}+x^{15}+x^{13}+x^{11}+x^{10}+x^8+x^6+x^5+x^3+x+1) \\
&\cdot (x^{16}+x^{15}+x^{13}+x^{12}+x^{10}+x^9+x^8+x^7+x^6+x^4+x^3+x+1) \\
&\cdot (x^{16}+x^{15}+x^{14}+x^8+x^2+x+1) \\
&\cdot (x^{16}+x^{15}+x^{14}+x^{12}+x^{10}+x^8+x^6+x^4+x^2+x+1) \\
&\cdot (x^{16}+x^{15}+x^{14}+x^{13}+x^9+x^8+x^7+x^3+x^2+x+1) \\
&\cdot (x^{16}+x^{15}+x^{14}+x^{13}+x^{11}+x^{10}+x^8+x^6+x^5+x^3+x^2+x+1) \\
&\cdot (x^{16}+x^{15}+x^{14}+x^{13}+x^{12}+x^{11}+x^8+x^5+x^4+x^3+x^2+x+1)
\end{aligned} \tag{4}$$

Each of the seventeen "small" polynomials is irreducible (cf. snippet A.1), which means that each of the rings r_i is actually a field—their moduli are all irreducible polynomials—but as the CRT applies to rings, we think of them as rings, and tacitly ignore their additional field structure. This means that we can construct one particular algebraic structure, which we will show in the next section to be isomorphic to the "big ring" R. It is the product ring of the "small rings", $\prod_{i=0}^{16} r_i$, which elements we will represent with a 17-tuple: $(r_0, r_1, \ldots, r_{16})$. We will apply the CRT to these structures having the "big ring" R play the same "role" as $\mathbb{Z}_N$, and similarly with the "small rings" r_i and $\mathbb{Z}_{q_i}$ (cf. §5.1). Then we can define an isomorphism, which allows us to switch between the two representations: to go from small to big we use the CRT, and to go from big to small use (polynomial) modular division. This is the subject of our next section.

5.2 POLYNOMIAL RING ISOMORPHISM

With the structure described at the end of the previous section, computing the CRT basis results in polynomials of degree 252–256. To go from polynomials to bits, we now derive the matrix to transform the vector ("string") in the small rings into the corresponding vector in the big ring R (the matrix that reverses the process is its inverse).

Let $p = [p_0, p_1, \ldots, p_{16}]$ be a row vector of the polynomials in the small rings ($deg(p_0) = 0$, the others have degree at most 15). The CRT basis is, as mentioned, composed by 17 polynomials of degree ≥ 252. If we denote it as column vector $e = [e_0, \ldots, e_{16}]^T$, then the application of Equation 3 is straightforward: one has only to do the matrix multiplication

$$p \cdot e = \sum_{i=0}^{16} p_i e_i \tag{5}$$

To go from polynomials to bits, consider the *second* term of that summation (we will deal with the first one in a moment). We have $p_1 e_1 = p_{1,0} x^0 e_1 + p_{1,1} x^1 e_1 + \cdots + p_{1,15} x^{15} e_1$. If we let $(xe_1)_j$ denote the coefficient of the jth term of the polynomial xe_1 (and similarly for the other $x^i e_1$), then we can already write $p_1 e_1$ as a product of a bit row vector by a bit matrix:

$$p_1 e_1 = [p_{1,0}, \cdots, p_{1,15}] \begin{pmatrix} e_{1,0} & e_{1,1} & \cdots & e_{1,256} \\ (xe_1)_0 & (xe_1)_1 & \cdots & (xe_1)_{256} \\ (x^2 e_1)_0 & (x^2 e_1)_1 & \cdots & (x^2 e_1)_{256} \\ \vdots & \vdots & \ddots & \vdots \\ (x^{15} e_1)_0 & (x^{15} e_1)_1 & \cdots & (x^{15} e_1)_{256} \end{pmatrix} \tag{6}$$

The key observation here is that when multiplying a polynomial by a scalar, the coefficients of the result polynomial are obtained by multiplying the corresponding coefficients of the

argument polynomial with the scalar. So Eq. 6 really yields the coefficients of $p_1 e_1$, ordered similarly as the coefficients of p_1 are in the row vector.

Continuing this process, we can write a bit row vector for the coefficients of all the terms of summation 5, like so:

$$[p_{0,0}, p_{1,0}, \cdots, p_{1,15}, \cdots, p_{16,0}, \cdots, p_{16,15}] \tag{7}$$

For this 257 bit row vector, the corresponding matrix is the 257×257 CRT matrix (horizontal lines added for readability):

$$CRT = \left(\begin{array}{cccc}
(e_0)_0 & (e_0)_1 & \cdots & (e_0)_{256} \\
\hline
(e_1)_0 & (e_1)_1 & \cdots & (e_1)_{256} \\
(xe_1)_0 & (xe_1)_1 & \cdots & (xe_1)_{256} \\
(x^2e_1)_0 & (x^2e_1)_1 & \cdots & (x^2e_1)_{256} \\
\vdots & \vdots & \ddots & \vdots \\
(x^{15}e_1)_0 & (x^{15}e_1)_1 & \cdots & (x^{15}e_1)_{256} \\
\hline
(e_2)_0 & (e_2)_1 & \cdots & (e_2)_{256} \\
(xe_2)_0 & (xe_2)_1 & \cdots & (xe_2)_{256} \\
\vdots & \vdots & \ddots & \vdots \\
(x^{15}e_2)_0 & (x^{15}e_2)_1 & \cdots & (x^{15}e_2)_{256} \\
\hline
\vdots & \vdots & \ddots & \vdots \\
\hline
(e_{16})_0 & (e_{16})_1 & \cdots & (e_{16})_{256} \\
(xe_{16})_0 & (xe_{16})_1 & \cdots & (xe_{16})_{256} \\
\vdots & \vdots & \ddots & \vdots \\
(x^{15}e_{16})_0 & (x^{15}e_{16})_1 & \cdots & (x^{15}e_{16})_{256}
\end{array}\right) \tag{8}$$

This also shows why we did not use the first term in summation 5: because p_0 is a polynomial of degree 0, it only has one monomial, and thus only "contributes" one row to the matrix, viz. the first one, comprised of the coefficients of e_0.

Conversely, if we *invert* the CRT matrix, and then multiply it by a bit row vector of the coefficients of an element of R (independent term on the left; padded to length 257 as needed), we obtain the coefficients of the remainders in the small rings, ordered as in expression 7.

Because we can represent the CRT transformation by a matrix, we know it is an homomorphism. More concretely, if $p = (p_0, \ldots, p_{16})$ and $p' = (p'_0, \ldots, p'_{16})$ are two elements in r, and the CRT transformation is denoted by f, then from the properties of matrices we know that

$$f(p + p') = f(p) + f(p') = P + P'$$

where P and P' are the elements of R corresponding to p and p' respectively. A similar property holds for multiplication, thus the transformation is homomorphic. The fact that the matrix is squared and invertible means the CRT transformation is also a bijection, which in turn means the homomorphism is actually an isomorphism. Furthermore, its inverse transformation is also isomorphic (because the inverse of a square invertible matrix is another square invertible matrix).

5.2.1 *The parity bit*

Let P be a polynomial in R, and $(p_0, \ldots, p_{16})$ its corresponding representation in the product ring. Then, according to the CRT, $p_0 = P \mod (x+1)$. This can be written as $P = (x+1)Q(x) + p_0$, for some polynomial $Q(x)$ of degree 255 (remember that p_0 is a polynomial of degree 0). If we set $x = 0$, then we get $\sum_{i=0}^{i=256} P_i = p_0$, i.e. p_0 is the parity of the bitstring that contains the coefficients of P.

5.3 DIFFUSION ANALYSIS OF THE CRT MATRIX

Just as for *confusion*, the concept of *diffusion* was introduced in a seminal paper by Claude Shannon (Shannon, 1949):

> Two methods (other than recourse to ideal systems) suggest themselves for frustrating a statistical analysis. These we may call the methods of diffusion and confusion. In the method of diffusion the statistical structure of M which leads to its redundancy is "dissipated" into long range statistics—i.e., into statistical structure involving long combinations of letters in the cryptogram. The effect here is that the enemy must intercept a tremendous amount of material to tie down this structure, since the structure is evident only in blocks of very small individual probability. Furthermore, even when he has sufficient material, the analytical work required is much greater since the redundancy has been diffused over a large number of individual statistics.

This is an admittedly abstract "definition" and, to make matters worse, that abstractedness is somewhat inherent, as the authors of Rijndael readily concede: *«The exact meaning of the term diffusion depends strongly on the context in which it is used.»* (Daemen and Rijmen, 2002, §9.2.3). In the context of block cipher design, what this means is that changing one bit of the input should change "as many" bits of the output as possible. More formally, define a *multipermutation* as follows (Vaudenay, 1994):

Definition 5.3.1. *A (r, n)-multipermutation over an alphabet Z is a function f from Z^r to Z^n such that two different (r + n)-tuples of the form (x, f(x)) cannot collide in any r positions.*

Note that this definition is more general than bits, in that it applies to functions that take values from an essentially arbitrary set of symbols (of which $\mathbb{F}_2$ is a concrete example). A function with that property achieves *perfect diffusion*: changing t symbols of the input changes at least $n - t + 1$ symbols of the output (*ibid.*). Indeed, if you change t symbols from x, obtaining x', then by Definition 5.3.1 $(x, f(x))$ and $(x', f(x'))$ cannot collide in r or more components—which means they have to collide in *less* than r components, or equivalently, they have to *differ* in *at least* $n + 1$ components. If x and x' already differ in t components, then $f(x)$ and $f(x')$ have to differ at least in $n + 1 - t$ components—thus f achieves perfect diffusion.

Besides verifying Definition 5.3.1, if f is a *linear transformation*, then it can be shown that the set of $(r + n)$-tuples of the form $(x, f(x))$ constitute a *Maximum Distance Separable* (linear) code, or MDS code for short[4]. Accordingly, we have the following definition:

Definition 5.3.2 (MDS matrix)**.** *A matrix is said to be an* MDS matrix *if it is the matrix of a linear transformation f, such that the set of all tuples $(x, f(x))$ forms an MDS code.*

Unfortunately, the CRT matrix is **not** an MDS matrix, a fact that is readily illustrated by noting the following: say your input is a vector that consists only of zeros. Flip the bit in position i. Then the number of bits that are flipped in the output equals the Hamming weight of row i of the CRT matrix. That value is 257 for the first row, and 144 for all the others, which is less than $257 - 1 + 1 = 257$, which is the number of bits that should be flipped in CRT was indeed an MDS matrix. The Hamming weights of the CRT matrix are calculated with the code in script `crt_hw.sage`.

This fact is not specific to the CRT matrix: in fact, for any *non-trivial binary* matrix, it will *not* be MDS—see §B.4 for details. But it did suggest another approach: the CRT matrix is not MDS when seen as a *binary* matrix, but maybe it would be if it is seen as operating on some larger blocks. In other words, if we discard the bit from r_0 and look at the (256 bit) input vector as being composed of, for example, 32 "substrings", each of length 8. We multiply it by the CRT matrix and look at the output in a similar manner, counting how many "symbols" (each a substring of length 8) have changed.

The described approach is implemented in file `crt_flipmap.sage`. The input is divided into blocks of the given length, a number of those blocks is chosen randomly, and each is changed, also randomly. Then if the corresponding block in the output is changed, a 1 is printed; otherwise 0. Thus we obtain a "matrix of changes". Note that because this maps differences, the linearity of the CRT matrix makes the actual values irrelevant; thus the initial value was the 0 vector, which was randomly flipped. It was not possible to test all possible changes, the results obtained suggest that closest this matrix comes from showing the MDS property is when it is seen as mapping 16 input symbols (each of length 16) to 16 output symbols (idem.). Then, in all the tests, changing any (one) block always led to all

4 See Appendix B for details.

blocks being changed in the output. Note that this result does *not* show that the 16×16 transformation is MDS. But *if* it is, then it *has* to be with this block size—for smaller sizes there are elements that remain unchanged, and this 16 is the largest possible block size.

5.4 Constants in R and "full diffusion"

As described in chapter 4, changing the S-Box layout, to something that no longer coincided with the small rings tuple, weakened the case for using the CRT matrix to provide diffusion. One of the alternatives thought up to replace the diffusion mechanism was to use specifically chosen values in R, which are multiplied with the output of the S-Boxes, seen as an element of the big ring. This seems to work well, as described below.

The values for the constants were selected taking into account their Hamming weight. More precisely, they were chosen to have approximately half of the bits set to 1. Furthermore, they were constructed so as to have the 1s uniformly distributed. Multiplication by a polynomial, in R, can be seen as a combination right-shifting and round-robin: to "multiply" a bitstring by x^m, you just shift all bits m positions to the right (increasing index values), and after index 256 you go back to index 0. If the polynomial has more than one monomial, you just repeat the operation for the new exponent.

The CRT isomorphism allows this multiplication to be seen as taking place in the small rings. And because they are "mis-aligned" with the S-Box layout, rotating the bitstring in the small rings will spread the bits to the adjacent S-Boxes. Although this can seem a relatively weak diffusion mechanism, a simple test shows it to be stronger than what it might appear. Choosing the constants to have half the bits set to 1 was a first guess; it is still not clear whether it can be improved. The constants were calculated using the script `diffusion_constants.sage`.

That script just chooses constants with appropriate Hamming weight; the values that are actually used can be seen in the `round_F2_17_19_constants.sage` script. As mentioned in §4, the same script also implements a simple test for the diffusion capabilities of the round construction described in that chapter. The idea comes from (Daemen and Rijmen, 2002, §3.5), where it is stated that *«[t]wo rounds of Rijndael provide 'full diffusion', in the following sense: [...] a change in one state bit is likely to affect half of the state bits after two rounds»*. To apply this idea to our round design, two functions were implemented in the aforementioned SAGE script: one to generate a random input, encrypt it, flip one bit to the input, encrypt the modified input, and count the differences in the outputs; and another function to run this test over a pre-specified number of iterations. Actually, the "flipping" function that was implemented is more general, because it allows for the number of input bits to be flipped to be given as an argument to it. Thus the test that was implemented to test changing 1, 10, 100, 200 and 256 (i.e. flip all but one) input bits, and see how many changes

occurred in the output; for each case ten thousand runs were executed. The results were the following:

```
Flipping 1 bit(s): 128 bits flipped on output
Flipping 10 bit(s): 128 bits flipped on output
Flipping 100 bit(s): 128 bits flipped on output
Flipping 200 bit(s): 128 bits flipped on output
Flipping 256 bit(s): 128 bits flipped on output
```

The first thing to note about the results is that they are very uniform. Contrast with what happens with ten rounds—there is more dispersion:

```
Flipping 1 bit(s): 131 bits flipped on output
Flipping 10 bit(s): 123 bits flipped on output
Flipping 100 bit(s): 131 bits flipped on output
Flipping 200 bit(s): 129 bits flipped on output
Flipping 256 bit(s): 128 bits flipped on output
```

The next thing is that although 128 is very close to half, it is *not* half of the 257. Given the high number of iterations, the displayed number of output changes is an average; but still, it seems that on some cases 129 bits should have been flipped. If the process was random one would expect to see the number of bits flipped oscillate between 128 and 129. This could have been caused by rounding errors, although more thorough testing is needed to ascertain if that was the case.

5.5 CONCLUSIONS

This chapter described the two linear constructions that could be used to provide diffusion. The one that ended up being chosen—the second one—seems promising, although more tests have to be done. In particular, it still has to be checked that the flipped bits occur at random locations. I.e. that there isn't one specific output bit that is likely to be changed (or not to be changed) when a specific input bit is changed. Nevertheless, the result already achieved—that for a number of (different) input changes, half of the output bits change—is a promising starting point. Another thing to verify is if, with the round design used, the fact that the diffusion primitive is not a multipermutation introduces any significant weaknesses.

6

NON-LINEARITY

> *The design took advantage of certain cryptanalytic techniques, most prominently the technique of "differential cryptanalysis", which were not known [at the time] in the public literature.*
>
> Don Coppersmith

Nonlinearity refers to the mechanism used to provide *confusion*, which is typically provided in a construction called *S-Box*. This chapter begins with a thorough description of differential cryptanalysis, one of the oldest attacks to target specific weakness in S-Box design. That analysis is used to assess the choices made for the round function's nonlinear components.

Before proceeding, it must be noted that in the following sections, two of cryptography's most famous *dramatis personæ*, Alice and Mallory, are used sometimes, to refer to the honest party and the active attacker (i.e. the cryptanalyst) respectively.[1]

6.1 DIFFERENTIAL CRYPTANALYSIS

Differential cryptanalysis is a chosen plaintext attack. It was first (publicly) described in Biham and Shamir (1991). The objective is to determine the value of key bits of the round key for a certain round, usually the last one. It was originally applied to DES; but the latter's structure tends to make the essence of the attack harder to grasp. A better explanation might be Heys (2002), in which differential cryptanalysis is applied to a simplified cipher (but having the SP-network structure). The approach we will follow here is based on that one.

The idea of the attack is to exploit the existence of correlations between the input and output differences. Let f be a generic Boolean function, and consider two (input, output) pairs: (x_0, y_0) and (x_1, y_1). If $x_0 + x_1 = a$, then we say a is the *input difference* (i-XOR) of the pair ("difference" meaning bitwise XOR). Similarly, if $y_0 + y_1 = b$, the *output difference*

1 In full disclosure, the author must note that he was *very* tempted to replace those characters with their Hindu counterparts, as suggested here: http://drpartha.org.in/publications/alicebob.pdf. But in the end he chose not to, not least because cryptography is already complicated enough with the existing cast—no need for a new one.

(o-XOR) is b. These differences might also be denoted by Δx and Δy respectively. For *one* encryption round, we can calculate the probability $P(\Delta y = b \mid \Delta x = a)$—i.e. the probability of, given x_0 and x_1 as above, having $round_{enc}(x_0) + round_{enc}(x_1) = b$. The objective of differential cryptanalysis is to predict, from Δx, the input difference to the last encryption round, and use that information to extract bits from the last round key. This can be done if there exists one input difference (to the cipher) that will produce a specific output difference (at the end of $n-1$ rounds), with probability significantly higher (or lower) than most other output differences (for that fixed input difference).

In what follows, the pair of differences at specific points of the cipher (e.g. input and output), $(\Delta x, \Delta y)$, is called a *differential*. A *differential characteristic* is a differential that contains all the intermediate differences, if any, i.e. $(\Delta x, \Delta w_0, \Delta w_1, \ldots, \Delta y)$. The goal is to construct a high probability differential between a plaintext (input) difference, and the input difference to the last round (i.e. the output difference after $n-1$ rounds). Let us break this problem into parts. A final note: because differential cryptanalysis is a chosen plaintext attack, in the following sections Mallory is tacitly assumed to be able to obtain ciphertexts for plaintexts of his choice.

6.2 THE S-BOXES

The first step is for each type of S-Box[2], to tabulate how, for a fixed input difference, different pairs of inputs give raise to different possible output differences. To illustrate this, consider the S-Box of the Mini-AES (toy) cipher (Phan, 2002), specified in Table 3. Note that this is a boolean function where both input and output consist of bitstrings of length 4, and it is a bijection, and thus invertible. The ensuing discussion, although also illustrated with examples of length 4, generalises, *mutatis mutandis*, for bit strings of arbitrary length.

Input	Output	Input	Output
0000	1110	1000	0011
0001	0100	1001	1010
0010	1101	1010	0110
0011	0001	1011	1100
0100	0010	1100	0101
0101	1111	1101	1001
0110	1011	1110	0000
0111	1000	1111	0111

Table 3.: The S-Box of Mini-AES.

2 Note that while some ciphers only use one specific S-Box (e.g. AES), others do not: DES for instance, uses 8 different S-Boxes.

Consider now a specific i-XOR, say 1011. We can map the output differences, by constructing a table like Table 4.

o-XOR	Input values
0010	0000, 1011, 0001, 1010, 0011, 1000, 0110, 1101
0101	0100, 1111
0111	0010, 1001
1101	0111, 1100
1111	0101, 1110

Table 4.: o-XOR distribution for fixed i-XOR 1011.

This table must be read as follows: for any o-XOR, corresponds a list of input values. For each of these values, there exists another input value such that XORing them gives the fixed i-XOR (1011 in the case of Table 4), and XORing the *output* of both yields the corresponding value in the o-XOR column. If the i-XOR is different than 0000 (see below), this allows to group the input values pairwise, and indeed in Table 4 they are juxtaposed, meaning the pairs for e.g. o-XOR 0010 are $(0000, 1011)$, $(0001, 1010)$, etc. Note that due to the way it is constructed, in any given distribution (table), all the input values are listed, exactly once.

There are a couple of remarks that must be made about the o-XOR distribution table. First, if the i-XOR happened to be 0000, the table would have only one line, namely for the o-XOR value 0000—and that entry would contain *all* 2^4 possible values. If we were to list the pairs, it would contain all 2^4 values, *twice*, for to obtain an i-XOR of 0000, we would have to XOR each input value with itself. Hereinafter we shall assume that this sort of table is always constructed for an i-XOR different than 0000.

Second, not all possible o-XOR values are listed in the left column. What this means is that for this S-Box, there is *no* pair of input values (with XOR 1011) such that the XOR of their outputs is one of the missing values. For example, there is no pair of input values which has an o-XOR of 1000. Furthermore, it would not be possible—in an S-Box with the same number of input and output bits—to have *all* possible XORs in the left column. To see why, let the boolean function implemented by the S-Box be denoted by f, a be an input difference, and b an output difference. If there is an input value x, for which there exists another input value $x + a$ such that $f(x) + f(x + a) = b$, then both x and $x + a$ will appear under the entry b, *and nowhere else!* This effectively partitions the 2^4 values into groups of 2, and there are 2^3 such groups—and thus the table's left column can have at most 2^3 entries. More generally, for length n, the table can have at most 2^{n-1} entries—and there will be much more to say about the boolean functions for which this is the case.

Third, this allows us to compute conditional probabilities of the kind mentioned above. For example, the probability that, given inputs with i-XOR 1011, the o-XOR is 0010 is $8/16 = 1/2$, because there are 16 input values for which there exists another input such that

their XOR is 1011, and from these, there are 8 such that their o-XORis 0010. Symbolically $P(\Delta y = 0010 \mid \Delta x = 1011) = 1/2$.

Fourth, note that Table 4 is, in some sense, very "non-uniform": for example, the first row has half of the input values! The most "non-uniform" case is when f is *linear*: indeed for any x_1, x_2, we have $f(x_1 + x_2) = f(x_1) + f(x_2)$. If we were to construct an o-XOR distribution table, then $\forall\, x_1, x_2 : x_1 + x_2 = a$, so $f(x_1) + f(x_2) = f(a)$. Thus the table would have only one entry, corresponding to the o-XOR value $f(a)$.

Although S-Boxes are supposed to be non-linear, it is worthwhile to take a closer look to what happens when the function f it implements is linear. Linearity can be seen as a degenerate case of non-linearity[3], and analysing those extreme cases can provide valuable insight into the way an attack (or more generally, an algorithm) work. This is the topic of the next section.

6.2.1 *The simplest case*

Consider the construction depicted in Figure 6.1. This is as simple a cipher (with an S-Box) as the author can envisage: to encrypt plaintext x, XOR it with key k, and the result y is put through the S-Box (which implements boolean function f) producing ciphertext z (only the key is assumed to be secret). Note that the length of the key and of the input to the S-Box are the same. This diagram can be considered a "zoom-in" on the diagram of a generic SP-cipher: x would be part of the input (or of the output of the previous round), k would be part of the round key, and z would be part of the round output. It could also be seen as a standalone construction, which can be made secure as long as the S-Box is "big enough"—at the very least, it should make building a distribution table an impractical task[4]. However, in this section and the next two, the purpose is to convey the intuition behind differential cryptanalysis and thus all the S-Boxes involved will be "realistic", in the sense of being small enough to make computing distribution tables something easily done via computer.

For carrying out differential cryptanalysis, we start by supplying the oracle with a plaintext message x_0, and obtaining the corresponding ciphertext z_0. Similarly, we obtain the pair (x_1, z_1), and compute $x_0 + x_1 = a$. Notice that if $x_0 + k = y_0$ and $x_1 + k = y_1$, then $y_0 + y_1 = x_0 + x_1 = a$. I.e. we know the i-XOR to the function f, which means we can

3 If f is linear when $f(a+b) = f(a) + f(b)$, then it is nonlinear when this conditions fails, i.e. when $f(a+b) = f(a) + f(b) + \gamma$, for some nonzero γ. Thus the linear case can be seen as a degenerate case of nonlinearity, with γ set to 0.

4 The reason for using the wording "can be made secure" is because as depicted, it is *not* secure, for reasons explained at the beginning of §6.2.2. But barring that issue, this construction is secure because *«[c]onfusion alone is enough for security. An algorithm consisting of a single key-dependent lookup table of [adequate size] would be plenty strong. The problem is that large lookup tables require lots of memory to implement [...]. The whole point of block cipher design is to create something that looks like a large lookup table, but with much smaller memory requirements.»* (Schneier, 1996, §14.10). A lookup table being of "adequate size" means precisely that it is infeasible to construct a distribution table for it.

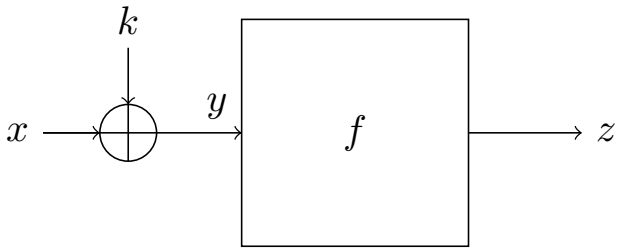


Figure 6.1.: Simple cipher.

compute the input distribution for it, for input difference a. XORing z_0 with z_1 gives the o-XOR of the pair, and from the distribution we then obtain a list of possible inputs *to the function* f (i.e. the position y in the diagram). Denote this set of potential inputs (to f) by $\mathfrak{Y}$, and consider one of the inputs (to the cipher), say x_0. Then it must be the case that $x_0 + k \in \mathfrak{Y}$. We can then construct a set of potential keys, like so: $\mathfrak{K} = \{y + x_0 | y \in \mathfrak{Y}\}$. This set *must* contain the real key k (among other wrong keys). If we repeat this experiment with a different pair, we obtain a new set $\mathfrak{K}$ of potential keys. We can keep a counter of how many times each potential key has shown up so far; the correct key is the one the shows up *every* time (we keep repeating the experiment until there is only one key that shows up every time). Using the S-Box of AES and the Gold function, this process is exemplified in file `dc.sage`.

Note that, with the toy example of Figure 6.1, the *seemingly* hardest case is when f is linear, because it is then that the set of potential keys is the largest—which in turn requires a larger number of pairs for a successful attack. But a closer look reveals an unexpected subtlety: in the linear case, the distribution table only contains one row, which contains all possible inputs, and furthermore this is true for *any* distribution, regardless of its i-XOR. Which means the set of possible keys will contain *all* keys, always[5]. What this means is that irrespective of the number of (plaintext, ciphertext) pairs that are used, the set of possible keys will still be the set of *all* keys. Thus one is led to the conclusion that not only a linear cipher is the hardest case, it is, in fact *impossibly hard: differential cryptanalysis will fail against any linear cipher!*[6]

Of course a linear cipher is not a good cipher, because it is simple to distinguish from a random function. But still, it is a surprising conclusion, that differential cryptanalysis completely fails when applied to one. With hindsight, it makes some sense, because differential cryptanalysis is based on exploring how input differences propagate, and with linearity, they

5 Remember that the set of possible keys is constructed by XORing one of the inputs with all the possible inputs. If the latter always contains all bit strings of length n, then XORing all its elements with a constant always yields a permutation of the same set of all bit strings of length n. This means the set of all possible keys always remains constant, and indeed equal to the set of all bit strings of length n.

6 The reason this holds for *any* linear cipher is that any cipher is an implementation of some Boolean function, and thus can be "written" in the form of Figure 6.1. Note that even when considering a cipher that does not start with XORing a round key, if the former is linear, then the XOR operation preserves that property. It is this fact that allows the claim to be stated in its most general form, for *any* linear cipher—because if f is linear, then Figure 6.1 can represent any linear cipher.

always propagate in the same way, regardless of the key. It seems an analogous situation to trying to recover the key from the "ciphertext" in the "cipher" defined as $E(k, m) = m$; even though the "cipher" is useless, attempting to recover the key will always result in failure.

Faced with so surprising a result, one can naturally ask what would be the hardest *but possible* case. That is, what would the distribution table have to look like in order to make a differential cryptanalysis attack possible, in the hardest case for the adversary. A similar reasoning as applied to the linear case suggests two alternatives. We could try to weaken the linear case, having a distribution table with two rows, one having one pair of inputs, and the other one having the remaining pairs. Thus we get a distribution like Table 5.

o-XOR	**Input values**
b	x_1, x_2
b'	All except x_1, x_2

Table 5.: Almost linear function, for i-XOR a.

Note that $x_2 = x_1 + a$. We can compute the probability of b and b'. The probability of b is the probability of choosing an input value x such that $f(x) + f(x + a) = b$. There are two such values, viz. x_1 and x_2, and thus $P(\Delta y = b \mid \Delta x = a) = 2/2^n$, where n is the bit string length. For b' there are $2^n - 2$ values, and so $P(\Delta y = b' \mid \Delta x = a) = (2^n - 2)/2^n = 1 - 2/2^n$. Differential cryptanalysis is a chosen plaintext attack, which means the adversary can choose what plaintexts get encrypted; furthermore, the attack proceeds by correlating input and output differences, which do not depend on, for example, the meaning of the plaintext. This is why it seems reasonable to assume that all plaintext blocks are equi-probable—which is the assumption underlying the calculation of $P(\Delta y = b \mid \Delta x = a)$ and $P(\Delta y = b' \mid \Delta x = a)$. Let $N_{b'}$ be the set of all input values in the second row of Table 5. When the attacker begins trying input/output pairs, it is very likely that the inputs will belong to $N_{b'}$; but because this set does *not* contain all strings of length n, XORing its elements with the inputs will yield *different* sets of possible keys, depending on the specific input. It is the fact that this difference exists that means that some keys will be seen more frequently than others, and that makes it possible to guess the right key—or at least constrain the set of possible right keys. But because this difference is small—at most two elements are different—the attacker might be required to try a large number of pairs before there exists one (or a small number) of keys that show up significantly more often than others. Of course if the attacker happens to try either input x_1 or x_2, then the game is up—he now has a set of two possible keys, which he can just try and see which one fits (remember it is assumed that all the details of the cryptosystem are know to him).

Which leads us to the second alternative, which is to use a function with a distribution like Table.

o-XOR	Input values
b	half of the input values
b'	other half of the input values

Table 6.: "Almost linear function", for i-XOR a.

We have $P(\Delta y = b \mid \Delta x = a) = P(\Delta y = b' \mid \Delta x = a) = 2^{n-1}/2^n = 1/2$. In either case, the sets of possible keys half the size of the set of all possible keys. While this is worse for Alice and better for Mallory (the sets of possible keys are smaller than with b' in Table 5), it at least does not have the catastrophic weakness of o-XOR b in that same distribution.

Lastly, one can ask the converse question, viz. how such the distribution table look to make Mallory's job easier. The naïve answer would be to say for each o-XOR there should exist only one input; but as seen above, that is not possible (because inputs must be grouped pairwise). The next best for Mallory is when there exist two inputs, $(x, x + a)$, for each o-XOR. In such a scenario, Mallory tries one random pair, obtains two possible keys, and disambiguates by trial and error. However, as we shall see, as the diagram becomes more complex, the conclusions change considerably.

6.2.2 *The* iterated *simplest case*

The situation completely changes when you start *iterating* that construction, i.e. adding more *rounds*. Before going to that example however, it must be pointed out that as depicted in Figure 6.1, the construction has one glaring fault: because f is publicly known, the cryptanalyst can compute y, given z. Which means the oracle can be used to obtain a two time pad, three time pad, etc. The construction was used like this to illustrate the simplest usage of the o-XOR distribution, but a more realistic model would XOR the output of the last S-Box with a round key, and *that* would be the output of the cipher. We shall do this in our next construction, with two rounds, depicted in Figure 6.2.

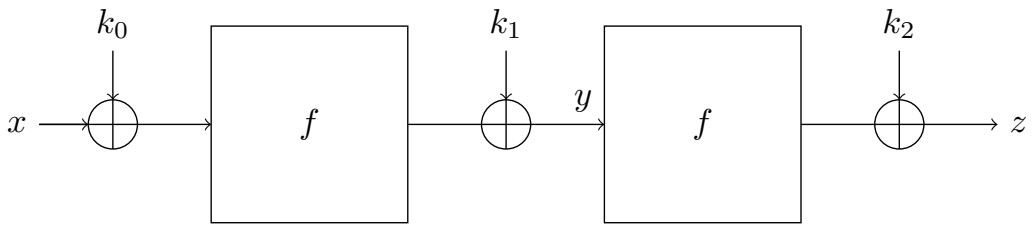


Figure 6.2.: Two round simple cipher.

Observe that here the previous strategy will not work, not only because there is a final XOR operation, but also because as there are two rounds, correlating input with output differences is no longer a straightforward task. Let us start by studying what happens when

f is *linear*—the impossibly hard case for cryptanalysis in the previous section. Say we have an input difference $\Delta x = x_0 + x_1$, which is unchanged by k_0. Because f is linear, we know—with probability 1—the output difference of the first S-Box: it is the o-XOR value of the only row of its distribution table. That difference is unchanged by k_1, which means it is equal the input difference to the second S-Box—position y in Figure 6.2. And again because of linearity, we know the output difference of the second S-Box (Δz because it is unchanged by k_2), which in turn means we know the output difference of the ciphertexts—all with probability 1. Knowledge of the difference at the output of the second S-Box is not very helpful—it is always equal to Δz. But knowing the difference at point y suggests a very simple algorithm to recover k_2: given two input/output pairs (x_0, z_0) and (x_1, z_1), compute $y_0 = f^{-1}(z_0 + k_2)$ and $y_1 = f^{-1}(z_1 + k_2)$, for all values of k_2. If the difference at point y is the expected value, then the guess for k_2 is a possible key—otherwise it is a wrong key and can be discarded. Except this will not work when f is linear. To see why, let b be the o-XOR of the second S-Box, and a its i-XOR, when the plaintext XOR is Δx (this means that $b = \Delta z$ and $a = \Delta y$). From the linearity of f we have $b = f(a)$. But this, together with the fact that f is a bijection, means that $a \neq \Delta y \Rightarrow b \neq \Delta z$, which is against the hypothesis. In other words, regardless of the value of k_2, the value of the i-XOR of the second S-Box is always the same—which gives us no way to constrain the possible values for k_2. Thus, just like in the case of Figure 6.1, with a linear cipher differential cryptanalysis does not work.

If the function f is not linear, then before worrying about generating the set of possible keys, we must first attempt to relate Δx with Δy. The non-linearity of f means that difference propagation is not deterministic: starting with an i-XOR Δx, there are several possible values for Δy[7]. We can refine the aforementioned algorithm for guessing k_2 if, for a given Δx there is one value of Δy that *is significantly more likely than all the others*. This is a particular case of a differential characteristic. Let us say that for a particular difference $\Delta x'$, there is a particular difference at point y, $\Delta y'$ that is more likely than the alternative values. A pair of plaintexts (with difference $\Delta x'$) that generates at point y the difference $\Delta y'$ is called a *right pair*. Otherwise it is called a *wrong pair*. We expect right pairs to be more likely, because of the high probability of the characteristic.

To recover k_2, we start with maintaining a counter for each key. For each pair of plaintext/ciphertext, we decrypt both ciphertexts with all possible keys, and compute Δy via partial decryption, like done above, and if $\Delta y = \Delta y'$, then we increment the counter for the key used. Because we expect right pairs to occur often, the right key counter will also be incremented often. But if a wrong pair has occurred, then partially decrypting with the right key is unlikely to cause $\Delta y = \Delta y'$ (it would only happen by chance)—which means the counter for the right key will likely *not* be incremented. And if, for a right pair, the decryption is

7 In terms of distribution tables what this means is that there is, for a given i-XOR, more than one row in the corresponding table.

made using a wrong subkey, then the bytes entering the "inverse S-Box" (f^{-1}) can essentially be considered random, which means it is also unlikely the previous equality will hold. All of this means that, after enough pairs have been tried, there will be one key with the highest counter, which will very likely be the correct key.

In section 6.2.1 the main concern was to make the size of the sets of possible keys as small as possible. Here that is at most, an ancillary concern. And that makes sense because, as previously mentioned, we assume the S-Boxes to be of realistic size, which means they are small, and that the distribution tables and sets of possible keys keys are always manageable by computers. The main concern now has been to choose a difference at a specific point which is more likely than the other possibilities. It is this that makes the attack possible. Which means that the hardest case for Mallory is when all differences have *the same* probability. The best case (for Alice) seems to be when for each o-XOR, there is only one possible input pair—this makes all differential characteristics have approximately the same probability, which means the above algorithm will not work, because all the counters will be incremented more or less the same number of times.

6.2.3 *Differential Cryptanalysis on a full SP-network*

Before tackling a full blown Substitution/Permutation-network, the first thing to do is to add one more round to Figure 6.2, and understand how that changes the calculation of probabilities. There, to calculate the i-XOR of the second round (o-XOR of first round) we just had to look at the distribution table for S-Box of the first round, and count the number of inputs that yielded the desired output. Here it is slightly more complicated.

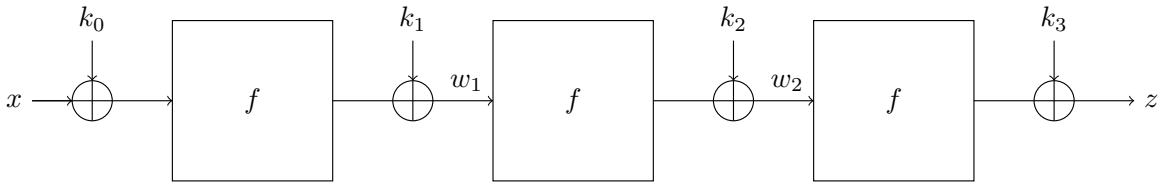


Figure 6.3.: Three round simple cipher.

If the attack is to focus on the last round again, then we must calculate the probability of Δw_2 having a specific value, from a given specific value of Δx. The standard way of doing this is to assume the differentials between two successive rounds to be *independent*, thus the probability of observing both is the product of their probability. This assumption is not strictly correct, but it works well in practice—more on this shall be said shortly.

To give a concrete example, suppose f is the S-Box of the Mini-AES, and consider again Table 4. In that table, built for fixed i-XOR 1011, for half of the input values the o-XOR is 0010. We want high probability differential characteristics, so we take $\Delta x = 1011$, which

means that $P(\Delta w_1 = 0010 \mid \Delta x = 1011) = 1/2$. We now construct the distribution table for i-XOR 0010, to see which o-XOR is more likely. The result is in Table 7.

o-XOR	Input values
0011	0000, 0010
0101	0001, 0011, 1000, 1010, 1100, 1110
0110	1001, 1011
0111	0101, 0111
1001	0100, 0110
1110	1101, 1111

Table 7.: o-XOR distribution for fixed i-XOR 0010.

Inspecting the table yields immediately that $P(\Delta w_2 = 0101 \mid \Delta w_1 = 0010) = 6/16 = 3/8$. Multiplying both values yields that $P(\Delta w_2 = 0101 \mid \Delta x = 1011) = 1/2 \times 3/8 = 3/16$. To see why we can multiply the probabilities, note that *for each* input value that, in the first S-Box, yields an o-XOR of 0010, there are 6 input values for the second S-Box. As there are 8 favourable input values for the first S-Box, the number of favourable cases for the differential is $8 \times 6 = 48$; by a similar reasoning the number of possible cases is $16 \times 16 = 256$, and $48/256 = 3/16$. This is where the assumption of independence manifests itself: in reality, the particular input bits to the second S-Box depend on the particular input bits to the first S-Box, as well as of the value of k_1, thus they are not independent. Nevertheless, because in differential cryptanalysis the input to the cipher is assumed random, and rounds keys are usually generated to be "as random as possible", the assumption of independence works well in practice.

Compare what would happen if f had been an APNL function: all the trails that lead to Δw_2 would have probability $2/16 \times 2/16 = 1/64$, a value *twelve times* lower! Thus we now have a high probability estimate for the differential $(\Delta x = 1011, \Delta w_2 = 0101)$, and so are now in conditions of using the algorithm of section 6.2.2 to recover the key.

The scenario with a full SP-network is very similar, the big difference being that there is more than S-Box per round, so the characteristics can be parallel, and in the last round, not all S-Boxes might be affected, which means the attack recovers key bits (instead of the full key). See Figure 6.4[8]; all S-Boxes implement the Mini-AES S-Box (the algorithm is the same if different S-Boxes were used, only more distribution tables would have to be constructed). To present the argument in its fullest generality, suppose that the diffusion is *non-linear*, meaning that an input difference that affects only S-Box S_{11} in the first round can generate input differences in (say) S-Boxes S_{21} and S_{22} (cf. (Heys, 2002, Figure 5, p. 24)). Suppose further that those second round differences propagate only to S_{33} in the third round (this is just to simplify the calculations).

8 Note that the last round has no diffusion; this is because as there are no more S-Boxes after that, no mixture is needed (it would not add anything to security).

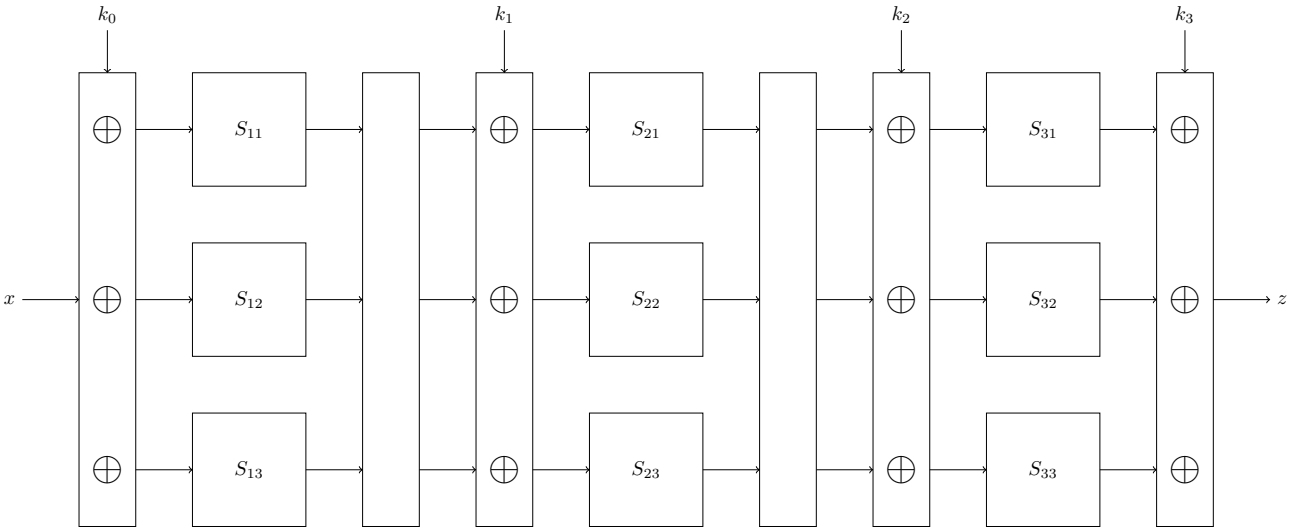


Figure 6.4.: Full SP-network. The blank rectangles represent diffusion.

We will use the following notation: Δx_{21} refers to the *input difference* of S_{21}; the *output difference* of for example S_{33} is denoted by Δz_{33}. The input difference to round 1 is ΔX_1 and the output difference is ΔZ_1. It follows that $\Delta x = \Delta X_1 = \Delta x_{11}\Delta x_{12}\Delta x_{13}$ and $\Delta z = \Delta Z_3 = \Delta z_{31}\Delta z_{32}\Delta z_{33}$, where juxtaposition means concatenation.

Let $\Delta X_1 = 1011\ 0000\ 0000$. Thus $\Delta z_{11} = 0010$ with probability $1/2$, and $\Delta z_{12} = \Delta z_{13} = 0000$. Suppose that through diffusion we have $\Delta X_2 = \Delta x_{21}\Delta x_{22}\Delta x_{23} = 1100\ 1110\ 0000$, i.e. only S_{21} and S_{22} are affected. We still have

$$P(\Delta X_2 = 1100\ 1110\ 0000 \mid \Delta X_1 = 1011\ 0000\ 0000) = 1/2$$

Now we construct the distribution tables for i-XOR's 1100 and 1110.

o-XOR	**Input values**
0001	0100, 1000
0100	0111, 1011
0101	0101, 1001
0110	0011, 1111
1011	0000, 1100
1101	0001, 1101, 0010, 1110, 0110, 1010

Table 8.: o-XOR distribution for fixed i-XOR 1100.

As done for the case of Figure 6.3, we want to choose highly likely o-XOR's, so we chose o-XOR 1101 for S_{21} and 1000 for S_{22}. With a similar reasoning as to what was done for that figure, we have

$$P(\Delta Z_2 = 1101\ 1000\ 0000 \mid \Delta X_2 = 1100\ 1110\ 0000) = 6/16 \times 6/16 = 9/64$$

o-XOR	Input values
0010	0111, 1001
0011	0001, 1111, 0101, 1011
0100	0100, 1010
1000	0010, 1100, 0011, 1101, 0110, 1000
1110	0000, 1110

Table 9.: o-XOR distribution for fixed i-XOR 1110.

And thus

$$P(\Delta Z_2 = 1101\ 1000\ 0000 \mid \Delta X_1 = 1011\ 0000\ 0000) = 1/2 \times 9/64 = 9/128$$

If the diffusion is such that this difference, when propagated to round 3, will only affect, say S_{33}, then we now have a characteristic which can be exploited, like in section 6.2.2, to recover the bits of k_3 that are XORed with this S-Box.

Note that the assumptions on the effects of the diffusion steps do not make the attack any less general: for example if the propagation of Z_1 to round 2 led to other differences, in/or other S-Boxes, then the only change would be to calculate the distribution tables for those differences and S-Boxes.

6.3 APPLICATIONS TO THE CIPHER

As explained in chapter 4, with the goal of maximising the cipher's resistance to differential cryptanalysis, its S-Boxes implement *almost perfect nonlinear* (APNL) functions. These are defined as follows Berger and Canteaut (2006).

Definition 6.3.1 (Almost Perfect Nonlinear Function)**.** *Let f be a bijective map from $\mathbb{F}_2^n$ to $\mathbb{F}_2^n$, and let*

$$\delta(f) = \max_{a \neq 0, b} \#\{x \in \mathbb{F}_2^n \mid f(x) + f(x+a) = b\}$$

If $\delta(f) = 2$, then f is said to be an almost perfect nonlinear function*.*

This is basically a more formal way of saying that for a given Boolean function f, it is APNL if and only if its distribution table, for any fixed i-XOR, has no more than two elements per row. This means that for any i-XOR, the probability of obtaining any o-XOR is always the same—and as explained in the previous section, this is the hardest scenario for differential cryptanalysis.

Currently, as section 4.4 states, the only criteria for the choice of the APNL function to be used was simplicity. This choice in practice ment choosing an exponent for a power function because, as mentioned in (Carlet, 2009, §3.1.7), *«Until recently, the only known examples of*

APN[L] functions were [...] power functions $x \to x^d$». Some of the possible values for d are the following (*ibid.*):

- $d = 2^n - 2$, n odd.
- $d = 2^i + 1$, $\gcd(i, n) = 1$ (Gold function).
- $d = 2^{2i} - 2^i + 1$, with $\gcd(i, n) = 1$ (Kasami function).
- $d = 2^{\frac{4n}{5}} + 2^{\frac{3n}{5}} + 2^{\frac{2n}{5}} + 2^{\frac{n}{5}} - 1$, with n divisible by 5.

The simplest choice is an exponent verifying the condition of the Gold functions, viz. $\gcd(i, n) = 1$—and the smallest possible i is 3 (which is coprime to both 17 and 19). This choice makes sense so far, because in what resistance to differential cryptanalysis is concerned, all APNL functions are equivalent (because they all have uniformly distributed o-XOR's). This is not necessarily the case when algebraic cryptanalysis is considered, as discussed in chapter 8.

The round function, as mentioned in chapter 4, uses APNL functions. To study its resistance to differential cryptanalysis, the first question asked was: if a Boolean function is constructed with parallel APNL S-Boxes, is it also APNL? And if not, are there any constraints that can be placed (e.g. in the diffusion mechanism) to make them so? Unfortunately, the answer to the first question is *no*, and furthermore there does not seem to exist an easy way to make the answer to the second question be yes. To see why, let f and f' be two APNL functions, and let x, a and b be an input, an input difference and an output difference for f; and respectively x', a' and b' for f'. Let $F = f \,||\, f$, where $||$ denotes concatenation (note that it takes precedence over addition). We have $F(x \,||\, x') = f(x) \,||\, f(x')$, $f(x) + f(x + a) = b$ and $f'(x') + f'(x' + a') = b'$. Furthermore,

$$\begin{aligned} & F(x \,||\, x' + a \,||\, a') \\ = {} & F\left[(x + a) \,||\, (x' + a')\right] \\ = {} & f(x + a) \,||\, f'(x' + a') \end{aligned}$$

Thus $F(x \,||\, x') + F(x \,||\, x' + a \,||\, a') = b \,||\, b'$. If F was APNL, then the *only* input pair to produce an output difference of $b \,||\, b'$ should be $(x \,||\, x',\ (x + a) \,||\, (x' + a'))$. But this is not the case, because the pair $(x \,||\, (x' + a'),\ (x + a) \,||\, x')$ also produces an output difference of $b \,||\, b'$, as can be readily verified. Adding more "small" functions, the resulting "big" function will still *not* be APNL: not only there exists the extra pair resulting from the first two functions; but as more functions are concatenated, more extra pairs can be created.

Reasoning by induction, it can be shown that this result generalises to the concatenation of any number of "small" APNL functions: the resulting "big" function will *not* be APNL.

And this seems to be an implicit characteristic of the construction; or to put it another way, it is not at all obvious how to add something to the scheme (diffusion for example) in such a way that the "big" function would be APNL.

So another strategy is required to analyse the round's resistance to differential cryptanalysis. This attack is harder to implement the more uniform the probability of the differential characteristics is. The probability of one such characteristic is given by a product like $\varepsilon_1\varepsilon_2\cdots\varepsilon_n$, where $0 < \varepsilon_i < 1$. This product is largest when the ε_i are each as large as possible, and the number of ε_i is as small as possible. Remember that for the 19-bit S-Box, each of the possible o-XOR's has a probability of $1/2^{18}$, and for the 17-bit ones $1/2^{16}$. Thus, assuming r rounds, the largest probability for a trail is $(1/2^{16})^r$. As for the smallest, because changes in the round input cause on average for half of the output bits to change, it seems reasonable to assume that a characteristic will affect, on average, seven S-Boxes. Accordingly the smallest probability for a characteristic is approximated as

$$\left[\left(\frac{1}{2^{18}}\right)\left(\frac{1}{2^{16}}\right)^7\right]^r$$

In SAGE, assuming 3 rounds, we obtain the following numbers:

```
1 sage: r=3
2 sage: (1/2.0^16)^r - ((1/2.0^18)*(1/2.0^16)^7)^r
3 3.55271367880050e-15
4 sage: (1/2.0^16)^r - ((1/2.0^18)*(1/2.0^16)^14)^r
5 3.55271367880050e-15
```

Note that even for a characteristic in which *all* fourteen 17-bit S-Boxes (plus the 19-bit one) are affected, the probability difference is so small that is it remains the same, up to the displayed precision—cf. lines 4–5 of the above snippet. As far as the author current understanding goes, this probability difference—with only three rounds—seems small enough to pronounce the design immune to differential cryptanalysis.

6.4 CONCLUSIONS

This chapter focuses on the only nonlinear component of the cipher: the S-Boxes. In particular, it focuses on a powerful attack, differential cryptanalysis, the gist of which consists of exploring a certain type of weakness in the design of said boxes.

Differential cryptanalysis is described at length, and that description serves as the basis from which it is argued that the current round design is—if not theoretically then at least in practice—immune to differential cryptanalysis.

This chapter ends with some remarks on the analysis of differential cryptanalysis it describes:

- The hypothetical functions of Tables 5 and 6 have been assumed to exist without proof. While correct, this criticism is ameliorated by two remarks: first, there seems to be nothing in the definition of both Boolean functions and the o-XOR distribution table that prevents functions with such tables from existing; and second, even if that were the case, those examples are only given as aid to better transmit the intuition behind the powerful attack of differential cryptanalysis.
- The construction of the characteristics is easier when using tables like (Heys, 2002, Table 7, p. 21); however this was not done here because it makes explaining the attack harder.

7

OTHER ATTACKS

The great thing about attackers is that there are so many to choose from!

Daniel J. Bernstein

After having, in the previous chapter, described at some length the nonlinear constructions used in the cipher, and the attack of differential cryptanalysis, in this section two other attacks are described: linear cryptanalysis and algebraic cryptanalysis. Note that both of these have variants—as does differential cryptanalysis—here we just sketch the attacks, to give a more detailed description of what has to be done in assessing the cipher's security.

7.1 LINEAR CRYPTANALYSIS

Unlike differential cryptanalysis, linear cryptanalysis is a known plaintext attack, first published in Matsui (1993). But just like its differential counterpart, it is also focused on the S-Boxes. The overview given here again draws on Heys (2002). Consider the variation of Figure 6.1 shown in Figure 7.1, and again suppose that f implements the Mini-AES S-Box (Phan (2002), 4 bits of input and output).

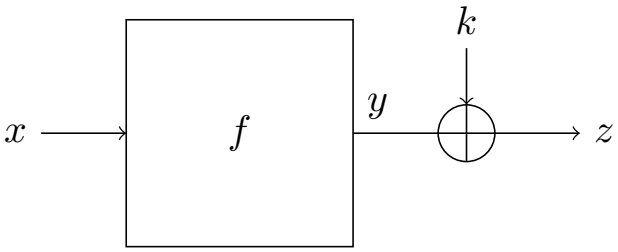


Figure 7.1.: Example target of linear cryptanalysis.

7.1. Linear Cryptanalysis

Let $x = x_1x_2x_3x_4$ and $y = y_1y_2y_3y_4$. The objective of linear cryptanalysis is to construct expressions Eq. 9, which are either very likely, *or* very unlikely (remember that + is addition in $\mathbb{F}_2$, i.e. XORing).

$$x_1 + x_2 + x_3 + x_4 + y_1 + y_2 + y_3 + y_4 = 0 \tag{9}$$

Some of the terms (but not all) can be 0. If all the bits were chosen randomly (independently), Eq. 9 would hold with probability exactly $1/2$. But if it holds with some probability $p_L \neq 1/2$, then that can be exploited to retrieve bits of the last round key. But first, the 0 on the right hand side must be explained. Linear cryptanalysis uses linear approximations between some input bits, some output bits, *and some key bits*. It is these key bits that are implicitly represented by the 0. To understand what this means, suppose the actual key bits the 0 is representing do XOR to 0. Then the *probability bias* of Eq. 9 is $p_L - 1/2$. Now suppose the converse is true, viz. those bits XOR to 1. If we replace 0 with 1 in Eq. 9, we obtain another equation which holds with probability $1 - p_L$. This means that the bias of the new equation is $(1 - p_L) - 1/2 = 1/2 - p_L$. I.e. the bias has only changed in sign, but its absolute value, $|p_L - 1/2|$ remains the same. Because linear cryptanalysis only depends on the absolute value of the deviation ("either very likely, *or* very unlikely"), we can fix the right hand side of Eq. 9 to 0, and work from there.

Suppose we have one such linear approximation for the "cipher" of Figure 7.1, with a large bias (ignore for the moment how would one *search* for such expressions). Because it is a known plaintext attack, suppose also we have n pairs (plaintext, ciphertext). To obtain the key, for each pair, decrypt the ciphertext with all possible keys, to obtain a possible output of the S-Box (y in the diagram), and then see if that value, together with the plaintext bits, verifies the linear approximation. If the probability of that expression was $1/2$, it would be expected that approximately $n/2$ of the n pairs would verify it, and the other half would not. But because of the large bias, there must be one key value for which the linear expression is either verified for a number distant (either above or below) from $n/2$, or fails for a number distant (idem.) from $n/2$. In other words, the possible key for which the hits is farthest from $n/2$, in absolute value, is with high probability the correct key.

Note that the particular case of when f is *linear*—which was impossibly difficult for differential cryptanalysis—here is, in fact, the *easiest* case: indeed it just means that there exists a trivial linear "approximation", which holds with probability 1 (i.e. it has a bias of $1/2$, the highest possible).

That was the simplest case. If we now proceed to a more complicated scenario, viz. having more than one round, we need a way to *combine* different linear approximations. This is done with the *Piling-Up Principle* ((Heys, 2002, §3.2), (Matsui, 1993, §5, Lemma 3)).

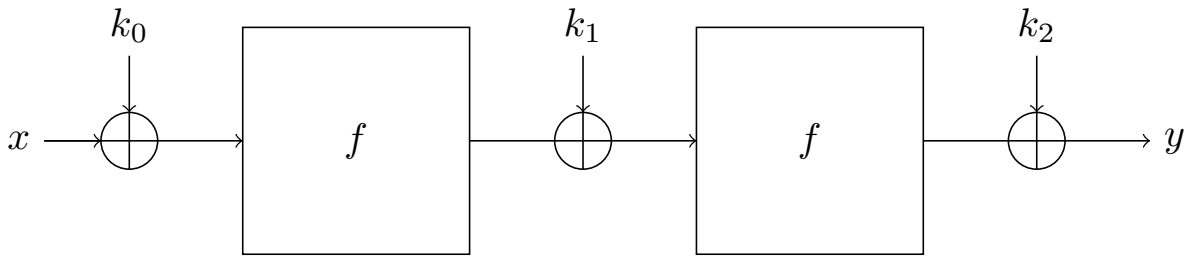


Figure 7.2.: Two round example of linear cryptanalysis.

Consider Figure 7.2 (f is again Mini-AES). We use the following notation: x is the plaintext, x_i is the bit in position i of the plaintext (starting at 0); similarly for y and y_i and the ciphertext. The input to the left S-Box is u_1, and $u_{1,i}$ is the bit in position i; the output of that S-Box is v_1, and a second subscript is similarly added to reference a particular bit within it. For the right S-Box, input is u_2, output v_2, and a similar observation for the second subscript applies. Second subscripts will also be used to refer to particular bits of the round keys.

For the first S-Box we will use $u_{1,3} = v_{1,1} + v_{1,2} + v_{1,3}$ with probability $14/16$. For the second S-Box, $u_{2,1} + u_{2,2} + u_{2,3} = v_{2,1} + v_{2,3}$, with probability $4/16$. For details on how these expressions were found, see A.5. To see how these expressions can be combined, notice that because of the properties of the XOR, we can re-write them as:

$$\begin{cases} u_{1,3} + v_{1,1} + v_{1,2} + v_{1,3} = 0, \text{ with probability } 14/16 \\ u_{2,1} + u_{2,2} + u_{2,3} + v_{2,1} + v_{2,3} = 0, \text{ with probability } 4/16 \end{cases} \tag{10}$$

Notice also that we can define *random variables* for (the left hand side of) each equation. So let X_1 be a random variable that represents the left hand side of the first equation, and similarly X_2 for the second one. Thus $P(X_1 = 0) = 14/16$ and $P(X_2 = 0) = 4/16$. Then we can ask what is the probability of $X_1 + X_2 = 0$ (this is useful because it will lead to a combination of both linear expressions, as shall be shortly seen). Remember the addition is done in $\mathbb{F}_2$, so that expression represents the probability of X_1 and X_2 having equal outputs. Calculating $P(X_1 + X_2)$ is done with the aforementioned Piling-Up Principle.

Theorem 7.1.1 (Piling-Up Principle)**.** *Let $X_1, \ldots, X_n$ be independent binary linear random variables. Furthermore, let $P(X_1 = 0) = 1/2 + \varepsilon_1, \ldots, P(X_n = 0) = 1/2 + \varepsilon_n$. Then we have*

$$P(X_1 + \cdots + X_n = 0) = 1/2 + 2^{n-1} \prod_{i=1}^{n} \varepsilon_i$$

Equivalently, $\varepsilon_{1,2,\ldots n} = 2^{n-1} \prod_{i=1}^{n} \varepsilon_i$, where $\varepsilon_{1,2,\ldots n}$ is the probability bias of the equation $X_1 + \cdots + X_n = 0$.

Going back to the linear expressions of Eq. 10, we have that $X_2 = u_{2,1} + u_{2,2} + u_{2,3} + v_{2,1} + v_{2,3}$, but this is not useful. The reason is that we want to relate the output of the second S-Box ($v_{2,i}$) with the input to *the cipher*. To do this, we can go "backwards", using bits from the fixed round key k_1 to re-write the input to the second S-Box like so:

$$X_2 = v_{1,1} + k_{1,1} + v_{1,2} + k_{1,2} + v_{1,3} + k_{1,3} + v_{2,1} + v_{2,3}$$

Remember $X_1 = u_{1,3} + v_{1,1} + v_{1,2} + v_{1,3}$. XORing both we get

$$X_1 + X_2 = u_{1,3} + k_{1,1} + k_{1,2} + k_{1,3} + v_{2,1} + v_{2,3}$$

This is close, but still not good enough, because we want an expression with the plaintext bits. But we can replace $u_{1,3}$ with $x_3 + k_{0,3}$ (again observe k_0 is fixed), obtaining

$$X_1 + X_2 = x_3 + k_{0,3} + k_{1,1} + k_{1,2} + k_{1,3} + v_{2,1} + v_{2,3} \tag{11}$$

Notice the substitutions do not affect the probabilities of the random variables, because they are, in a sense, "deterministic": they do not add or remove to the uncertainty of the expression. The Piling-Up Principle now yields that $P(X_1 + X_2) = 1/2 + 2 \times (14/16 - 1/2) \times (4/16 - 1/2) = 5/16$, with $\varepsilon_{1,2} = -3/16$. If the key bits in Eq. 11 XOR to 0, the equation $x_3 + v_{2,1} + v_{2,3} = 0$ has a probability bias of $-3/16$; if they XOR to 1 the bias is $3/16$. Thus we have a linear expression, relating input bits to output bits of the second S-Box; we can now proceed as initially to recover the corresponding bits of k_2—viz. $k_{2,1}$ and $k_{2,3}$.

To generalise this method to a fully-fledged SP-network, there is an additional step that is required, which is to track how the changed bits are spread by the diffusion mechanism, similarly to that was done in §6.2.3.

7.2 Algebraic Cryptanalysis and SAT

Algebraic cryptanalysis can be summed up as consisting of two steps: the first is to *«convert the cipher into a system of polynomial equations»*, and second to *«solve the system of equations and obtain from the solution the secret key of the cipher»* (Bard, 2007, §2.3.1). Rijndael's simplicity of design makes it (at least *without* hindsight) potentially vulnerable to this new type of attack. Representing a cipher as a set of equations can be done via Multivariate Quadratic equations (Courtois and Pieprzyk, 2002). But the fact that it is generally easier to solve linear systems, rather than quadratic ones, lead to the development of relinearization (Kipnis and Shamir, 1999), and subsequent improvements named XL (Courtois et al., 2000) and XSL (Courtois and Pieprzyk, 2002). As far as the author could research, the strength and efficiency of this attack is still unknown.

The above equations can be seen from a different perspective, viz. as a *satisfiability problem* (shortened to just SAT). Very roughly, the goal is, given a Boolean formula, either find a valoration that makes it true, or prove that none exists. What makes this approach promising is that the equations that mimic ciphers tend to have an enormous number of variables, which makes their manipulation cumbersome, to say the least. But by leveraging the power of modern SAT solvers, the problem becomes much more tractable (Massacci and Marraro, 2000). An example of the result of applying this technique was the discovery of several weak keys of a derivative of the IDEA cipher (Lafitte et al., 2014).

7.3 CONCLUSIONS

This chapter describes two other powerful attacks against symmetric ciphers, viz. linear and algebraic cryptanalysis. Perhaps the most surprising observation that can be made here is that, for a cipher intended to have a simple *algebraic* description, the role of *algebraic* attacks against it in the study that has been done has been minuscule indeed. Discussion of this—very pertinent—point shall be taken in chapter 8.

About linear cryptanalysis, it must be pointed out that a more rigorous analysis requires the use of spectral techniques—namely the *Walsh-Hadamard transform* and *correlation matrices*. Such a path was not pursued however, because doing so would required far more time than what was available for this work.

8

CONCLUSIONS AND FUTURE WORK

Schneier's Law: any person can invent a security system so clever that she or he can't think of how to break it.

Bruce Schneier et al.

The first principle is that you must not fool yourself—and you are the easiest person to fool.

Richard P. Feynman

8.1 CONCLUSIONS

The work heretofore presented failed to uncover any weakness that would make the proposed round structure unsuitable to be used as a basis for a realistic cipher. At least in theory: even a few iterations seem to make differential cryptanalysis all but infeasible, and the components used seem to lend themselves to efficient implementations. But even on realm of theory alone, there is still a sizeable amount of work to be done, as will be described in §8.2. The rest of this section is essentially an elaborated comment contrasting the initial goals and motivations with what was achieved and learned throughout the development of this work.

Besides pure research interest *per se*, one of the motivations was to take advantage of the algebraic descriptions to try to provide stronger rationale for the security of the final cryptographic scheme. It is too early to tell if this was achieved. But the path crossed so far suggests it is indeed possible to do better than to design something that seems "complicated enough" to be secure. That being said, the author cannot "shake the feeling" that for all its power, mathematics can only take one so far. For example, on the choice of the round constants, why a Hamming weight of 128? Can there be better constants (stronger diffusion)?

For more details on Schneier's Law, see https://www.schneier.com/blog/archives/2011/04/schneiers_law.html

With greater Hamming weight? Or smaller? Or in the layout of the S-Boxes, is any difference (i.e. does the layout matter at all)? If so, then how to establish which one is the more suitable? Currently these and other questions have been answered with a combination of intuition and æsthetics. Ascertaining just how far mathematics can take us in answering these and other questions is perhaps a suitable way to characterise the direction towards which future work must be steered.

8.2 Prospect for future work

In more concrete steps than those outlined in the previous paragraph, the two more immediate next steps are to see how the construction resists linear cryptanalysis and algebraic cryptanalysis. Concerning the former, the theory of Walsh-Hadamard transforms must be brought to bear on the cipher, to see how it holds against it. Regarding the latter, it is a much newer attack, that did not even exist when Rijndael was proposed. Therefore it is still essentially an open question of how to design a cipher that is resistant to it. The most obvious way of attacking this problem seems to be to focus on the APNL functions: while any such function is equally (well) resistant to differential cryptanalysis, such may not be the case when algebraic attacks are considered. Which choice of function makes relinearization harder, or yields a greater increase in the number of variables of the resulting system, are the questions that must be tackled next.

After that, two related steps: on the one hand, see if the rounds symmetric layout can lead to an attack, and if so, how to defend against it; related to the latter, on the other hand, is the design of the key schedule, which can be used to compensate for the round's symmetry. Following that, study if are any improvements possible to the round constants. In particular, if there are more suitable Hamming weights for the bitstrings. Also analyse if, with the current constants, there exists some bias in the number of bits that are flipped in the output[1]. It would also be of interest to see what insights the *Wide Trail Strategy* (Daemen and Rijmen, 2002, §9) can provide, when applied to the current design.

Lastly, investigate implementation-related issues, namely the most pressing one: can a construction with 257 bits be implemented efficiently on current architectures? And while we are on the subject of implementation, an attack that depends crucially on the implementation (of the cipher in general and of the S-Boxes in particular) is Daniel Bernstein's *cache timing attack* Bernstein (2004). How—if at all possible—to mitigate that attack, in a cipher which security depends strongly on its S-Boxes, is a question also worthwhile looking into.

1 Remember that with the current design, after ten thousand runs, and with a varying number of input changes, the output changed, on average, 128 bits. For a truly random process one would hope to see 128 and 129 bit changes, approximately the same number of times.

A

SAGE MATHEMATICS

SAGE ("System for Algebra and Geometry Experimentation", also known as SageMath) *«is mathematical software with features covering many aspects of mathematics, including algebra, combinatorics, numerical mathematics, number theory, and calculus»* Wikipedia (d). It is a free and open software system (released under the GNU General Public License) that is built *«on top of many existing open-source packages: NumPy, SciPy, matplotlib, Sympy, Maxima, GAP, FLINT, R and many more»* and that provides *«Access their combined power through a common, Python-based language»* (Team, b). Thus SAGE can be seen as providing a superset of the Python programming language. It should be stressed that "Python" here refers to *version 2* of the language (Team, a), with which the reader is assumed to be familiarised.

In this chapter a very quick introduction is given to SAGE's functionality that this most relies on.

A.1 TERMINOLOGY AND PYTHON FACTS

Lists are mutable tuples. They are delimited by square brackets, and its elements separated by commas. Indexing a list `L` like `L[a:b]` means the sub-list that begins with element `L[a]`, and ends with the element *immediately before* `L[b]`. To select a sub-list of 3 elements of `L`, beginning with `L[a]`, you do `L[a:a+3]`. Adding (`+`) two lists concatenates their elements. E.g. `[a, b]+[c, d]` yields the list `[a, b, c, d]`.

Method, function and procedure are used interchangeably. The same applies to list and array.

A.2 BASICS

In the present the SAGE system has been primarily useful as a way to do calculations over finite fields, in particular Galois fields. They are defined by `GF(2)` or `FiniteField(2)`, for the finite field with two elements. The ring of polynomials with coefficients in $\mathbb{F}_2$, with indeterminate X, is defined as:

```
GF2=GF(2)
P_GF2.<X> = PolynomialRing(GF2)
```

To define a ring with a given modulus, do:

```
big_modulus = X^257+1
R.<x> = P_GF2.quotient_ring(big_modulus)
```

The polynomials that belong to R will have indeterminate x. To compute the inverse (in R) of one of such polynomial, say `poly`, do (note that the inverse might not exist if the ring's modulus is not irreducible):

```
inv_poly = P_GF2(list(poly)).inverse_mod(big_modulus)
```

This also shows that (at least in some cases) you can change the indeterminate of a polynomial: `P_GF2(list(poly))` returns `poly`, but with the indeterminate changed from `x` to `X`, the indeterminate of `P_GF2`. The `list` function, applied to a polynomial returns a list of its coefficients (including the zeros), the left-most being the independent term. It can also be used as `poly.list()`.

But the syntax can vary, which is something of a double-edged sword. For example, Code Snippet A.1 shows one way of testing if the small moduli of the CRT isomorphism are irreducible (cf. §5).

A.3 THE CRT MATRIX

In this section we will walk through the code necessary to construct the CRT matrix (Eq. 8, §5.2). The full code is available in the file `crtmatrix_test.sage`. Before we begin, the reader might wish to skim over §5.2 again, as we will associate the code to the corresponding steps, laid therein, that result in the CRT matrix. The fundamental excerpt is shown in snippet A.2.

We will work from the bottom up. The `fill257` method receives a polynomial object and straightforwardly returns its coefficient list (leftmost entry is the independent term)—padded with zeros if degree is less than 256, so that the returned list is always of length 257.

The `mult_and_expand` method receives a polynomial, and multiplies it by x^0, x^1, …, x^{15} (modulo $x^{257}+1$). It places each of these polynomials in an array, and then the `fill257` function replaces each of them with (a sub-array) of its respective coefficients. Thus, it returns an array of length 16, in which each of the entries is itself an array, of length 257.

```
sage: R.<x> = PolynomialRing(GF(2),'x')
sage: big_modulus = x^257+1
sage: F = factor(big_modulus)
sage: small_moduli = map(lambda (a,b) : a, list(F))
sage: for i in small_moduli: i.is_irreducible()
```

Code Snippet A.1: Using SAGE to show the small moduli are irreducible (it outputs `True` 17 times).

```
def get_crt_matrix():
  global b_crt
  BB = reduce(lambda a,b: a + b, [mult_and_expand(b_crt[i+1]) for i in range(16)],
              [fill257(b_crt[0])])
  CRT = matrix(BB)
  return CRT

def mult_and_expand(e):
  xpow16  = [x^k for k in range(16)]
  u = map(lambda a: (a*e) % big_modulus, xpow16)
  u = map(fill257,u)
  return u

def fill257(p):
  d = 256 - p.degree()
  c = p.coeffs()
  if d>0:
    return c + [0 for k in range(d)]
  return c
```

Code Snippet A.2: Using SAGE to construct the CRT matrix. File `crtmatrix_test.sage`.

Now we get to the `get_crt_matrix`, where the "bulk" of the logic is. The list comprehension `[mult_and_expand(b_crt[i+1]) for i in range(16)]` produces an array of length 16, where each of its entries are the length 16 array of arrays returned by `mult_and_expand`, described in the previous paragraph. Let's deconstruct this expression.

For `i=0`, we have `mult_and_expand(b_crt[1])`, which yields in a first step a list of polynomials like so: $[e_1, xe_1, x^2e_1, \ldots, x^{15}e_1]$, and on a second step another list obtained by replacing each of the polynomials with their respective coefficients. I.e. we obtain: $[[e_{1,0}, \cdots, e_{1,256}], [(xe_1)_0, \cdots, (xe_1)_{256}], \ldots, [(x^{15}e_1)_0, \cdots, (x^{15}e_1)_{256}]]$. This is the first item of the list comprehension shown above. Notice that it already resembles lines 2 through 17 of the *CRT* matrix (p. 38). Iterating the list comprehension for $e_2, \ldots, e_{16}$, we obtain similar results, which will be the remaining elements of the array resulting from the list comprehension. The (list) adding operation done by the `lambda` function will concatenate all such elements, yielding an array of length 256, in which each entry is an array of length 257, which correspond to CRT matrix lines 2–257. Line 1 is given by `[fill257(b_crt[0])]`, which is prepended to the result by Python's `reduce` function.

To test this construction, we generate a random 257 bitstring, which we shall interpret as containing the sequence of (the coefficients of the) polynomials in the small rings, then use the CRT matrix to compute the corresponding polynomial in the big ring R, and test that this is indeed the correct one by dividing it by the different moduli, and comparing the remainders with the polynomials in the small rings that we had randomly generated initially. The relevant code is in snippet A.3.

```
a = fill257(R.random_element(256))
big_coeffs = vector(a) * crt_matrix
big = coeffs2poly(list(big_coeffs))
small_polys = [[ a[0] ]]
for i in range(1, 256, 16):
  small_polys = small_polys + [ a[i:i+16] ]

if big % small_moduli[0] == coeffs2poly(small_polys[0]):
  print("True 0!")
if big % small_moduli[1] == coeffs2poly(small_polys[1]):
  print("True 1!")
```

Code Snippet A.3: Using SAGE to test the CRT matrix. File `crtmatrix_test.sage`.

A.4 Simple differential cryptanalysis

The simplest case of the differential cryptanalysis attack—the one against a construction like Figure 6.1—is implemented in file `dc.sage`. Although the implementation is generic, because it receives as a parameter the function to which the attack is to be applied, in this case there were two implemented functions: the S-Boxes of AES and the Gold function. The first step in this attack is to construct a distribution table for a fixed i-XOR (because of the simplicity of the construction any value will do, we can just choose randomly). The distribution table is created as a Python dictionary—`d[o-XOR]=[list of inputs]`—in the method `create_distribution()`. The code in Snippet A.4 simply generates random values, the distribution for those values' i-XOR, and from that a set of possible keys (`crypto_analyser`). Then the process is iterated, removing wrong keys from that set, until there is only one left (`cryptanalyse()`). Remember that differential cryptanalysis is a chosen plaintext, so an `oracle()` is required, to produce the (plaintext, ciphertext) pairs chosen by the attacker.

Note that it can correctly determine the key the oracle is using, when the S-Box function `F` is either the AES one, or the Gold one. But it completely fails when it is linear; and moreover, the set of possible keys, after cryptanalysis, in this latter case, is the set of *all* 256 bitstrings of length 8, just as predicted in §6.2.1.

A.5 Simple (partial) linear cryptanalysis

Using Mini-AES as an example, the function `create_linexp_table()`, in file `miniAES.sage`, illustrates one process of finding linear expressions for that S-Box. If its input and output are denoted by $x_1x_2x_3x_4$ and $y_1y_2y_3y_4$ respectively, then the goal is calculate the probability of expressions like the following, where some terms may be 0. See §7.1 for details.

$$x_1 + x_2 + x_3 + x_4 + y_1 + y_2 + y_3 + y_4 = 0 \tag{12}$$

The number of possible combinations of the inputs bits is given by the *powerset* of $\{1,2,3,4\}$, minus the empty set; and similarly for the output bits. Because the implementation will use arrays, which are indexed starting at 0, the script actually uses the powerset of $\{0,1,2,3\}$. A (double) dictionary `d` is used to store the different linear expressions, as well as a counter for how often they are verified. The key is a string containing the indexes of the nonzero bits (in our particular scenario this causes no ambiguity). Thus `d=['01']['23']` corresponds to $x_1 + x_2 + y_3 + y_4 = 0$. For each entry in the dictionary a counter is kept, that is incremented whenever there is an input/output pair which verifies the expression. The algorithm simply runs over all inputs, and for each tests all expressions. See Code Snippet A.5.

```
# set here the function to be used by oracle
# eg square, gold_f9, sub_bytes
F = square

def crypto_analyser(cipher):
  # generate two input
  input1 = "{:0>8b}".format(randint(0, 255))
  input2 = "{:0>8b}".format(randint(0, 255))
  i_xor = xor(input1, input2)

  output1 = cipher(input1)
  output2 = cipher(input2)
  o_xor = xor(output1, output2)

  d = create_distribution(F, i_xor)

  possible_inputs = d[o_xor]
  possible_keys = [ xor(input1, elem) for elem in possible_inputs ]

  return possible_keys

def oracle(byte):
  key = "10101101"
  return F(xor(byte, key))

def cryptanalyse():
  keys = crypto_analyser(oracle)
  aux = crypto_analyser(oracle)
  for i in keys:
    if aux.count(i) == 0:
      keys.remove(i)

  print(keys)
  print(len(keys))
```

Code Snippet A.4: Using SAGE to do differential cryptanalysis.

```
def create_linexp_table(f):
  # powerset minus empty set
  isubs = [x for x in powerset([0 .. 3])][1:]
  osubs = [x for x in powerset([0 .. 3])][1:]

  d = {}
  for c in range(16):
    x = "{:0>4b}".format(c)
    y = f(x)
    for i in isubs:
      if not int_array2str(i) in d:
        d[int_array2str(i)] = {}
      for j in osubs:
        if not int_array2str(j) in d[int_array2str(i)]:
          d[int_array2str(i)][int_array2str(j)] = 0

        isum = reduce(operator.add, [int(x[idx]) for idx in i])
        osum = reduce(operator.add, [int(y[idx]) for idx in j])

        if ((isum + osum) % 2)==0:
          d[int_array2str(i)][int_array2str(j)] = \
              d[int_array2str(i)][int_array2str(j)] + 1
  return d
```

Code Snippet A.5: Using SAGE to do compute linear expressions used for linear cryptanalysis.

B

CODES, MATRICES AND DIFFUSION

B.1 CODING THEORY

The acronym "MDS" originates from coding theory, so in order to explain what an "MDS matrix" is, we first need to introduce some notions from said theory. The very brief exposé given here is based on Huffman and Pless (2003)—to which the reader is also referred to for (much) more details.

Definition B.1.1. *Let the set* $A = (a_1, a_2, \ldots, a_q)$ *of size* q*, be the* code alphabet*; its elements are the* code symbols*. A* q-ary block code *of length* n *over* A *is a non-empty set* C *of vectors (of length* n*) of symbols of* A*. Its elements are called* codewords*. The* size *of the code is the size of* C*. A code of length* n *and size* M *is called a* (q-ary) (n, M)-code*.*

An important metric is *distance*. One of the most widely used metrics is the *Hamming distance*, denoted hd, and which we define below, together with the related concept of *Hamming weight* (denoted hw).

Definition B.1.2 (Hamming distance)**.** *Let* $\mathbf{x}$ *and* $\mathbf{y}$ *be two codewords of some code* C*. The* Hamming distance *is defined as the number of different components in* $\mathbf{x}$ *and* $\mathbf{y}$*.*

Definition B.1.3 (Hamming weight)**.** *Let* $\mathbf{x}$ *be a codeword of some code* C*. The* Hamming weight *is defined as* $hd(\mathbf{x}, \mathbf{0})$*.*

In a given code C, if d is the *minimum distance* between any two codewords in C, then that code is often denoted as an (n, M, d)-code. The minimum distance is denoted as $d(C)$.

Usually the code alphabet is some finite field $\mathbb{F}_q$. If the vectors of size n over this field have (or can be endowed) with the structure of a *vector space*, then any subset that constitutes a subspace is called a *linear code*. These are denoted $[n, k]$-linear codes, where k is the *dimension* of the subspace (note that this means the number of codewords is q^k). If the minimum distance is d, then we say the code is a $[n, k, d]$-linear code. If C constitutes a linear code of dimension k (and length n), then its *dual code*, denoted $C^\perp$, is the linear code that corresponds to the orthogonal complement of C. From well-known facts from linear algebra immediately stems the property that $dim(C^\perp) = n - k$.

Let C be a linear code of dimension k in some vector space $\mathbb{F}_q^n$—i.e. the set of vectors of length n, which components are elements of $\mathbb{F}_q$. Given that C is a subspace of $\mathbb{F}_q^n$, we can compute a *basis* for it, say $(r_1, \ldots, r_k)$. We can arrange them into matrix form, each vector forming a row. We then obtain a $k \times n$ matrix. Let G be that matrix. To *encode* a vector $\mathbf{u}$, we compute $\mathbf{v} = \mathbf{u}G$. Note that $\mathbf{u}$ is a row vector of size k, and $\mathbf{v}$ is a row vector of size n. The process of encoding can be visualised by thinking of $\mathbf{u}$ as a vector expressed in the coordinate system of the basis of the subspace, and of $\mathbf{v}$ as the same vector, but now expressed in the coordinates of the underlying vector space $\mathbb{F}_q^n$.[1] To decode, one reverses the process. The matrix G is called the *generator matrix*. If this matrix is written in the form $[I_k|X]$, where I_k is the $k \times k$ identity matrix, then the generator is said to be in *standard form*.

We now arrive to the most important result in this brief tour of coding theory: the *Singleton bound* (for proofs and further details, see (Huffman and Pless, 2003, §2.4)).

Theorem B.1.4 (**Singleton bound**). *For any $[n, k, d]$-linear code over $\mathbb{F}_q^n$, we have: $k + d \leq n + 1$.*

Definition B.1.5 (**MDS code**). *If a linear code verifies $k + d = n + 1$, we say it is a* Maximum Distance Separable *(MDS) code.*

Section B.4 briefly mentions a somewhat unintuitive consequence the definition of MDS matrices in $\mathbb{F}_2$. For that however, we first need to introduce the notion of *trivial* codes—and for that in turn, a notion of when two error-correcting codes are *equivalent* is required.

Definition B.1.6. *Two* binary (n, M)*-codes are* equivalent *if one can be obtained from the other via a sequence of operations of the following two types:*[2]

1. *permutation of the digits of the codewords;*
2. *multiplication of the digits appearing in a fixed position by a non-zero scalar.*

It is possible for a linear code to not have a generator matrix in standard form. But there always exists another linear code, equivalent to it and that does possess a standard form generator (*ibid.*, §1.6.2).

Definition B.1.7. *An MDS code over $\mathbb{F}_q^n$ is* trivial *iff it verifies one of the following conditions:*

1. $C = \mathbb{F}_q^n$;

1 In effect, encoding consists in the addition of some redundancy which, in the event of transmission errors, can be used recover the original information, without re-transmission.

2 In (Huffman and Pless, 2003, §1.6, 1.7), three forms of code equivalence are defined: *permutation equivalence*, *monomial equivalence*, and ("just") *equivalence*. For binary codes, all these notions amount to the same, hence we just define "equivalence".

2. *C is equivalent to the code generated by $\mathbf{1} = (1, \ldots, 1)$;*

3. *C is equivalent to the dual of the code generated by $\mathbf{1}$.*

The following results are also needed in §B.4 (cf. Theorem 2.4.3, p. 71, in Huffman and Pless (2003)).

Theorem B.1.8. *Let G be the $k \times n$ generator matrix of an $[n, k, d]$-linear code C. Then C is MDS iff every set of k columns of G is linearly independent.*

Theorem B.1.9. *Let C be a linear code over $\mathbb{F}_q$. Then $d(C)$ is equal to the smallest non-zero weight of the codewords of C.*

Proof. By the definitions of Hamming weight and Hamming distance, and minimum distance, we have: $\forall\, x, y, x \neq y : wt(x - y) = hd(x, y) \geq d(C)$. But if the code is linear, and of $x \neq y$, then $x - y$ is always a (non-zero) codeword of C. Thus the minimum distance is the smallest of such non-zero codewords. ■

B.2 MATRICES AND DIFFUSION

As is so often the case in the early stages of (at least the author's) research, one of the first stops was Wikipedia (b). An MDS matrix is defined as:

> Technically, an $m \times n$ matrix A over a finite field K is an MDS matrix if it is the transformation matrix of a linear transformation $f(x) = Ax$ from K^n to K^m such that no two different $(m + n)$-tuples of the form $(x, f(x))$ coincide in n or more components. Equivalently, the set of all $(m + n)$-tuples $(x, f(x))$ is an MDS code, i.e. a linear code that reaches the Singleton bound.

We are actually given two equivalent definitions here. To better understand the notions at hand, we will show that equivalence. First we have the definition of a matrix being MDS: "no two different $(m + n)$-tuples of the form $(x, f(x))$ coincide in n or more components". We can rephrase this as "all two $(m + n)$-tuples (i.e. all pairs of tuples) coincide in less than n components", which implies that "all pairs of tuples **differ** in $m + 1$ or more components". This implies that the *minimum* **Hamming distance** between any two tuples $(x, f(x))$ is $d = m + 1$. If each tuple is seen as a codeword, then given that that code has dimension n (see §B.3) and codeword size $n + m$, it is trivial to observe that the Singleton bound is verified (with equality)—thus the code is MDS, which is in agreement with the second definition.

The second definition borrows the MDS concept from coding theory: a matrix is MDS if the set of all $(m + n)$-tuples of the form $(x, f(x))$ constitute an MDS code. We show in §B.3 that that set forms indeed a linear code, with dimension n. If that code is MDS, then the

Singleton bound (Theorem §B.1.4: for any linear code $[\aleph, k, d]$[3] we have $k + d \leq \aleph + 1$), is verified with equality: $k + d = \aleph + 1$. Given that $k = n$ and $\aleph = n + m$, we obtain $d = m + 1$—which is consistent with our first definition.

Thus we conclude that the two given definitions are indeed equivalent, and furthermore that a matrix is MDS if it can be used to form the codewords of a linear $[n + m, n, m + 1]$ code.

So far we have the following two (equivalent) definitions, for an MDS matrix: either the set of all $(n + m)$-tuples of the form $(x, f(x))$ form an MDS code ($[n + m, n, m + 1]$), or for any two such tuples, they have to coincide in less than n components (i.e. they have to differ in at least $m + 1$).

On the other hand, a (not necessarily linear) function is a *multipermutation* if (Junod and Vaudenay, 2004):

> A diffusion function f from K^n to K^m is a multipermutation if for any $x_1, \ldots, x_p \in K$ and any integer r with $1 \leq r \leq n$, modifying r input values on $f(x_1, \ldots, x_n)$ results in modifying at least $m - r + 1$ output values.

Suppose the function is linear, and that its action can be represented by some $m \times n$ matrix[4]. If its matrix is MDS, that means that the minimum distance between codewords of the form $(x, f(x))$ is $m + 1$, and thus modifying r bits in x has to modify at least $m + 1 - r$ bits in $f(x)$—i.e. it is a multipermutation. Conversely, if flipping r bits in x flips at least $m + 1 - r$ in $f(x)$, then how many bits remain unflipped in the $(n + m)$-tuple? Well, at most $(n - r) + (m - (m + 1 - r)) = n - 1 < n$. In other words, any two $(n + m)$-vectors coincide in less than n components—which means its matrix is MDS. So we conclude that if f is a linear transformation that can be represented as a matrix, it is a multipermutation iff its corresponding matrix is MDS.

B.3 A basis for the $(x, f(x))$ codewords

Let $\mathbb{K}$ be a field, and let $f : \mathbb{K}^n \to \mathbb{K}^m$ be the linear transformation associated to an $m \times n$ matrix A. Here we show that the set of codewords of length $n + m$ and of the form $(x, f(x))$ constitutes a linear code of dimension n. We do so by constructing the generator matrix for it, in standard form.

3 The Hebrew letter *aleph* is used to denote the length of the codewords here, instead of n, so as to avoid confusing it with the n that comes Wikipedia's definition.

4 While multiplication by a matrix is always a linear transformation, the converse is false—there are linear transformations that cannot be expressed via a matrix multiplication. The reasoning herein set forth (and its respective conclusion) are only applicable if that is *not* the case—i.e. if the linear transformation to which it applies *can* be represented as a matrix.

Consider the standard base $\langle e_1, \ldots, e_n \rangle$ for $\mathbb{K}^n$. Construct an $n \times (n+m)$ matrix G like so: the first row is the tuple $(e_1, f(e_1))$, the second row if $(e_2, f(e_2))$, and so on until row n. Let $(x, f(x))$ be an element of the code we are considering. x can be written as $x = \lambda_1 e_1 + \cdots + \lambda_n e_n$, for some $\lambda_1, \ldots, \lambda_n$. Then if $\mathbf{u} = (\lambda_1, \ldots, \lambda_n)$, and taking into account the linearity of f, we have $\mathbf{u}G = ((\lambda_1 e_1, \lambda_1 f(e_1)) + \cdots + (\lambda_n e_n, \lambda_n f(e_n)) = (x, f(x))$. This shows that every $(n+m)$-tuple of the form $(x, f(x))$ can be written as a linear combination of the n vectors that constitute G's rows—and in conjunction with the fact that, due to the way G is constructed, its n rows are linearly independent, we can conclude that G's rows form a basis for the linear code at hand, and that said code has dimension n, as was to be proved.

As a final remark, note that the generator matrix just constructed provides us with a (conceptually, if not computationally) simple test to see whether a given matrix is MDS or not. Let A be the matrix that corresponds to the linear transformation f. Note that $f(e_1)$, when interpreted as a row vector (as we did above), corresponds to the first column of A, transposed. The same is true for $f(e_2), \ldots, f(e_n)$. That means $G = [I_n | A^T]$. Theorem B.1.8 then allows us to conclude that A is an MDS matrix iff every subset of n columns from G is linearly independent. This is precisely the same criteria stated in Wikipedia (b).[5]

B.4 MDS matrices in $\mathbb{F}_2$

The concept of MDS matrix has a somewhat unexpected consequence when working with matrices of bits (i.e. in $\mathbb{F}_2$). Consider the identity matrix I_k, for some positive integer k greater than 1. Its k columns are obviously linearly independent, and its rows form the basis for the code $(\mathbb{F}_2)^k$. It is an MDS code, and according to Definition B.1.7 (1), it is also a trivial one.

Now consider we try to add (meaning "append") one k-length bit column vector $\mathbf{c}$ to the right of I_k. This is still a generator matrix (in standard form $[I|X]$) of some $[n,k]$-linear code (in this case, $n = k+1$). If $\mathbf{c}$ has only one zero entry, then it can be written as the sum of precisely $k-1$ columns of I_k—i.e. all except the one with a 1 in the same position of the 0 in $\mathbf{c}$. Then according to Theorem B.1.8, the new extended matrix cannot correspond to an MDS code (it has a set of k columns which is not linearly independent—viz. the $k-1$ columns from I_k and $\mathbf{c}$). It is straightforward to notice that the same is true whenever $\mathbf{c}$ has any zero entries at all.

Given that we are working in $\mathbb{F}_2$, the only remaining possibility is for $\mathbf{c}$ to be a vector in which all entries are 1. It is again straightforward to notice that any subset of k columns of $[I_k|\mathbf{c}]$ is linearly independent, which means its rows are again the basis for an MDS code. But this as far as we can go. If we add another column—*any* column—to this $(k+1)$-column

5 An application of this algorithm to the matrices of AES and Twofish (Schneier et al., 1999) is coded in `twofish_aes_MDS.sage`—both are MDS.

matrix, then either it has one or more zero entries, in which case it is a linear combination of at most $k-1$ rows of I_k, or it has no zero entries, which means it is equal to the previously added column (i.e. $\mathbf{c}$), and thus that it is trivial to find a subset of k linearly dependent columns. In either case the corresponding linear code will not be MDS.

Thus we conclude that for $k \geq 2$, the only possible MDS codes are the ones equivalent to $[I_k|\mathbf{1}_k]$, with $\mathbf{1}_k = [1,\ldots,1]^T$. If $k = 1$, note that for a linear code of codeword size n to be MDS, we must have $d = n$. Given that for linear codes, the minimum distance is equal to the smallest non-zero Hamming weight of its codewords (Theorem B.1.9), the codeword which is nearest to zero is the all 1 codeword. There is only one such code, which is composed of two codewords: the all 0 codeword, and the all 1 codeword. Furthermore, for any n, and setting $k = n-1$, simple algebraic manipulations show that the codes corresponding to the generator matrices $[I_k|\mathbf{1}_k]$ and $[1,\ldots,1]_{1\times n}$ are the dual codes of one another. Thus we can strengthen our initial conclusion, generalising it to state that in $\mathbb{F}_2$, the only MDS codes are the trivial ones (as per Definition B.1.7)[6]. The implication for MDS matrices is obvious.

6 Our reasoning in this section assumes that we can always write a generator matrix in standard form. Although this is not the case, this limitation can be overcome by producing an equivalent code which does possess a standard generator. As the definition of trivial codes already uses that of equivalent ones, our conclusion remains valid.

BIBLIOGRAPHY

Ross Anderson, Eli Biham, and Lars Knudsen. Serpent: A proposal for the advanced encryption standard. In *in First Advanced Encryption Standard (AES) Conference*, 1998.

Ross J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley Publishing, 2 edition, 2008. ISBN 9780470068526. URL http://www.cl.cam.ac.uk/~rja14/book.html.

Ross J. Anderson and Eli Biham. Two practical and provably secure block ciphers: Bears and lion. In Dieter Gollmann, editor, *FSE*, volume 1039 of *Lecture Notes in Computer Science*, pages 113–120. Springer, 1996. ISBN 3-540-60865-6. URL http://dblp.uni-trier.de/db/conf/fse/fse96.html#AndersonB96a.

Gregory V. Bard. *Algorithms for Solving Linear and Polynomial Systems of Equations over Finite Fields with Applications to Cryptanalysis*. PhD thesis, Department of Mathematics, University of Maryland, USA, 2007. URL http://www.sagemath.org/files/thesis/bard-thesis-2007.pdf.

Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In Tatsuaki Okamoto, editor, *Advances in Cryptology — ASIACRYPT 2000*, volume 1976 of *Lecture Notes in Computer Science*, pages 531–545. Springer Berlin Heidelberg, 2000. ISBN 978-3-540-41404-9. doi: 10.1007/3-540-44448-3_41. URL http://dx.doi.org/10.1007/3-540-44448-3_41.

Steven M. Bellovin. Problem Areas for the IP Security Protocols. In *Proceedings of the 6th Conference on USENIX Security Symposium, Focusing on Applications of Cryptography - Volume 6*, SSYM'96, Berkeley, CA, USA, 1996. USENIX Association. URL https://www.cs.columbia.edu/~smb/papers/badesp.pdf.

TP Berger and Anne Canteaut. On almost perfect nonlinear functions over Fn2. *IEEE Transactions on Information Theory*, 52(9):4160–4170, 2006. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1683931.

Daniel J. Bernstein. Cache-timing attacks on AES, 2004. URL http://cr.yp.to/papers.html#cachetiming.

Eli Biham and Adi Shamir. Differential cryptanalysis of des-like cryptosystems. In *CRYPTO'91*, 1991.

C. Carlet. Vector boolean functions for cryptography. In Yves Crama and Peter Hammer, editors, *Boolean Methods and Models*. Cambridge University Press, United Kingdom, 2009. URL http://www.math.univ-paris13.fr/~carlet/chap-vectorial-fcts-corr.pdf.

Nicolas Courtois and Josef Pieprzyk. Cryptanalysis of block ciphers with overdefined systems of equations. In *Advances in Cryptology - ASIACRYPT 2002, 8th International Conference on the Theory and Application of Cryptology and Information Security, Queenstown, New Zealand, December 1-5, 2002, Proceedings*, volume 2501 of *Lecture Notes in Computer Science*, pages 267–287. Springer, 2002. doi: 10.1007/3-540-36178-2_17. URL http://www.iacr.org/cryptodb/archive/2002/ASIACRYPT/19/19.pdf.

Nicolas Courtois, Alexander Klimov, Jacques Patarin, and Adi Shamir. Efficient algorithms for solving overdefined systems of multivariate polynomial equations. In *Advances in Cryptology - EUROCRYPT 2000, International Conference on the Theory and Application of Cryptographic Techniques, Bruges, Belgium, May 14-18, 2000, Proceeding*, volume 1807 of *Lecture Notes in Computer Science*, pages 392–407. Springer, 2000. doi: 10.1007/3-540-45539-6_27.

Joan Daemen and Vincent Rijmen. *The Design of Rijndael*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002. ISBN 3540425802.

Yves Edel and Alexander Pott. On designs and multiplier groups constructed from Almost Perfect Nonlinear functions. *Cryptography and Coding*, 2009. URL http://link.springer.com/chapter/10.1007/978-3-642-10868-6_23.

N. Ferguson and B. Schneier. *Practical Cryptography*. Wiley & Sons, 2003.

Oded Goldreich. *Foundations of Cryptography: Volume 1*. Cambridge University Press, New York, NY, USA, 2006. ISBN 0521035368.

Howard M. Heys. A tutorial on linear and differential cryptanalysis. *Cryptologia*, 26(3): 189–221, July 2002. ISSN 0161-1194. doi: 10.1080/0161-110291890885. URL http://dx.doi.org/10.1080/0161-110291890885.

W. C. Huffman and V. Pless. *Fundamentals of error-correcting codes*. Cambridge Univ. Press, 2003. URL http://adrem.ua.ac.be/sites/adrem.ua.ac.be/files/FEC.pdf.

Pascal Junod and Serge Vaudenay. Perfect diffusion primitives for block ciphers. In Helena Handschuh and M. Anwar Hasan, editors, *Selected Areas in Cryptography, 11th International Workshop, SAC 2004, Waterloo, Canada, August 9-10, 2004, Revised Selected Papers*, volume 3357 of *Lecture Notes in Computer Science*, pages 84–99. Springer, 2004. ISBN 3-540-24327-5. doi: 10.1007/978-3-540-30564-4_6. URL http://dx.doi.org/10.1007/978-3-540-30564-4_6.

Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography (Chapman & Hall/Crc Cryptography and Network Security Series)*. Chapman & Hall/CRC, 2007. ISBN 1584885513.

Aviad Kipnis and Adi Shamir. Cryptanalysis of the hfe public key cryptosystem by relinearization. In *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 19–30. Springer, 1999. doi: 10.1007/3-540-48405-1_2.

Donald E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997. ISBN 0-201-89684-2.

Frédéric Lafitte, Jorge Nakahara Jr., and Dirk Van Heule. Applications of SAT Solvers in Cryptanalysis: Finding Weak Keys and Preimages. *Journal on Satisfiability, Boolean Modeling and Computation*, 52(9):1–25, 2014. URL http://satassociation.org/jsat/index.php/jsat/article/download/114/104.

Fabio Massacci and Laura Marraro. Logical cryptanalysis as a sat-problem: Encoding and analysis. *In Journal of Automated Reasoning*, 24:165–203, 2000.

Mitsuru Matsui. Linear cryptanalysis method for DES cipher. In *Advances in Cryptology - EUROCRYPT '93, Workshop on the Theory and Application of of Cryptographic Techniques, Lofthus, Norway, May 23-27, 1993, Proceedings*, pages 386–397, 1993. doi: 10.1007/3-540-48285-7_33. URL http://dx.doi.org/10.1007/3-540-48285-7_33.

Sean Murphy. The cryptanalysis of FEAL-4 with 20 chosen plaintexts. *J. Cryptology*, 2(3):145–154, 1990. doi: 10.1007/BF00190801. URL http://dx.doi.org/10.1007/BF00190801.

W.Keith Nicholson. *Introduction to abstract algebra. 2rd ed.* Wiley-Interscience. Hoboken, NJ: John Wiley & Sons, 2007.

Raphael Chung-Wei Phan. Mini Advanced Encryption Standard (Mini-AES): a testbed for cryptanalysis students. *CRYPTOLOGIA*, 26(4):283–306, 2002. ISSN 0161-1194 (print), 1558-1586 (electronic).

Alexander Pott and Yves Edel. A new almost perfect nonlinear function which is not quadratic. *Advances in Mathematics of Communications*, 3(1):59–81, January 2009. ISSN 1930-5346. doi: 10.3934/amc.2009.3.59. URL http://www.aimsciences.org/journals/displayArticles.jsp?paperID=3958.

Bruce Schneier. *Applied Cryptography*. Wiley, 2nd edition, 1996. ISBN 0471117099.

Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, and Niels Ferguson. *The Twofish Encryption Algorithm: A 128-bit Block Cipher*. John Wiley & Sons, Inc., New York, NY, USA, 1999. ISBN 0-471-35381-7.

Claude E. Shannon. Communication Theory of Secrecy Systems. *Bell Systems Technical Journal*, 28:656–715, 1949. Accessed: 2014-11-28.

The Sage Development Team. Sagemath frequently asked questions. http://doc.sagemath.org/html/en/faq/faq-usage.html#can-i-use-sage-with-python-3-x, a. Accessed: 2015-08-16.

The Sage Development Team. Sagemath homepage. http://www.sagemath.org/, b. Accessed: 2015-08-16.

Serge Vaudenay. On the need for multipermutations: Cryptanalysis of MD4 and SAFER. In Bart Preneel, editor, *Fast Software Encryption: Second International Workshop. Leuven, Belgium, 14-16 December 1994, Proceedings*, volume 1008 of *Lecture Notes in Computer Science*, pages 286–297. Springer, 1994. doi: 10.1007/3-540-60590-8_22. URL http://dx.doi.org/10.1007/3-540-60590-8_22.

Serge Vaudenay. Security flaws induced by cbc padding - applications to ssl, ipsec, wtls. In *Advances in Cryptology - EUROCRYPT 2002, International Conference on the Theory and Applications of Cryptographic Techniques, Amsterdam, The Netherlands, April 28 - May 2, 2002, Proceedings*, volume 2332 of *Lecture Notes in Computer Science*, pages 534–546. Springer, 2002. doi: 10.1007/3-540-46035-7_35. URL http://www.iacr.org/cryptodb/archive/2002/EUROCRYPT/2850/2850.pdf.

Guobian Weng, Yin Tan, and Guang Gong. On Quadratic Almost Perfect Nonlinear Functions and Their Related Algebraic Object. *Pre-proceedings of the International …*, pages 1–22, 2013. URL http://cacr.uwaterloo.ca/techreports/2013/cacr2013-18.pdf.

Wikipedia. Chinese remainder theorem. http://en.wikipedia.org/wiki/Chinese_remainder_theorem, a. Accessed: 2014-11-24.

Wikipedia. Mds matrix. http://en.wikipedia.org/wiki/MDS_matrix, b. Accessed: 2014-11-28.

Wikipedia. Rot13. http://en.wikipedia.org/wiki/ROT13, c. Accessed: 2015-01-22.

Wikipedia. Sagemath. https://en.wikipedia.org/wiki/SageMath, d. Accessed: 2015-08-16.

Wikipedia. Caesar's vvv. http://en.wikipedia.org/wiki/Veni,_vidi,_vici, e. Accessed: 2015-01-22.

INDEX

Advanced Encryption Standard, 3, 4, 16, 18, 44, 47, 68, 76
  Mini, 44, 51, 52, 58, 60, 69
AES, *see* Advanced Encryption Standard
Almost Perfect Nonlinear function, 21–23, 26, 52, 54–56, 64
  Gold, 26, 27, 33, 47, 55, 68
APNL, *see* Almost Perfect Nonlinear (function)
Attacks
  chosen plaintext, 11, 43, 44, 48, 68
  known plaintext, 11, 58, 59

Chinese Remainder Theorem, 3, 9, 21–23, 25–27, 35–41, 66
  basis, 35, 37
  matrix, 23, 38–41, 66–68
Chosen plaintext attack, *see* Attacks, chosen plaintext
Confusion, 4, 12, 19, 43
CRT, *see* Chinese Remainder Theorem
Cryptanalysis
  algebraic, 4, 55, 58, 64
  differential, 4, 11, 22, 33, 43, 44, 46–48, 50–52, 54–59, 63, 64, 68, 70
  linear, 4, 11, 58–60, 62, 64, 69, 71

Data Encryption Standard, 43, 44
DES, *see* Data Encryption Standard
Differential cryptanalysis, *see* Cryptanalysis, differential
Diffusion, 4, 12, 19, 35, 36, 39, 72, 74

Galois field, 7, 29, 65
  GF(2), 7–9, 21, 36, 40, 59, 60, 65, 73, 76, 77
Gold function, *see* Almost Perfect Nonlinear (function), Gold

Hamming, 40, 41, 63, 64, 72, 74, 77
  distance, 72, 74
  weight, 40, 41, 63, 64, 72, 74, 77

key schedule, 21, 34, 64

LFSR, *see* Linear Feedback Shift Register
Linear Feedback Shift Register, 8, 9, 22, 30

Maximum Distance Separable, 40, 41, 72–77
MDS, *see* Maximum Distance Separable
Mini-AES, *see* Advanced Encryption Standard, Mini

National Institute of Standards and Technology, 3
NIST, *see* National Institute of Standards and Technology

Rijndael, 3, 4, 12, 15, 17–19, 34, 39, 61, 64

S-Box, 19, 22, 23, 25–27, 29–33, 41, 43–47, 49–52, 54–56, 58–61, 64, 68, 69
SAGE, 8, 26, 27, 29, 30, 35, 36, 41, 56, 65, 67, 68, 70, 71
Shannon, Claude Elwood, 13–15, 19, 39
Singleton, 73, 74
  bound, 73, 74
Substitution box, *see* S-Box

Index

ACRONYMS

AES	Advanced Encryption Standard. 3, 4, 16, 18, 44, 47, 68, 76
APNL	*Almost Perfect Nonlinear* function. 21–23, 26, 52, 54–56, 64
CRT	Chinese Remainder Theorem. 3, 9, 21–23, 25–27, 35–41, 66–68
DES	Data Encryption Standard. 43, 44
LFSR	Linear Feedback Shift Register. 8, 9, 22, 30
NIST	National Institute of Standards and Technology. 3
SAGE	"General algebra" mathematics software. 8, 26, 27, 29, 30, 35, 36, 41, 56, 65, 67, 68, 70, 71