Cryptanalysis of AES

By Kristoffer Rye

Dissertation for the degree of Master's in Secure and reliable communication
at the University of Bergen

2020

Acknowledgements

First of all, I would like to thank my supervisor Sondre Rønjom for letting me work on this very interesting project in cryptanalysis of AES. It has been a pleasure to work on this thesis under his guidance. He has shown passion and interest in new ideas and guided me in the right directions when help was needed. He has always answered my questions related to the topic of cryptography and his insightful comments and encouragement have been essential to my understanding of the material.

Many hours have been spent on programming, writing, trying out different attack methods, thinking of possible solutions to different problems, calculating and understanding complex mathemathics in cryptography and trying to improve existing techniques. I will therefore also take this opportunity to give a huge thanks to my friends and family for being patient and understandable when work had to be prioritized.

Kristoffer Rye
Bergen, 16.06.2020

Abstract

This is my thesis for the degree of Masters in Secure and reliable communicationation at the University of Bergen, Department of Informatics at the Faculty of Mathematics and Natural Sciences. The aim of this project is to implement and study subspace attacks on 5-rounds AES and the so-called yoyo-trick, two recent and successful techniques for analysing AES. An important part of the project is to implement these two attacks and investigate and experiment with them to see whether there is room for improving the attacks and/or whether there exist variations of the attacks that are better. We will list the conclusions drawn from first-hand experimenting with the techniques in Chapter 6, present some newly discovered properties of AES in Chapter 5 and also make some suggestions for further research as a summary. The project has also a theoretical survey component consisting of presenting and surveying these attacks in the context of cryptanalytics attacks on AES, thus we introduce the knowledge, concepts within cryptanalysis and the building blocks for how to perform these attacks in great detail.

Outline

This thesis consists of an introductory part of how the symmetric cipher AES works in detail and how truncated differential cryptanalysis can be used to break some round reduced versions of AES. All the main attacks in this analysis has been successfully implemented using *C#*, and we have improved the overall data complexity of the key recovery attack on 5 rounds from $2^{11.3}$ ACC to roughly $2^{10.59}$ ACC by doing some slight adjustment in the implementation of the attack.

Chapter 1 gives a short introduction to the concepts within cryptography and the importance of cyber security in general. Chapter 2 gives an insight in how symmetric cryptosystems works and the design principles of block ciphers and SP-networks. Chapter 3 gives a detailed explanation of how the AES algorithm works. Chapter 4 gives an overview of existing attack methods that is applied on round reduced versions of AES. Chapter 5 lists all the cryptanalysis tools to perform the exchange attack on AES. The main attacks on 5 rounds for this thesis is described in Chapter 6 where we also add some self-made attacks based on the same principles of the main attacks. We list our conclusions and findings in Chapter 7.

Contents

Chapter 1

Introduction

Cryptography is the science of protecting information and communication through the use of secret codes with the goal of hiding the meaning of the message from malicious third parties. The field of cryptography deals with the practice and study of different cryptographic techniques that provides the most secure form of communication in the presence of such adversaries. We all use cryptography almost daily whether we know it or not. When we surf the web, communicate with our cell phones, unlock the car or when we shop groceries at the store with our credit card, cryptography is used in one way or another. The purpose of cryptography is to be able to communicate with something or someone safely without anyone else being able to see the content, intercept and change the content without our knowledge and to prevent anyone else from pretending to be someone else. These categories fall under the topic of authentication, authorization, accounting and identity management, all of which are sub-genres in cryptography.

The specific techniques and methods for achieving these goals are through the use of different cryptographic algorithms, also referred to as ciphers. The earliest forms of cryptography dates back to ancient Egypt in the year 2000 B.C. in the form of secret hieroglyphs and in ancient Rome, techniques to manipulate traditional text by scrambling individual characters in a messages was developed. Since then, the field of cryptography has evolved into a much more sophisticated field of science where we distinguish between the old classical cryptography and modern cryptography.

While classical cryptography is mainly based on security through obscurity by using secret ciphers that only the involved parties known about, modern cryptography relies on publicly known mathematical algorithms for encoding information. This is achieved by mixing a secret key into a cryptographic algorithm which produces a unique data stream that is dependent on the secret key. We categorize cryptography into three main branches, all of which have their own purpose and uses: symmetric ciphers, asymmetric ciphers and cryptographic protocols. Symmetric ciphers use the same secret shared key for encryption and decryption, asymmetric ciphers are often used in key exchanges and use different encryption and decryption keys while cryptographic protocols deals with the use and implementation of cryptographic algorithms. In this theses we will focus on symmetric cryptography, as AES falls under this category.

History is filled with examples of situations where the importance of hiding information, or communicating through a secure channel is critical. The importance of cryptography has become more relevant today as the world is becoming more and more connected and hackers trying to get their hands on others information for malicious purposes. Therefore, the field of cryptography is constantly evolving to develop newer and more secure technologies to make the world safer. At the same time, a specific field of study is evolving for cracking cryptographic codes known as cryptanalysis, often referred to as the art of breaking cryptosystems. As cryptographic methods becomes more secure and reliable, different techniques for exploiting vulnerabilities in ciphers also becomes more complex and sophisticated. Out of those who deals with code breaking, we distinguish between black hat, white hat and gray hat hackers. Black hat hackers are the ones who break cryptosystems for malicious reasons, white hats tries to find and repair security holes before someone with evil intentions does and gray hats is in a gray area in between. As in many other areas, the best defense mechanism is often to attack first, as cryptanalysis is all about analyzing cryptosystems to find vulnerabilities which can be exploited and attacked before other takes advantage of it. When such a vulnerability has been found and exploited in a way that weakens the cipher, the cipher is considered as broken and no longer secure.

The two fields of cryptography and cryptanalysis is commonly generalized under the term cryptology, which is the study of the mathematics, number theory and the cryptographic algorithmns that underpin cryptography and cryptanalysis. In cryptology, there is a constant race between those who tries to make secure codes and those who tries to break them. Thus, the progress and findings in cryptology are distinguished by the hidden work of the military, the secret work of black hats and the open academic community where scientific discoveries are shared in public.
There is an ongoing debate in the academic cryptology community about how much work is done secretly, especially considering the work of the USA National Security Agency (NSA) which is the largest employer of mathematical personal in the world and who has designed many of the cryptographic standards used worldwide. There has also recently been a growing number of cyber attacks against large corporations from malicious hackers that we constantly hear about in the media, which means that we must constantly develop new methods to secure our systems. It is therefore fair to say that considerable work has been done in secret and that people who understand cryptography and cyber security in general are needed to help maintain better system security and prevent new cyber attacks from happening in the future.

Chapter 2

Symmetric cryptosystems

In this chapter we will cover the basics in symmetric cryptography and how encryption and decryption works in a symmetric cryptographic system to give the reader a basic understanding of the terms and concepts used later in this theses.

To encrypt a message means that we make the message unreadable or not understandable to those who see the message. This is done by manipulating the information in such a way that if the information gets intercepted, it's virtually worthless without the code or key that would revert it to its original format. The whole purpose of encrypting data is to make the information unavailable for others who should not have access to the information. Decrypting a message is the direct reverse process of encryption and means that we make an incomprehensible message understandable. We refer to an unencrypted human readable message as a plaintext and an encrypted message as a ciphertext.
Traditionally, cryptography has been used to manipulate text directly by scrambling individual letters in a specific way according to a specific pattern which also relies on a secret key. Doing these processes manually is both slow and often requires the entire codebook to communicate securely. Today, encryption and decryption is done automatically by computers in a few milliseconds and instead of using encryption and decryption directly on text, it is used on bits and bytes that represent information. Many cryptosystems rely entirely on software, while others are implemented directly in hardware. Advanced Encryption Standard (AES), which is the most widely used symmetric key cryptosystem, can be implemented in both. Since the focus of this thesis is about cryptanalysis of AES, we need to be familiar with the underlying algorithms in cryptography and cryptosystems in general. We begin by introducing the most fundamental properties of symmetric key cryptosystems.

2.1 Symmetric ciphers

While asymmetric encryption is a relatively new form of cryptography, symmetric encryption is the oldest cryptographic technique in the book. A symmetric cipher is a specific symmetric key algorithm that uses the same secret key for encryption and decryption, hence the name symmetric. Because of the use of only one single key, this it is also generally faster then asymmetric encryption. A symmetric key cryptosystem works in both directions as it converts a plaintext to a ciphertext and a ciphertext to a plaintext by the help of a cryptographic algorithm and one single secret key that is shared between those communicating through the cryptosystem. One can think of the specific cipher used as a black box that takes a plaintext and a secret key as input and mixes them together with a mathematical function to produce a ciphertext as output. To get the corresponding plaintext from the ciphertext, the same key needs to be used for decryption, as using another key than the original will destroy the information and make it unreadable.

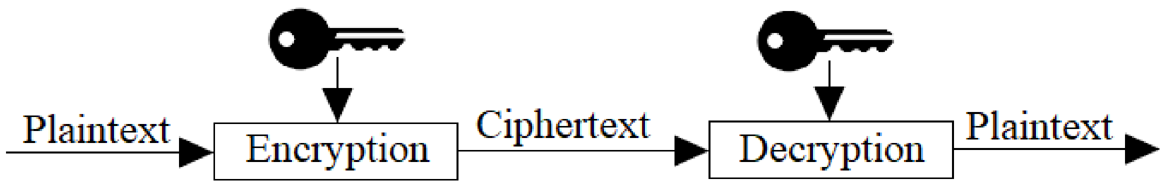


Figure 2.1: Symmetric key cryptosystem

Sending a message to another person can be done in many ways through various mediums, such as an electrical wire or through the air via electromagnetic signals. The medium we send the message in is referred to as a channel where we distinguish between secure channels (encrypted channels) and insecure channels (unencrypted channels). If we have complete control over those who have access to the channel, it is not necessary to encrypt the information because we can be sure that no one else can access it, but since there are countless ways to eavesdrop a channel, we should always assume that the channel we are sending information through is unsecured. Especially when we communicate over the Internet where anyone with a little technical knowledge can access the information through various interception attacks, like for instance a so-called *Man In The Middle* attack. We therefore secure an insecure channel by encrypting the traffic we send through it, so that if such an attacker gets hold of the information, they would just see the unreadable ciphertext instead.

In mathematical terms we can think of the cryptographic algrorithm used for encryption and decryption as a function F that takes two input arguments and returns an output where F_{enc} works in the encryption direction and F_{dec} works in the decryption direction. Let P denote the plaintext, C the ciphertext and K the secret key. Then C and P is expressed as follows: $C = F_{enc}(P,K)$, $P = F_{dec}(C,K)$.
We see that given P and K, we can calculate C directly by the function F which we also calculate P directly with, given C and K. In a strong cipher, there should be impossible to guess either P or C without K. Some cryptographic attacks which is called "known plaintext/ciphertext" attacks is based on finding the secret key given the plaintext and the ciphertext as input. A cipher is considered much stronger if it can resist such an attack.

A cryptographic algorithm usually consists of many sub-functions with their own properties, and for it to be a symmetric algorithm it needs to be reversible. There are mainly two categories when it comes to symmetric ciphers that use such algorithms: stream ciphers and block ciphers. The main differences between these are that stream ciphers encrypt each bit of a text individually, while block ciphers encrypts a whole block of text at the same time. We will take a closer look at block ciphers because this is the type of symmetric cipher that AES is.

2.1.1 Block ciphers

A block cipher is a symmetrical cipher that operates on groups of fixed lengths of bits. To encrypt a text with a block cipher, we must first convert each letter of the text into its binary equivalent, which is usually done by ASCII conversion where each character is mapped to a specific byte value. The bytes are then fed into a block of fixed length called a block state, which we will represent as a matrix $[0,1]^n$. For a block size of length n bits, there can be 2^n different block states. The block sizes vary between the different block ciphers, but the most commonly used block sizes are 64, 128, 192 and 256 bits. If the length of the text is greater than the length of the block, the rest of the text moves to a new block and so on. The same process is also applied to the secret key. A block cipher algorithm takes the block states of the plaintext and the key as input and produces a new block state of the same size as the output, which is the ciphertext. The encryption process is described as

$$E_k(P) = E(K,P) = [0,1]^n_K \times [0,1]^n_P \to [0,1]^n_C$$

where E is the encryption function and $[0,1]^n_K$, $[0,1]^n_P$ and $[0,1]^n_C$ represent the block states of the key, plaintext and ciphertext respectively. The decryption function D can be expressed as the inverse of the encryption function

$$D_k(C) = E_k^{-1}(C) = D(K,C) = [0,1]^n_K \times [0,1]^n_C \to [0,1]^n_P.$$

The most common structure of a block cipher is to use an iterative algorithm where several sub-functions are repeated one after another a limited number of rounds where the output of one round is used as input in the next round. In a such iterated block cipher, several round keys are generated from the main key by a key schedule algorithm and mixed into each round function to destroy any traces of the original text where the goal is to scramble the original text so much that it is unrecognizable and looks like random gibberish. Each round R has an associated round key rk which is different from the other round keys such that each iteration becomes unique based on the round key. For n number of rounds, this process can be described as

$$R^n(x,K) = F(x,rk_0) \times F(x,rk_1) \times \ldots \times F(x,rk_{n-1})$$

where x is any block state and F is just a representation of all the sub-functions included in a one round. It usually becomes harder to find any patterns in the cipher the more rounds that's used as this mixes up the block state even more. However, this also affects the efficiency of the cipher since this means that the algorithm takes more time to complete. The time also depends on how much computation must be performed in each round function. In today's society when time is of the essence, we want to have an encryption method that is instantaneously and at the same time as secure as possible, and thus we must limit the number of rounds and make sure that the sub-functions used in each round mixes up the block state enough to to provide a good safety margin.

Furthermore, a good block cipher should also be easy to analyze so that it can be shown how many rounds that's enough to be confident that the cipher is secure.

There are usually two common ways of constructing the round functions in an iterated block cipher. One of them is to split the intermediate values of the block state in half and perform a function and a key addition to one of the half's, which is what a feistel network does. The other design principle is called an SPN cipher, which we will take a closer look at.

2.1.2 Substitution Permutation Network

An SP network (Substitution Permutation network) consists of a combination of substitution layers and permutation layers that contain different properties. In a block cipher that is designed by the SPN principle, each block state goes through the different layers in the SP network in each round.

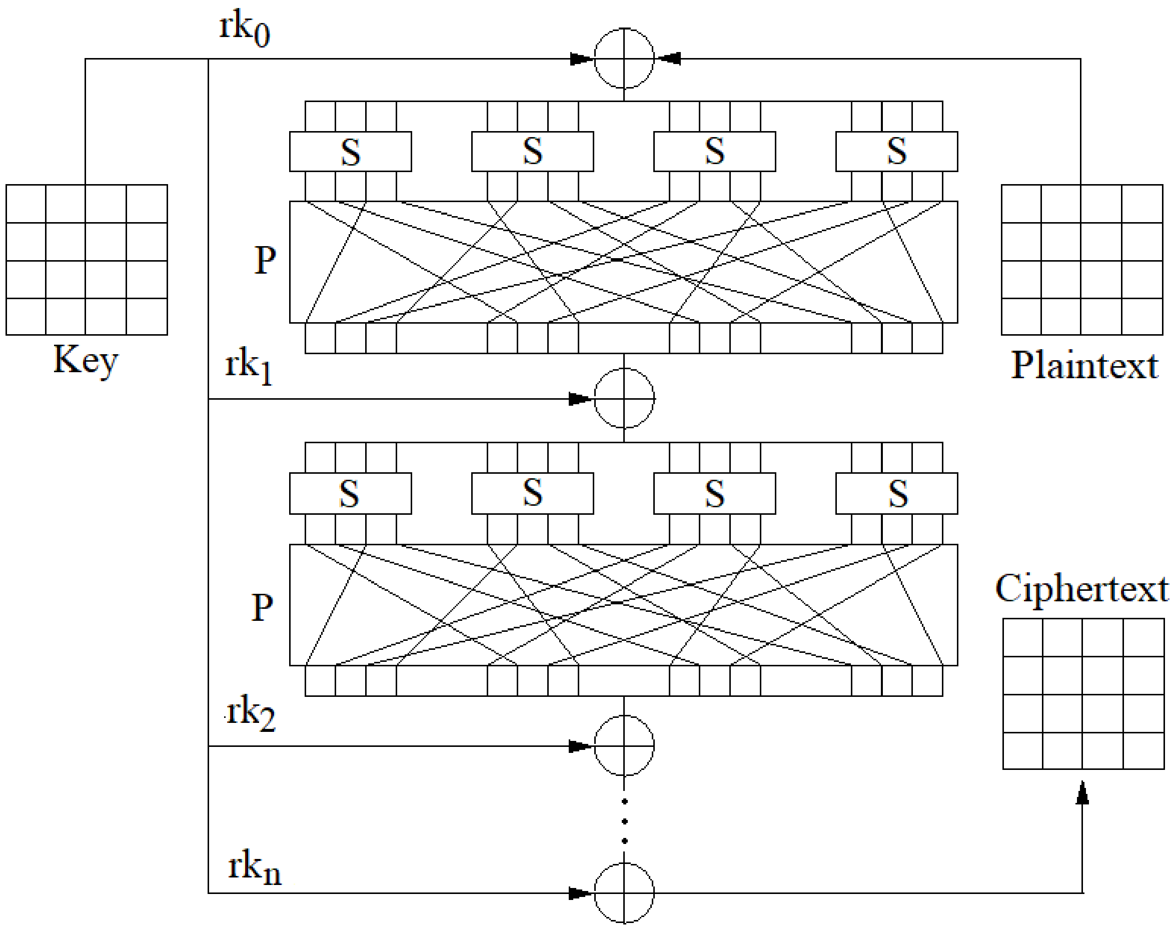


Figure 2.2: Block cipher encryption through a SPN network

The substitution layer is a non-linear transformation which usually consist of a mapping function called an S-box where all the bytes are mapped to another byte. The transformation is performed on each individual byte regardless of the value of the other bytes in the block state. A requirement for decryption is that the S-box uses a bijective mapping system to prevent collisions, e.g. the letter A is mapped to the letter E and in the reverse S-box, E is then mapped to A and so on. This adds a level of confusion to the cipher and many ciphers are built only by this principle, like for instance the classical substitution cipher which only needs one lookup table to encrypt/decrypt a letter. An SPN cipher can use many different S-boxes per round, but considering the implementation of the cipher, this also requires storing more lookup tables in memory.

The permutation layer is a linear mapping function applied to an array of bytes where the byte positions are interchanged. An example of this is when we shuffle a deck of cards to destroy the order. It becomes more difficult to restore the deck back to its original state, the more we shuffle the cards. In cryptography this is known as diffusion, which is one of the key elements in many ciphers, such as transposition cipher and affine cipher to name a few. An array of n bytes has exactly $(n!)$ ways to be permuted and we call a specific permutation function a P-box.

The substitution and permutation layers in an SP network are often abbreviated as the S and L layer (due to the linearity of the P-box). We denote $\alpha = (\alpha_0, \alpha_1, ...\alpha_{n-1})$ as the representation of a block state. The encryption and decryption of one round in a SL system can then be described as

$$\begin{aligned} &R(\alpha) = S(L(\alpha)) \\ &R^{-1}(\alpha) = S^{-1}(L^{-1}(\alpha)) \end{aligned}$$

where R^{-1} is the decryption of one round an S^{-1} and L^{-1} are the inverse function of S and L respectively. In an SPN cipher, a block states goes through multiple S-boxes and P-boxes before its get totally encrypted. A simplified version of an SP network in the encryption direction is depicted in Figure 2.2 where the block states of a key and a plaintext is used to produce a ciphertext. Decryption is easily done by doing all the steps in reverse, as an SP network is easy to reverse in itself when we know the algorithm. Because of this, it would be useless without a secret key. Normally the different round keys are added in between each round by a bitwise XOR of the intermediate state and the round key to make each round different. Frequently, *prewhitening* is used in addition to this where the original key rk_0 is usually XOR-ed directly with the plaintext before the round functions begin to make the cipher even more secure.

Chapter 3

Description of AES

AES (Advanced Encryption Standard) is the best known and most widely used secret key cryptosystem in the world. It is a strong symmetric block cipher used in many real world applications today to encrypt and decrypt data. Almost all secure connections on the Internet use AES as its cryptosystem. The cipher is based on the Rijndael algorithm developed by Vincent Rijmen and Joan Daemen in 1998, who submitted the algorithm as a proposal to NIST (National Institute of Standards and Technology) during the AES selection process that lasted from 1997 - 2000. The Rijndael algorithm was selected as the new encryption standard in the year 2000 as a replacement for the earlier DES (Data Encryption Standard) [28] cipher.
DES (developed by IBM) had been used as the encryption standard since the 1970s and was considered secure at that time, but as computing power increased and new attacks evolved, it was later considered broken. This was primarily because of the small key length of only 56 bits which could easily be brutefored, but there was also shown that the DES algorithm itself had some weaknesses. There was also some controversy about how the cipher was chosen as the encryption standard since the National Security Agency (NSA) along with the National Bureau of Standards (NBS) had chosen a slightly modified version of IBM's version of DES with some classified element design, leading to speculation about that the NSA had planted a back door in the cipher. In addition, DES was designed primarily for hardware and was relatively slow when implemented in software.
When NIST requested a new encryption standard in 1997, there were certain requirements it had to meet. First of all, it had to be impossible to break the new cipher with the technology of the time, but also in the future. To ensure this, it was decided that the new cipher needed to be able to support implementation of 128, 192 and 256 bit keys. Unlike the DES development, the AES selection was an open process with several contributions where the security of each proposal was rigorously tested and checked for any back doors so that people could feel confident using it. It also had to be able to run fast in software so that it wouldn't slow down any applications. The Rijndael algorithm met all of this requirements and was verified by many cryptologists around the world who tried to break the cipher, but none of them had any luck. Thus, AES was selected as the new encryption standard. Today, AES is one of the most well studied block ciphers and there is yet no known attacks who can break it.

3.1 The AES algorithm

The Rijndael cipher [10] itself is a flexible block cipher that can operate with different key and block sizes which can be any multiple of 32 bits. AES, on the other hand, is a variant of the Rinjndael algorithm with a fixed block size of 128 bits and a key size of 128, 192, or 256 bits. Each block is divided into a 4 x 4 matrix called a state, where each entry holds a value of 1 byte. When we encrypt a plain-text with a secret key using AES, each ASCCI character of the plain-text and the key is first transformed to its byte representation and stored in a block where the bytes are padded so that it is a multiple of the block size. The bytes are arranged like this:

α_0	α_4	α_8	α_{12}
α_1	α_5	α_9	α_{13}
α_2	α_6	α_{10}	α_{14}
α_3	α_7	α_{11}	α_{15}

Table 3.1: Arrangement of bytes in the block state

The key is used to scramble the plain-text to a cipher-text that is unrecognizable from the plain-text. Since AES is a Symmetric key cryptosystem, we use the same key to decrypt the cipher-text back to its original plain-text, as AES is a reversible algorithm.

The design principle of the algorithm is based on a Substitution - Permutation (SPN) network which is a series of linked mathematical operations performed on a block state that provides confusion and diffusion to the original input. In a SPN network, a state goes trough multiple layers (or rounds) of substitution boxes (S-boxes) and permutation boxes (P-boxes) to produce the output. The number of internal rounds a state goes trough in AES depends on the key length that's used. For a key length of 128 bits there are 10 rounds, for 192 bits there are 12 rounds and for 256 bits there are 14 rounds.
When it comes to the security of cryptosystems, it is usually better the more rounds you have, but this also effects the run time of the algorithm. The reason why AES can have a relatively small number of rounds compared to other cryptosystems is because all 128 bits of the block state gets totally encrypted independently of each other in each iteration. So 10, 12 and 14 roundts is more than enough to feel confident that this cipher is secure.

One round of AES consist of smaller, sub-algorithms which will be explained in more details in the next sub-sections. These are called SubBytes, ShiftRows, MixColumns and AddRoundKey. In the decryption direction we reverse these functions and they are then called InvSubBytes, InvShiftRows and InvMixColumns. The AddRoundKey function is the same in both directions, as this is just an XOR operation. Figure 2.1 shows the sequence of these functions in both the encryption and decryption direction for a key length of 128 bits. The sequence of the round functions is the same in all the AES variants where the only difference is number of rounds and key length. The MixColumn function is omitted in the last round for reasons discussed in section 2.1.4.

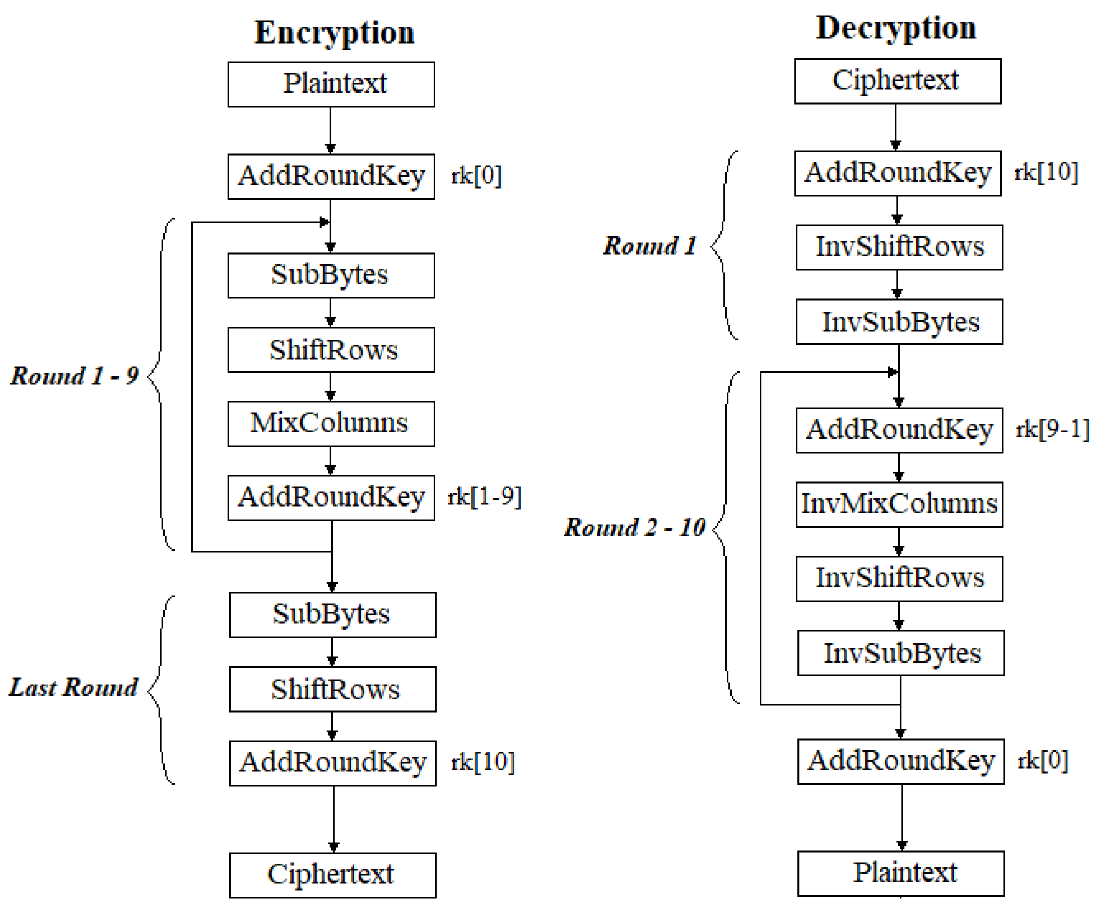


Figure 3.1: AES algorithm with 10 rounds

All the calculations operating on the block states are done within the realm of a particular finite field of order 2^8 called a Galois Field, denoted GF(2^8), where each byte is treated as polynomials of order x^7 with coefficients in GF(2). Within this finite field, we can add, subtract, multiply and divide without any overflow as long as we follow the rules of polynomial arithmetic. In cryptography, finite fields is of particular interest when designing cryptosystems because it gives us a limited number of elements to work with inside a closed system. GF(2^8) is used in AES because the total number of combinations is exactly 1 byte, making it easier to implement in software. Out of the $2^8 = 256$ possible polynomials, each element $A \in GF(2^8)$ is represented as:

$$A(x) = a_7x^7 + a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$$

Addition and subtraction is simply an XOR operation in the underlying field GF(2), which is done by adding all the bits individually and reduce the sum modulo 2, wheres multiplication and division is a bit more tricky. Multiplication of two polynomials is done by first multiplying the polynomials by regular polynomial multiplication, then dividing the product by the irreducible polynomial

$$P(x) = x^8 + x^4 + x^3 + x + 1$$

and then taking the remainder after the polynomial division. This is done because when we usually multiply polynomials, it is possible that the product might end up with a higher power outside the finite field. To stay inside GF(2^8), we therefore have to reduce the product modulo $P(x)$, which is a part of the AES specification. This is written as $C(x) \equiv A(x) \cdot B(x) \bmod P(x)$

Division, on the other hand, is the same as inverting a polynomial $A(x)$ in GF(2^8) and is done by finding the multiplicative inverse $A^{-1}(x)$ of the polynomial $A(x)$ that makes the product $A(x) \cdot A^{-1}(x)$ congruent to 1 modulo P(x), which can be done using e.g. the Extended Euclidean Algorithm. Multiplication and inversion in GF(2^8) is an important part of the SubBytes and MixColumn layer.

3.1.1 AddRoundKey

AddRoundKey is the first function in the AES algorithm in the encryption direction where we XOR the key with the plain-text. In cryptography, this step is often called "pre whitening" (or key whitening) because it is usually applied before the first round to increase the complexity of a brute force attack. AddRounKey is also applied in every round function by XOR-ing the intermediate state with the corresponding round key for each round. For a block state $\alpha = (\alpha_0, \alpha_1, ..., \alpha_{n-1})$ and a key $K = (K_0, K_1, ..., K_{n-1})$, the AddRoundKey function (denotes as AK) is described as:

$$AK(\alpha, K) = (\alpha_0 \oplus K_0, \alpha_0 \oplus K_1, ..., \alpha_{n-1} \oplus K_{n-1})$$

where each byte in the block state are XOR-ed individually.

3.1.2 SubBytes

The first function in the internal rounds in the encryption direction is SubBytes. This is a substitution function (denoted as SB) where each byte α_i of the state gets individually substituted with another byte β_i according to a substitution box (S-box). In contrast to other cryptosystems, the AES S-box is the same in every round. In the S-Box, all of the 256 possible bytes are one-to-one mapped to different output bytes without any collisions, i.e., different bytes cannot be transformed into the same byte. This is called bijective mapping and allows us to reverse the substitution function, which is needed for decryption. For two states α and β, it holds that SB(α) + SB(β) $\neq$ SB($\alpha + \beta$), making the substitution layer a highly nonlinear transformation and assures that changes in individual state bits propagate quickly across the data path.

The structure of this one-to-one mapping is not random, but designed with an algebraic structure that gives the S-box some unique mathematical properties that provides confusion to the algorithm, making the cipher stronger against analytical attacks. It consists of two steps: first an inversion in GF(2^8) followed by an affine mapping function. A table of all 256 possible inversions in GF(2^8) is given in table 2.2.

We display the coefficients in their hexadecimal equivalent of the byte value. The only byte mapped to itself is 0, which the inverse in GF(2^8) does not exist, but defined this way in the AES S-box so that each input value has an output value. To invert

Table 3.2: Multiplicative inverse in $GF(2^8)$

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	00	01	8D	F6	CB	52	7B	D1	E8	F4	29	C0	B0	E1	E5	C7
1	74	B4	AA	4B	99	2B	60	5F	58	3F	FD	CC	CC	FF	40	EE
2	3A	6E	5A	F1	55	4D	A8	C9	C1	0A	98	15	30	44	A2	C2
3	2C	45	92	6C	F3	39	66	42	F2	35	20	6F	77	BB	59	19
4	1D	FE	37	67	2D	31	F5	69	A7	64	AB	13	54	25	E9	09
5	ED	5C	05	CA	C4	24	87	BF	18	3E	22	F0	51	EC	61	17
6	16	5E	AF	D3	49	A6	36	43	F4	47	91	DF	33	93	21	3B
7	79	B7	97	85	10	B5	BA	3C	B6	70	D0	06	A1	FA	81	82
8	83	7E	7F	80	96	73	BE	56	9B	9E	95	D9	F7	02	B9	A4
9	DE	6A	32	6D	D8	8A	84	72	2A	14	9F	88	F9	DC	89	9A
A	FB	7C	2E	C3	8F	B8	65	48	26	C8	12	4A	CE	E7	D2	62
B	0C	E0	1F	EF	11	75	78	71	A5	8E	76	3D	BD	BC	86	57
C	0B	28	2F	A3	DA	D4	E4	0F	A9	27	53	04	1B	FC	AC	E6
D	7A	07	AE	63	C5	DB	E2	EA	94	8B	C4	D5	9D	F8	90	6B
E	B1	0D	D6	EB	C6	0E	CF	AD	08	4E	D7	E3	5D	50	1E	B3
F	5B	23	38	34	68	46	03	8C	DD	9C	7D	A0	CD	1A	41	1C

e.g. the byte $\alpha_i = 0\text{x}45$, we read the table from the vertical index number 4 and then the horizontal index number 5 and end up with the inverted value α_i^{-1}, which is 0x31.

After the inversion in GF(2^8), each inverted byte α_i^{-1} goes trough an affine mapping process where each byte is multiplied by a constant 8 x 8 bit-matrix and then added with a constant 8 bit vector like so:

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} \equiv \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} a_0^{-1} \\ a_1^{-1} \\ a_2^{-1} \\ a_3^{-1} \\ a_4^{-1} \\ a_5^{-1} \\ a_6^{-1} \\ a_7^{-1} \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \mod 2,$$

where the output value $\beta_i = (b_7, ..., b_0)$ is the new substituted value for α_i. The constant matrices in this affine step is carefully chosen to make sure that the structure in GF(2^8) gets destroyed, preventing attacks that can exploit the inversion step. One can of course compute this calculation for each byte in the state matrix, but in software it is much faster to implement a pre-built lookup table where each possible substitution value exist. For decryption, we reverse the GF(2^8) inversion and the affine mapping step to make an inverse S-box, which also is usually stored as a lookup table in software.

Table 3.3: The AES S-box for encryption

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1	CA	82	c9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

Table 3.4: The AES S-box for decryption

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
0	52	09	6A	D5	30	36	A5	38	BF	40	A3	9E	81	F3	D7	FB
1	7C	E3	39	82	9B	2F	FF	87	34	8E	43	44	C4	DE	E9	CB
2	54	7B	94	32	A6	C2	23	3D	EE	4C	95	0B	42	FA	C3	4E
3	08	2E	A1	66	28	D9	24	B2	76	5B	A2	49	6D	8B	D1	25
4	72	F8	F6	64	86	68	98	16	D4	A4	5C	CC	5D	65	B6	92
5	6C	70	48	50	FD	ED	B9	DA	5E	15	46	57	A7	8D	9D	84
6	90	D8	AB	00	8C	BC	D3	0A	F7	E4	58	05	B8	B3	45	06
7	D0	2C	1E	8F	CA	3F	0F	02	C1	AF	BD	03	01	13	8A	6B
8	3A	91	11	41	4F	67	DC	EA	97	F2	CF	CE	F0	B4	E6	73
9	96	AC	74	22	E7	AD	35	85	E2	F9	37	E8	1C	75	DF	6E
A	47	F1	1A	71	1D	29	C5	89	6F	B7	62	0E	AA	18	BE	1B
B	FC	56	3E	4B	C6	D2	79	20	9A	DB	C0	FE	78	CD	5A	F4
C	1F	DD	A8	33	88	07	C7	31	B1	12	10	59	27	80	EC	5F
D	60	51	7F	A9	19	B5	4A	0D	2D	E5	7A	9F	93	C9	9C	EF
E	A0	E0	3B	4D	AE	2A	F5	B0	C8	EB	BB	3C	83	53	99	61
F	17	2B	04	7E	BA	77	D6	26	E1	69	14	63	55	21	0C	7D

3.1.3 ShiftRows

ShiftRows is the second step in the internal rounds, and constitutes the diffusion layer in AES along with the MixColumns transformation. Together, they make sure to mix up the state matrix in a way that only after three rounds, every byte of the state matrix depends on all bytes in the plaintext. In the ShiftRows function, bytes are shuffled around to destroy the individual byte position pattern in the block state. It does this by cyclically shifting the rows in the block state either some positions left or right according to the row numbers. We can think of the block state as a 4 x 4 matrix where the rows are indexed from 0 to 3. In the encryption direction, row 1 is shifted 1 position to the left, row 2 is shifted 2 positions to the left and row 3 is shifted 3 positions to the left, as depicted here:

$$\begin{array}{|c|c|c|c|} \hline \alpha_{0,0} & \alpha_{0,1} & \alpha_{0,2} & \alpha_{0,3} \\ \hline \alpha_{1,0} & \alpha_{1,1} & \alpha_{1,2} & \alpha_{1,3} \\ \hline \alpha_{2,0} & \alpha_{2,1} & \alpha_{2,2} & \alpha_{2,3} \\ \hline \alpha_{3,0} & \alpha_{3,1} & \alpha_{3,2} & \alpha_{3,3} \\ \hline \end{array} \rightarrow \begin{array}{|c|c|c|c|} \hline \alpha_{0,0} & \alpha_{0,1} & \alpha_{0,2} & \alpha_{0,3} \\ \hline \alpha_{1,1} & \alpha_{1,2} & \alpha_{1,3} & \alpha_{1,0} \\ \hline \alpha_{2,2} & \alpha_{2,3} & \alpha_{2,0} & \alpha_{2,1} \\ \hline \alpha_{3,3} & \alpha_{3,0} & \alpha_{3,1} & \alpha_{3,2} \\ \hline \end{array}$$

Table 3.5: ShiftRows in encryption direction

The first row always remains the same. This also applies to the decryption direction, where we instead of shifting to the left, we shift to the right like this:

$$\begin{array}{|c|c|c|c|} \hline \alpha_{0,0} & \alpha_{0,1} & \alpha_{0,2} & \alpha_{0,3} \\ \hline \alpha_{1,0} & \alpha_{1,1} & \alpha_{1,2} & \alpha_{1,3} \\ \hline \alpha_{2,0} & \alpha_{2,1} & \alpha_{2,2} & \alpha_{2,3} \\ \hline \alpha_{3,0} & \alpha_{3,1} & \alpha_{3,2} & \alpha_{3,3} \\ \hline \end{array} \rightarrow \begin{array}{|c|c|c|c|} \hline \alpha_{0,0} & \alpha_{0,1} & \alpha_{0,2} & \alpha_{0,3} \\ \hline \alpha_{1,3} & \alpha_{1,0} & \alpha_{1,1} & \alpha_{1,2} \\ \hline \alpha_{2,2} & \alpha_{2,3} & \alpha_{2,0} & \alpha_{2,1} \\ \hline \alpha_{3,1} & \alpha_{3,2} & \alpha_{3,3} & \alpha_{3,0} \\ \hline \end{array}$$

Table 3.6: ShiftRows in decryption direction

By shifting the rows in this way, we make sure that each column depends on bytes in other columns, which benefits the MixColumn layer. When it comes to differential cryptanalysis, it is good to know that this is a linear transformation and that the equation SR(α) + SR(β) = SR($\alpha + \beta$) holds. It is also worth mentioning that a state α becomes itself by applying the ShiftRows function 4 times in a row, which also applies to the InvShiftRows function

$$\alpha = SR(SR(SR(SR(\alpha)))) = SR^{-1}(SR^{-1}(SR^{-1}(SR^{-1}(\alpha)))).$$

3.1.4 MixColumns

One of the most interesting sub-layers in AES is the MixColumns step. This is a linear transformation that effects each of the columns in the state matrix individually and provides a high level of diffusion. MixColumns is one of the reasons why AES has a strong resistance against differential cryptanalysis, as this step destroys the differential pattern among chosen plaintexts after only few rounds which prevents many types of differential attacks.

The MixColumn function consists of multiplying each individual column in the state matrix with a fixed 4 x 4 matrix over GF(2^8). Each input column $\alpha_{i,0},...,\alpha_{i,3}$ is treated as individual vectors that does not effect the output for the other columns, as a matrix multiplication between a 4 x 4 matrix and a 4 x 1 matrix results in a new 4 x 1 matrix. The MixColumns matrix M is defined by the circular 4 x 4 matrix

$$\begin{bmatrix} \alpha & \alpha\oplus 1 & 1 & 1 \\ 1 & \alpha & \alpha\oplus 1 & 1 \\ 1 & 1 & \alpha & \alpha\oplus 1 \\ \alpha\oplus 1 & 1 & 1 & \alpha \end{bmatrix}$$

In general, we can treat each 4-byte column $\alpha(x)$ of the state as polynomials over GF(2^8), multiply them with a fixed polynomial $C(x) = 03\cdot x^3 + 01\cdot x^2 + 01\cdot x + 02$ and reduce it modulo $L(x) = x^4 + 1$ to get the new output column $\beta(x)$.

$$\beta(x) = C(x)\cdot\alpha(x) \pmod{x^4+1}$$

The fixed polynomial $C(x)$ is coprime to the reducible polynomial $L(x)$ in GF(2^8) and is therefore invertible, which is needed for decryption. For encryption, this modular multiplication can be written as the matrix multiplication:

$$\begin{bmatrix}\beta_0\\ \beta_1\\ \beta_2\\ \beta_3\end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \cdot \begin{bmatrix}\alpha_0\\ \alpha_1\\ \alpha_2\\ \alpha_3\end{bmatrix}$$

Here, $\alpha_0,...,\alpha_3$ is one input column and $\beta_0,...,\beta_3$ is the output column. The numbers 01, 02 and 03 in the fixed 4 x 4 matrix, represented in hexadecimal, were chosen based on the smallest possible values to get reliable diffusion with invertible properties and such that implementation in software became easier. This is due to the fact that multiplying a byte by 01 is the same as multiplying by the identity matrix, which does nothing. Multiplying by 02 is the same as a binary left shift by one place followed by a modular reduction with $P(x) = x^8 + x^4 + x^3 + x + 1$, which is the same as an XOR with the binary digits 100011011 (0x11B in hexadecimal) after the left shift if the leftmost bit of the original byte is 1. Multiplying by 03 can be done with a binary left shift by one place followed by an XOR with the original byte and then reduce the sum modulo $P(x)$. These are easy and fast computations for a computer, but implementation in software can also easily be done trough lookup tables.

For decryption, we use the inverse MixColumn function (InvMixColumns), which is similar to MixColumns, but instead of multiplying with $C(x)$, we multiply by the polynomial $D(x) = 0B \cdot x^3 + 0D \cdot x^2 + 09 \cdot x + 0E$. This polynomial is derived from

$$(03 \cdot x^3 + 01 \cdot x^2 + 01 \cdot x + 02) \cdot D(x) \equiv 1 \pmod{x^4 + 1}$$

where $D(x)$ is the polynomial that when multiplied by $C(x)$ makes them congruent to $1 \pmod{L(x)}$. The InvMixColumns matrix M^{-1} is the matrix version of $D(x)$ and the whole InvMixColumns step for a column $\alpha = (\alpha_0, \alpha_1, \alpha_2, \alpha_3)$ is represented by the following matrix multiplication:

$$\begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix} = \begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \cdot \begin{bmatrix} \alpha_0 \\ \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{bmatrix}$$

Similar to the ShiftRows function, a state $\alpha(x)$ becomes itself after 4 concatenating MixColumns transformations which also yields for 4 concatenating InvMixColumns transformations. When combining ShiftRows and MixColumns, then due to the linearity of the functions, they must be applied 8 times in a row on a state for it to become itself again, which we can write as

$$\alpha = (MC(SR(\alpha)))^8 = (MC^{-1}(SR^{-1}(\alpha)))^8$$

and when combined this way in an SP network forms a strong diffusion layer. Although the MixColumn function operates on seperate columns like a linear layer, it can be thought of as a non-linear diffusion layer because of the way it mixes each column. By not including the MixColumn layer in the final round, we save some computation time and it will not affect the overall security of the cipher.

3.1.5 Key Schedule

The key schedule is a recursive algorithm for expanding the main key into several independent round-keys that are used between every round function. The key schedule produces 10, 12 or 14 individual round keys according to the key length n. To make sure each round gets its own associated round-key, each round-key is produced by inputs of its previous key and a fixed array of size $4 \times n$ called *Rcon*. The *Rcon* matrix for AES-128 is shown in Table 3.7

Table 3.7: The Rcon matrix used for key expansion in AES-128

01	02	04	08	10	20	40	80	1b	36
00	00	00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00	00	00

where each 32-bit word (column) of *Rcon* is named $Rcon^0, Rcon^1, ..., Rcon^9$. We describe the key schedule for AES-128, which is the same as for the other versions except the number of round-keys. The expanded key can be viewed as a 4×44 matrix in AES-128 where each 32-bit word W is numbered from 0 to 43. This is the same as ordering 11 block states in succession where we have 1 initial block state which is the main key and the 10 others are the different round keys. We name each word (column) in the intermediate block state as W_0, W_1, W_2, W_3. Let $W^i = (W_0^i, W_1^i, W_2^i, W_3^i)$ denote an intermediate block state of the i-th round-key where each word W_i^i is column number 0, 1, 2 and 3 in the block state. Similarly, we define a new block state $W^{i+1} = (W_0^{i+1}, W_1^{i+1}, W_2^{i+1}, W_3^{i+1})$ which will be the next round-key. To produce W^{i+1} from W^i, we do the following operations:

We call $W_3^i = (W_{3,0}^i, W_{3,1}^i, W_{3,2}^i, W_{3,3}^i)$ the RotWord. The first action we perform on the RotWord is a cyclic shift of the byte positions in the word where the bytes are shifted left once and $W_{3,0}^i$ is moved to the end of the word so that the new RotWord becomes $W_3^{i'} = (W_{3,1}, W_{3,2}, W_{3,3}, W_{3,0})$. Then we perform a substitution of all the bytes in $W_3^{i'}$ using the AES s-box so that $S(W_{3,0}^{i'}) = (s(W_{3,0}^{i'}), s(W_{3,1}^{i'}), s(W_{3,2}^{i'}), s(W_{3,3}^{i'}))$ where s is the representation of 1 s-box. Now, to form the first word of the $(i+1)$-th round-key, we do the following XOR operation: $W_0^{(i+1)} = W_0^i \oplus W_3^{i'} \oplus Rcon^i$. To form the second word in the $(i+1)$-th round-key, we calculate: $W_1^{(i+1)} = W_0^{(i+1)} \oplus W_1^{i'}$. To form the third word in the $(i+1)$-th round-key, we calculate: $W_2^{(i+1)} = W_1^{(i+1)} \oplus W_2^{i'}$ and to form the fourth word in the $(i+1)$-th round-key, we calculate: $W_3^{(i+1)} = W_2^{(i+1)} \oplus W_3^{i'}$. This process repeats for the next round-keys where we use the last word in the current round-key as the RotWord.

Chapter 4

Attacks on AES

Ever since the introduction of the Rijndael algorithm and the AES selection, there have been many attempts to break the cipher. The different attack methods are often given their own names and have different performances. Some attacks methods are based on information gained from the implementation of the cipher rather than weaknesses in the cipher itself, such as Side Channel Attacks [2], Cross VM Attacks [20], Timing Attacks [1], Differential Power Analysis [13], Reply Attacks and Key Reinstallation Attacks [15]. Such attacks often requires technical knowledge of the internal operating system, network, software or hardware implementation of the cipher where vulnerabilities in the systems itself like power consumption, electromagnetic leaks, software bugs or transmission weaknesses are exploited. Although some of these attacks have been successfully demonstrated, it does not weaken the cipher itself, but rather shows that there will always be vulnerabilities in systems when the systems are not properly designed with the security in mind. In many scenarios when systems are breached, using a strong cryptographic algorithm can be critical to the security. The focus of this theses is cryptanalysis of the AES algorithm itself, where the system implementation of the cipher is of no significance.

4.1 Cryptanalysis of symmetric ciphers

Cryptanalysis is the study of analyzing ciphers and cryptosystems in general with the aim of understanding how they work and finding and improving techniques to break them. The act of breaking a crypto algorithm in a cipher is sometimes referred to as *cracking*, which stands out from the concept of *hacking* that can refer to any type of system manipulation. Although cracking or weakening a cipher is sometimes done with malicious intentions by some, the research results within the open academic cryptanalysis community are used by cryptographers to improve and strengthen a cipher for the benefit of everyone. This also includes replacing flawed algorithms that have been found to be vulnerable to any type of cryptanalysis attack before others takes advantage of them. Since AES is one of the most widely used ciphers today, it is also one of the most studied ciphers by cryptanalysts, but even though it has now been over 20 years since AES was put into use, no obvious weaknesses have been found that weaken the cipher. Some theoretical attacks on the full versions of AES 128, AES 192 and AES 256 has been proposed, but the computational complexity of these attacks is quite impractical. Because of this, AES is considered safe to use and is even used to encrypt top-secret documents, classified material and most of the Internet communications.

At least there has been some progress on round reduced versions of AES, which is just the AES algorithm, but with fewer rounds than normal. While this does not pose a direct threat to the full version of the cipher (yet), it does raise some concerns as the number of rounds begins to approach the number of rounds that the full version of AES uses. Time has shown that as technology improves, so does attacks on ciphers, and where some attack methods were assumed to apply only to a fixed number of rounds, it has recently been shown that they can be extended to cover even more rounds. Sometimes these attack methods reach limitations where they cannot be extended to multiple rounds simply because they are based on weaknesses that only appeal to properties found in a few rounds, and sometimes new techniques emerge that have not yet reached their full potential because they haven't been studied enough. It is therefore important to continue the research on such techniques that may be crucial for further analysis of AES. We will cover some of these recent techniques in the next chapters.

When analyzing a cipher, we look at certain properties in different scenarios to see how the cipher responds when manipulating it in a given way and try to see if we can get some information or find a specific pattern we can use to either predict the outcome in a certain setting or extract the key, plaintext or ciphertext without the knowledge of these. There are mainly two categories of attacks related to cryptanalysis of a cipher: distinguishing attacks and key recovery attacks.

4.1.1 Distinguishing attack

A distinguishing attack is an attack form where the goal is to distinguish a cipher from random data. Given a black box with an unknown block cipher, a distinguishing attack on the cipher can tell us which kind of cipher the black box uses. The goal of a strong cipher is to confuse a diffuse anyone who is trying to get some meaningful information from the encrypted data, so it is important that the cipher does not contain any obvious patterns that can be analyzed and understood. We can distinguish a cipher from pure randomness by analyzing the relation between different inputs or outputs in the cipher, and if the relation is predictable, we know that the cipher has some structural, non-random properties that can be exploited. In a real world scenario when the key is unknown to the attacker, this does not compromise the confidentiality of cipher, but as many key recovery attacks are developed from a distinguishing attack, it is impotent that a cipher is not distinguishable. The scenarios which these attacks applies to is where we are given some information, either a plaintext, ciphertext or a key or some of those, and wonder which encryption method that's used. For example, if we got ciphertext and do not know which encryption method is used on it, we can run a distinguishing attack on the ciphertext for a given cipher, and the result of the attack will give us an answer as to whether it is the particular cipher or not. A distinguishing attack is often the first attack method performed on a ciphertext where we do not know the encryption method used, as it becomes pointless to try to make sense out of the ciphertext when we do not know which cipher we are analyzing. When a cipher is identified by a distinguishing attack, it will be easier to perform further analysis on the ciphertext and even perform a key recovery attack if the cipher is vulnerable to that. If we can distinguish a cipher from random data faster than a brute force search, the cipher is considered broken. For this reason, modern ciphers are designed to be immune to such attacks, and considering AES, there have yet been a attack that can distinguish AES from random data.

4.1.2 Key recovery attack

A key recovery attack is an attack on a cipher where the goal is to recover the secret key that's used in the ciphertext. That way, we can read every encrypted message that uses the same key, so if a cipher is vulnerable the a key recovery attack, it is considered not safe to use. Most key recovery attacks on a ciphers require known values of plaintext and ciphertext to be performed, but some key recovery attacks can also be performed by only accessing the ciphertext if we know the encryption method used and that the cipher is weak. Most ciphers are immune to such attacks, but some symmetric ciphers have a weakness for key recovery attacks where both the ciphertext and its corresponding plaintext are known, especially when we are allowed to generate chosen plaintexts to encrypt and chosen ciphertexts to decrypt with the secret key. There is always a relation between a plaintext and its corresponding ciphertext that are dependent on the encryption key, and a usual method for key recovery attacks is to use a candidate key to encrypt or decrypt the information we have and compare the result to the original information. We can then search for differences that meets a certain requirement and reveal more and more information from the original key. This method becomes even more practical in the adaptively chosen plaintext / ciphertext ACP/ACC scenario where we can generate custom plaintexts or ciphertexts based on information gained in the attack process. Some of the key recovery attacks on round reduced versions of AES that we present in chapter 6 are build upon this principle. There are also many other ways to perform a key recovery attack and we list some of the most common attacks on round reduced version of AES in section 4.4.

4.2 How AES is designed to prevent attacks

There are various ways to make a cipher robust for attacks. In a scenario where we only know one ciphertext, one of the ways to recover the plaintext is to perform a frequency analysis on the ciphertext to see which character repeats the most. If the frequency of the characters in the plaintext is reflected in the ciphertext directly, such as in the classical substitution cipher, this is a poor design and makes the cipher vulnerable to a frequency analysis attack. The famous Enigma [21] machine solved this by always substituting a character to a different character so that the frequency pattern was destroyed. Another way to approach the plaintext from the ciphertext is just by brute-forcing all the key combinations and look at the decryption with each key to see if the plaintext is recognizable. To prevent this, a cipher should use a key space that is larger than what a computer can brute-force in a given time. If it only takes one year to brute-force all the possible key combinations for a computer today, we can be sure that a computer can do this way faster in a couple of years since the computing power always increases by the years. To have a good security margin, ciphers today use key spaces that takes longer to brute-force than our lifetimes, where time complexity is often measured in years. In AES, key lengths of 128, 192 and 256 bits are used, which makes the total key combinations of each key length 2^{128}, 2^{192} and 2^{256} respectively. For one of the fastest supercomputer today, Sunway TaihuLight [14] which can perform at 93 PetaFLOPS, it would take 1 billion billion years to crack AES-128 using a brute force attack, and considering AES-256, this would take $2.7337893 \cdot 10^{55}$ years.

So we can say with high confidence that AES is secure against pure brute-force attacks when we only know the ciphertext. But there are many other ways to attack a symmetric cipher than just brute-force, and the structure of a cipher is often based on the knowledge of these attacks so that the cipher can prevent such attacks. The most typical attack methods on symmetric ciphers is known-plaintext attacks (assuming we already know the ciphertext), chosen-plaintext attacks (where a ciphertexts from a chosen plaintext is requested), differential cryptanalysis (where differences in two or more plaintexts / ciphertexts are compared) and linear cryptanalysis (which is based on finding affine approximations to the action of the cipher). To make sure that AES was secure against such attacks, careful construction of the round functions for each round was designed, making sure that the structure of the intermediate block state gets more and more randomized.

The randomization in a cipher can be divided into two categories: confusion and diffusion. Confusion is the property of hiding the relationship between plaintext and ciphertext by making each byte in the ciphertext dependent on the entire key, so that if only one byte in the key was flipped, this change would affect most bytes in the encrypted text. In a SPN-network, the non-linear proved a high level of confusion in the cipher. The property of diffusion in a SPN-network is mostly accomplished by the byte permutation in the linear layers and have an effect to statistically change half of the bytes in the ciphertext. Since AES uses both linear and non-linear layers in every round that provides a high level of confusion and diffusion and also includes an XOR operation with the associated round-key, the level of randomness in the cipher increases for every round.

The MixColumn function in AES is one of the most interesting layers that provides an extra confusion effect by being a non-linear diffusion layer, which makes AES stand out among other ciphers. In a theoretically scenario where Mixcolumns is used in the last round, an attacker could swap the order of the last AddRoundKey function and the MixColumn function, undoing the inverse MixColumn step for free which theoretically can leave the attacker with some information from the last round-key. Although this can be shown to not add or remove any security to the cipher itself, excluding the MixColumn function in the final round prevents any structural pattern (if any) throughout the round functions to be seen in the ciphertext.

Since the number of rounds only increase the randomness in an iterative block cipher, one would think that the more rounds we have, the more secure the cipher is. Although this is true, it also affect applications that uses the encrypting by performing more slowly. Because of that, the number of rounds in AES is 10, 12 and 14, which is actually not many rounds compared to other block ciphers, but since AES provides a high level of confusion and diffusion in each round function, there is no need to increase the number of rounds.

4.3 Comparison of round reduced attacks on AES

In this section we will give a brief overview of the existing common attacks on round reduced AES and their performance. All the attacks works in the chosen plaintexts/-ciphertexts scenario (CP/CC) and some are based on the adaptively chosen plaintext/-ciphertexts (ACP/ACC) scenario where we can request custom plaintexts and ciphertexts.

4.3.1 Distinguishing attacks on round reduced AES

We begin by reviewing some common distinguishers for 3-6 rounds. We adopt the common calculation of the complexity of for the distinguishers where the data complexity is measured in a minimum number of chosen plaintexts/ciphertexts CP/CC or adaptively chosen plaintexts/ciphertexts ACP/ACC and the time complexity is measured in equivalent number of AES encryptions (E), memory accesses (M) and/or XOR operations (XOR) where 20M $\approx$1 round of AES.

Table 4.1: Comparison of Secret-Key Distinguishers for AES

Property	**Rounds**	**Data**	**Cost**	**Ref.**
Exchange Attack	3	3 ACC	2 XOR	[23]
Trun. Diff	3	$2^{4.3}$ CP	$2^{11.5}$ XOR	[7; 18]
Integral	3	2^{8} CP	2^{8} XOR	[9]
Exchange Attack	4	4 ACC	2 XOR	[23]
Imp Diff	4	$2^{16.25}$ CP	$2^{22.3}$ M	[5]
Mixture Diff.	4	2^{17} CP	$2^{22.3}$ M	[16]
Integral	4	2^{32} CP	2^{32} XOR	[9]
Multiple-of-8	4	2^{33} CP	2^{40} M	[19]
Exchange Attack	5	$2^{25.5}$ ACC	$2^{24.8}$ XOR	**Sec. 6.3**
Imp Diff	5	$2^{98.2}$ CP	2^{107} M	[5]
Struct. Diff	5	2^{33} CP	$2^{36.6}$ M	[19]
Integral	5	2^{128} CP	2^{128} XOR	[24]
Multiple-of-8	5	2^{32} CP	$2^{35.6}$ M	[19]
Variance Diff.	5	2^{34} CP	$2^{37.6}$ M	[17]
Boomerang	5	2^{39} CP/ACC	2^{39} M	[6]
Exchange Attack	6	$2^{122.83}$ ACC	$2^{121.8}$ XOR	[23].
Integral	6	$6 \cdot 2^{32}$ CP	2^{72} M	[22].
Boomerang	6	2^{71} CP/ACC	2^{71} M	[6]

There are many more distinguishers for round reduced AES that are not included in table 4.1, but we list some common ones to give an insight in the complexity of the distinguishers for the different rounds where we see that the average complexity of the distinguishers increases for every round. A computational complexity of 2^{32} is doable on a laptop in some minutes, but when the complexity gets round 2^{40}, it can take days to finish the attack as the time increases exponentially. We will cover the Exchange distinguishing attack in section 6.3.

4.3.2 Key recovery attacks on round reduced AES

We now show some common attacks for reduced AES that have a fairly low complexity that can be implemented and tested on a laptop. Although attacks on more than 5 rounds exists, they usually have a complexity that is too high to work in practice. We cover the Integral Exchange Attacks for 1-3 round in section 6.2 and the Exchange Attack on 5 rounds in section 6.4.

Table 4.2: Comparison of Key recovery attacks for round reduced AES

Attack	Rounds	Data	Computation	Memory	Ref.
Exchange	3	1CP/CC+16ACP	$16 \cdot 2^8$	$2^{20}(+2^{27})$	**Sec. 6.2**
Square	4	2^9 CP	2^6	small	[27].
Exchange	5	$2^{11.3}$ ACC	2^{31}	small	**Sec. 6.4**
Boomerang	5	2^{39} ACC	2^{39}	$2^3 3$	[6].
Imp. Polytopic	5	15 CP	2^{70}	2^{41}	[26]
MitM	5	8 CP	2^{64}	$2^5 6$	[12].
Imp. Polytopic	5	15 CP	2^{70}	2^{41}	[26]
Partial Sum	5	2^8 CP	2^{38}	small	[25]
Integral	5	2^{11} CP	$2^{45.7}$	small	[11]
Imp. Diff	5	2^{31} CP	$2^{33}(+2^{38})$	2^{38}	[16]
Mix. Diff	5	$2^{33.6}$ CP	$2^{33.3}$	2^{34}	[16]
Zero Diff.	5	$2^{29.19}$CP+2^{32}ACC	2^{31}	small	[3]

Lastly we mention some of the attacks that can in theory break 6 - 10 rounds of AES. By breaking a cipher, we mean that the attack is faster than an exhaustive brute-force attack, which is 2^{128} for AES-128 with 10 rounds.

The first attack that had a lower computational complexity than a brute-force attack on 10 rounds was the Biclique [8] attack which had a complexity of $2^{126.18}$, improving the complexity of a brute-force attack by a factor of four. There is also a Meet In The Middle attack that can break 9 out of 10 rounds. The Biclique attack can be thought of as an enhanced version of the Meet In The Middle attack. The best Square attack can break up to 8 rounds and the best Impossible Differential attack can break 7 rounds, which also the Boomerang attack can.

Chapter 5

Truncated differentials for AES

5.1 Introduction of truncated differential cryptanalysis

In cryptography, truncated differential cryptanalysis is a sub-genre within differential cryptanalysis against block ciphers that considers differences that are only partially determined between two block states while ordinary differential cryptanalysis in general is about analyzing the full difference between two states. This type of differential cryptanalysis can be used to makes predictions of one or some of the words or bytes in the block state instead of the full block.

In this chapter to combine truncated differential cryptanalysis with exchange functions of different subspaces in a block state. The exchange technique was first introduced by Eli Biham and Adi Shamir against the block cipher Skipjack (developed by the U.S. National Security Agency (NSA) in 1993) when they discovered an attack against 16 of the 32 rounds using exchange techniques between two block state which became known as The Yoyo Game [4]. The concept behind the Yoyo Game is as follows: given a plaintext pair and its corresponding ciphertext pair after encryption through a black box where the ciphertext pair has a specific property, how can we generate other plaintext pairs that makes the new ciphertexts have the same property? As there is always some structure in a block cipher, it is possible to exchange certain elements from the plaintext that makes different properties in the ciphertext in different round where there sometimes is some nested zero difference patterns. The specific differences through each round is also referred to as subspace trials, which we will cover more in detail.

This attack method is similar to the Boomerand attack [6], but with some certain differences. A main part of this thesis is to study and implement the so-called Yoyo trick (also known as adaptive exchange attack, mixture attack or exchange invariant generalized truncated differences) on 5 round AES to see if there is room for improvement. The attack is based on adaptively making new pairs of plaintexts and ciphertexts (called exchange pairs) that preserve a common property inherited from the original pair. The sum (or difference) in an exchange pair can typically be partitioned into subsets of other pairs in a set that satisfy the same property when they are closed under exchange operations within a certain subspace. A typical scenario is that a exchange pair shares a common zero difference in a subspace with other exchange pair after n rounds of ecryption that can be used to predict differences between other similar pairs in the same subspace. We will demonstrate how certain zero differences between exchange pairs and original pairs appeal directly to 4 rounds of AES that can be used to attack 5 rounds based on properties of the MixColumn function.

This chapter focuses mainly on introducing the concepts and cryptographic tools used in the attacks presented in chapter 6. We start by describing how the exchange functions work and its different properties.

5.2 Adaptive exchange functions

Let α be the representation of the block state consisting of $n = 4$ words: $\alpha_0, \alpha_1, \alpha_2, \alpha_3$ where each word α_i refers to a specific subspace in the state matrix over $\mathbb{F}_q^n$ where $q = 2^k$. An exchange function is simply a word-swapping operation between two block states that leaves us with a new pair of block states.

Definition of the exchange function:
For a vector $v \in \mathbb{F}_2^n$ and a pair of states $(\alpha, \beta) \in \mathbb{F}_q^n$ we define a new pair of states $(\alpha'$, $\beta') \in \mathbb{F}_q^n$ by interchanging words between α and β according to the binary coefficients of v.
Let $\alpha' = \rho^v(\alpha, \beta)$ and $\beta' = \rho^v(\beta, \alpha)$ where

$$\rho^v(\alpha,\beta)_i = \begin{cases} \alpha_i & \text{if } v_i = 1 \\ \beta_i & \text{if } v_i = 0 \end{cases}$$

$$\rho^v(\beta,\alpha)_i = \begin{cases} \beta_i & \text{if } v_i = 1 \\ \alpha_i & \text{if } v_i = 0 \end{cases}$$

which also can be written as: $\rho^v(\alpha,\beta)_i = \alpha_i v_i \oplus \beta_i(v_i \oplus 1)$, $\rho^v(\beta,\alpha)_i = \beta_i v_i \oplus \alpha_i(v_i \oplus 1)$. The new pair $(\alpha', \beta') = (\rho^v(\alpha,\beta), (\rho^v(\beta,\alpha))$ is then called a Yoyo pair.

Example: Let v = (0110), $\alpha = (\alpha_0, \alpha_1, \alpha_2, \alpha_3)$, $\beta = (\beta_0, \beta_1, \beta_2, \beta_3)$. Then

$$\alpha' = \rho^{0110}(\alpha,\beta) = (\beta_0, \alpha_1, \alpha_2, \beta_3)$$
$$\beta' = \rho^{0110}(\beta,\alpha) = (\alpha_0, \beta_1, \beta_2, \alpha_3)$$

If we let $\bar{v}$ be the complement of v, which from the example would be $\bar{v}$ = (1001), it is easy to see that $\rho^{\bar{v}}(\alpha,\beta) = \rho^v(\beta,\alpha)$ and also $(\rho^v(\alpha,\beta), \rho^v(\beta,\alpha)) = (\rho^{\bar{v}}(\alpha,\beta), \rho^{\bar{v}}(\beta,\alpha))$, implying that $\bar{v}$ and v results in the same exchange pair. There are in total 2^n swap combinations, but only 2^{n-1} swaps results in unique pairs including the original pair. For AES, where the block size is 128 bits, n = 4 and thus the number of possible unique exchange pairs is 2^{4-1} = 8. Table 4.1 lists all the possible exchange pairs for $v \in \mathbb{F}_2^4$.

If two states α and β are different in only 1 out of 4 words, we can not make a new exchange pair that differs from α and β since swapping the only different word among them only makes $\alpha = \beta$ and $\beta = \alpha$, which still is the same pair $(\alpha, \beta) = (\beta, \alpha)$. This also applies when there is no difference between α and β. When the difference is 2,

Table 5.1: All possible exchange pairs for $v \in \mathbb{F}_2^4$

v	$\alpha' = \rho^v(\alpha,\beta)$	$\beta' = \rho^v(\beta,\alpha)$	$\bar{v}$	$\alpha' = \rho^{\bar{v}}(\alpha,\beta)$	$\beta' = \rho^{\bar{v}}(\beta,\alpha)$
0000	$(\beta_0,\beta_1,\beta_2,\beta_3)$	$(\alpha_0,\alpha_1,\alpha_2,\alpha_3)$	1111	$(\alpha_0,\alpha_1,\alpha_2,\alpha_3)$	$(\beta_0,\beta_1,\beta_2,\beta_3)$
0001	$(\beta_0,\beta_0,\beta_0,\alpha_3)$	$(\alpha_0,\alpha_1,\alpha_2,\beta_3)$	1110	$(\alpha_0,\alpha_1,\alpha_2,\beta_3)$	$(\beta_0,\beta_1,\beta_2,\alpha_3)$
0010	$(\beta_0,\beta_1,\alpha_2,\beta_3)$	$(\alpha_0,\alpha_1,\beta_2,\alpha_3)$	1101	$(\alpha_0,\alpha_1,\beta_2,\alpha_3)$	$(\beta_0,\beta_1,\alpha_2,\beta_3)$
0011	$(\beta_0,\beta_1,\alpha_2,\alpha_3)$	$(\alpha_0,\alpha_1,\beta_2,\beta_3)$	1100	$(\alpha_0,\alpha_1,\beta_2,\beta_3)$	$(\beta_0,\beta_1,\alpha_2,\alpha_3)$
0100	$(\beta_0,\alpha_1,\beta_2,\beta_3)$	$(\alpha_0,\beta_1,\alpha_2,\alpha_3)$	1011	$(\alpha_0,\beta_1,\alpha_2,\alpha_3)$	$(\beta_0,\alpha_1,\beta_2,\beta_3)$
0101	$(\beta_0,\alpha_1,\beta_2,\alpha_3)$	$(\alpha_0,\beta_1,\alpha_2,\beta_3)$	1010	$(\alpha_0,\beta_1,\alpha_2,\beta_3)$	$(\beta_0,\alpha_1,\beta_2,\alpha_3)$
0110	$(\beta_0,\alpha_1,\alpha_2,\beta_3)$	$(\alpha_0,\beta_1,\beta_2,\alpha_3)$	1001	$(\alpha_0,\beta_1,\beta_2,\alpha_3)$	$(\beta_0,\alpha_1,\alpha_2,\beta_3)$
0111	$(\beta_0,\alpha_1,\alpha_2,\alpha_3)$	$(\alpha_0,\beta_1,\beta_2,\beta_3)$	1000	$(\alpha_0,\beta_1,\beta_2,\beta_3)$	$(\beta_0,\alpha_1,\alpha_2,\alpha_3)$

then only 2 new exchange pairs can be generated that differs from the original pair (α,β). With a difference in 3 out of 4 words, we can generate up to 6 new unique pairs and when all the words are different between α and β, in total 8 new unique pairs can be generated. Therefore, to produce a new unique exchange pair between two random states, we need them to be distinct in at least two words, which happens with probability $(1-2^{-94})$.

5.2.1 Properties of the exchange function

The exchange function has some interesting properties that allow us to use this function as a tool in cryptanalysis for AES. To better understand these properties, we need to be familiar with some basic terminologies and notations. We begin by defining The Hemming weight.

Definition of the Hemming weight:
For a vector $x = (x_0, x_1, ..., x_{n-1})$, we define the Hemming weight $\omega t(x)$ as the number of non-zero components of x

Example:
Let $x = (x_0, x_1, x_2, x_3)$ = (5B, 9A, 07, 00). Then the Hemming weight $\omega t(x) = 3$.

If every byte of the vector is zero, then the Hemming weight of that vector is also zero. We let $\omega t_c(\alpha)$ denote the number of non-zero vectors in a state α. A zero column is when every byte in a column is zero, thus ωt_c is also called the column weight. In a similar way, we define the diagonal weight $\omega t_d(\alpha)$ as the number of non-zero diagonals in a state α, where diagonal 0 to 3 is defined as $\alpha_{j,(i+j) \bmod 4}$ where i is the diagonal number and j increases from 0 to 3 for every byte position. In differential cryptanalysis we are often interested in a certain difference between states, or more precisely the zero difference. If two columns in different states are the same, they sum up to zero. This brigs us to the definition of the zero difference pattern of a state.

Definition of the zero difference pattern:
The zero difference pattern is simply a binary vector that indicate which words in a state that is zero. For a state $\alpha = (\alpha_0, \alpha_1, \alpha_2, \alpha_3) \in \mathbb{F}_q^n$. We define the zero difference pattern

$$\nu(\alpha) = (z_0, z_1, z_2, z_3) \in \mathbb{F}_2^n \textit{ where } z_i \begin{cases} 1 & \textit{if } \alpha_i = 0 \\ 0 & \textit{otherwise} \end{cases}$$

Definition of the activity pattern:
The activity pattern is the compliment of the zero difference pattern. While the zero difference pattern indicates a 1 when α_i is zero, the activity pattern indicates a 0 when α_i is zero. For a state $\alpha = (\alpha_0, \alpha_1, \alpha_2, \alpha_3) \in \mathbb{F}_q^n$. We define the activity pattern

$$\bar{\nu}(\alpha) = (z_0, z_1, z_2, z_3) \in \mathbb{F}_2^n \textit{ where } z_i \begin{cases} 0 & \textit{if } \alpha_i = 0 \\ 1 & \textit{otherwise} \end{cases}$$

Example: Let $\alpha = (\alpha_0, \alpha_1, 0, \alpha_3)$. Then the zero difference pattern for α is $\nu(\alpha)$ = (0010). The activity pattern is then $\bar{\nu}(\alpha)$ = (1101).

With these definitions in mind, we will explore some interesting properties of the exchange function. A simple one to observe is that if we have a pair of states (α, β) with a certain Hemming weight $\omega t(x)$ in their difference $\alpha \oplus \beta$, if we then make a new exchange pair (α', β'), we see that the Hemming weight in the difference $\alpha' \oplus \beta'$ is the same as $\omega t(x)$. We can keep on and make more and more exchange pairs and observe that $\omega t(\alpha \oplus \beta) = \omega t(\alpha' \oplus \beta')$ always holds because for all $v \in \mathbb{F}_2^n$ we have that $\omega t(\rho^v(\alpha, \beta) \oplus \rho^v(\beta, \alpha)) = \omega t(\rho^v(\beta, \alpha) \oplus \rho^v(\alpha, \beta))$. It also holds that $\nu(\alpha \oplus \beta) = \nu(\alpha' \oplus \beta')$. In other words, the zero difference pattern between two different states is preserved after applying the exchange function on them. This follows directly from that $\alpha' \oplus \beta' = \alpha \oplus \beta$, which can be proven by

$$\rho^v(\alpha, \beta)_i \oplus \rho^v(\beta, \alpha)_i = \begin{cases} \alpha_i \oplus \beta_i & \text{if } v_i = 1 \\ \beta_i \oplus \alpha_i & \text{if } v_i = 0. \end{cases}$$

Lets look at what happens with a exchange pair if we apply other functions to α and β. Let S be a nonlinear permutation s-box over $\mathbb{F}_q^n$ formed by concatenating n smaller s-boxes over $\mathbb{F}_q$ so that $S(\alpha) = (s(\alpha_0), s(\alpha_1), s(\alpha_2), s(\alpha_3))$. Then for two different states $\alpha \neq \beta$ it follows from the non-linearity of S that $S(\alpha) \oplus S(\beta) \neq S(\alpha \oplus \beta)$ since $\alpha_i \oplus \beta_i = 0$ if and only if $S(\alpha_i) \oplus S(\beta_i) = 0$, but we see that $\nu(\alpha \oplus \beta) = \nu(S(\alpha) \oplus S(\beta))$, thus the zero difference pattern in the difference is preserved trough S. This also means that $\nu(S(\alpha) \oplus S(\beta)) = \nu(S(\rho^v(\alpha, \beta)) \oplus S(\rho(\beta, \alpha)))$ since the zero difference is preserved trough the exchange function as well. In fact, the exchange function commutes directly with the s-box layer by operating independently on individual words in the block state so that $S(\rho^v(\alpha, \beta)) = \rho^v(S(\alpha), S(\beta))$ and $S(\rho^v(\beta, \alpha)) = \rho^v(S(\beta), S(\alpha))$, leading to $S(\alpha) \oplus S(\beta) = S(\alpha') \oplus S(\beta')$, which can easily be proven by

$$s(\rho^v(\alpha, \beta)_i) \oplus s(\rho^v(\beta, \alpha)_i) = \begin{cases} s(\alpha_i) \oplus s(\beta_i) & \text{if } v_i = 1 \\ s(\beta_i) \oplus s(\alpha_i) & \text{if } v_i = 0. \end{cases}$$

We now let L be a linear transformation $L(\alpha) = L(\alpha_0, \alpha_1, \alpha_2, \alpha_3)$ on the block cipher acting on n words over $\mathbb{F}_q^n$. Due to the linearity of L we have that $L(\alpha) \oplus L(\beta) = L(\alpha \oplus \beta)$ always holds. Since $\alpha \oplus \beta = \rho^v(\alpha, \beta) \oplus \rho^v(\beta, \alpha) = \alpha' \oplus \beta'$ it follows directly that also $L(\alpha) \oplus L(\beta) = \alpha' \oplus \beta'$ and thus $L(\alpha) \oplus L(\beta) = L(\alpha') \oplus L(\beta')$. We see that this relation is similar to the S functions relation $S(\alpha) \oplus S(\beta) = S(\alpha') \oplus S(\beta')$. Lets see what happens if we try to combine the S function and the L function together. Since $S(\alpha) \oplus s(\beta) = S(\rho^v(\alpha,\beta)) \oplus S(\rho^v(\beta,\alpha))$ holds, we clearly have that $L(S(\alpha)) \oplus L(S(\beta)) = L(S(\rho^v(\alpha,\beta))) \oplus L(S(\rho^v(\beta,\alpha)))$ also holds since applying any linear transformation after a linear relation does not effect the linearity.

$$L(S(\rho^v(\alpha,\beta)_i)) \oplus L(S(\rho^v(\beta,\alpha)_i)) = \begin{cases} L(S(\alpha_i)) \oplus L(S(\beta_i)) & \text{if } v_i = 1 \\ L(S(\beta_i)) \oplus L(S(\alpha_i)) & \text{if } v_i = 0 \end{cases}$$

However, $S(L(\alpha)) \oplus S(L(\beta)) = S(L(\rho^v(\alpha,\beta))) \oplus S(L(\rho^v(\beta,\alpha)))$ does not generally hold because applying a non-linear function after L will destroy the linearity, but since the zero difference pattern between two pairs does not change when we apply S or L on them $\nu(S(\alpha) \oplus S(\beta)) = \nu(S(\rho^v(\alpha,\beta)) \oplus S(\rho^v(\beta,\alpha)))$, $\nu(L(\alpha) \oplus L(\beta)) = \nu(L(\rho^v(\alpha,\beta)) \oplus L(\rho^v(\beta,\alpha)))$, we have that $\nu(S(L(\alpha)) \oplus \nu(S(L(\beta))) = \nu(S(L(\rho^v(\alpha,\beta))) \oplus S(L(\rho^v(\beta,\alpha))))$ holds. Same goes for $\nu(L(S(\alpha)) \oplus \nu(L(S(\beta))) = \nu(L(S(\rho^v(\alpha,\beta))) \oplus L(S(\rho^v(\beta,\alpha))))$ which directly implies that $\nu(S(L(S(\alpha))) \oplus \nu(S(L(S(\beta)))) = \nu(S(L(S(\rho^v(\alpha,\beta)))) \oplus S(L(S(\rho^v(\beta,\alpha)))))$ also holds. Adding different functions one after another to a block state like this is called concatenation. For the sake of notation we often put a "$\circ$" between the functions where the rightmost function is the first one to be applied on the block state. The relation above is then written as $\nu(S \circ L \circ S(\alpha) \oplus S \circ L \circ S(\beta)) = \nu(S \circ L \circ S(\alpha') \oplus S \circ L \circ S(\beta'))$. As we will see in many of the exchange attacks, this relation is the primary property of AES we take advantage when we generate adaptive chosen plaintext and ciphertext.

5.3 Simplification of the AES rounds

The L and S notation for the linear and nonlinear layers are used to describe the different rounds in SPN networks and also in AES where $S = SB \circ MC \circ SB$ and $L = SR \circ MC \circ SR$. One round of AES is written as $AK \circ MC \circ SR \circ SB$, but the AddRoundKey AK is usually omitted in terms of differential cryptanalysis, as applying this function to two different states does not change the difference between the states.

Two rounds of AES can therefore be expressed as $R^{2'} = (MC \circ SR \circ SB)^2$. Although the order of $MC \circ SR \circ SB$ is given in the definition of AES, we can actually mix the order of these in a certain way without causing any change. Since the SubBytes function works individually on each byte and the ShiftRows function only changes the position of the bytes in the state, it doesn't matter what order we do it in. We can therefore paraphrase two rounds as $R^{2'} = MC \circ SR \circ SB \circ MC \circ SB \circ SR$ by switching positions between SB and SR in the first round. We can now substitute $SB \circ MC \circ SB$ with only S so we get $R^{2'} = MC \circ SR \circ S \circ SR$. Because the first ShiftRows function does not affect the difference between two states we can now also omit this function which leaves us with $R^2 = MC \circ SR \circ S$.

While three rounds of AES is initially $R^{3'} = (MC \circ SR \circ SB)^3$, it can be simplified down to a shorter expression by removing the linear SR layer before the first s-box and the linear layer $(MC \circ SR)$ after the last s-box layer to get the following:

$$\begin{aligned}
R^{3'} &= (MC \circ SR \circ SB) \circ (MC \circ SR \circ SB) \circ (MC \circ SR \circ SB) \\
R^{3'} &= MC \circ SR \circ (SB \circ MC \circ SR) \circ (SB \circ MC \circ SB) \circ SR \\
R^{3'} &= MC \circ SR \circ Q \circ S \circ SR \\
R^{3} &= Q \circ S
\end{aligned}$$

where $Q = SB \circ MC \circ SR$. In a similar way, the three rounds in reverse can be expressed as $R^{-3} = S^{-1} \circ Q^{-1}$ where $Q^{-1} = SR^{-1} \circ MC^{-1} \circ SB^{-1}$.

To simplify four rounds, we use the same reduction by removing the leading and trailing linear layers before the first and after the last s-box to end up with $R^4 = S \circ L \circ S$. In SPN networks like AES, we often simplify the rounds by compositions of linear and non-linear layers and this SLS-form applies directly to even number of rounds, which makes six rounds equal to $R^6 = S \circ L \circ S \circ L \circ S$ and so on. For odd numbers of rounds, we use the same SLS-system and usually replace the first linear layer with a round $Q' = SR \circ MC \circ SB$, which makes five rounds equal to $R^5 = S \circ L \circ S \circ Q'$ and so on.

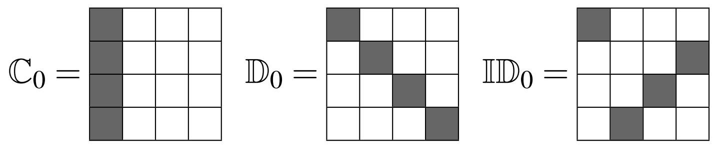
5.4 Subspace trails

In differential cryptanalysis, there is an interest in differences within a set of block states that share a common property. Many differential attacks on block ciphers is based on bruteforcing all possible combinations of a particular vector space V in a block state, often called a subspace. One can think of the whole block state as one big space over $\mathbb{F}_q^{4\times 4}$ where we can divide the block state into several subspaces. The definition of a subspace is as follows:

Definition of subspaces
For a binary vector $z \in \mathbb{F}_2^4$ of weight t let V_z denote the subspace of $q^{4\cdot(4-t)}$ states $x = (x_0, x_1, x_2, x_3)$ formed by setting x_i to all possible values of $\mathbb{F}_q^4$ if $z_i = 0$ or zero otherwise.

Example:
Let $t = 1$, $z = (1,0,0,0)$ and $V_z = V_{(1,0,0,0)}$ so that $q^{4\cdot(4-t)} = 2^{8^{4\cdot(4-3)}} = 256^4$. Then subspace $V_{1,0,0,0}$ represents all 2^{32} different states x formed by setting all of the bytes in the first column $x_{0,0}, x_{0,1}, x_{0,2}, x_{0,3}$ to all possible 2^{32} values and x_1, x_2, x_3 to zero.

Vector spaces in columns is also referred to as column spaces $\mathbb{C}_i$ where the column space $\mathbb{C}_i =< x_{0,i}, x_{1,i}, x_{2,i}, x_{3,i} > \mid \forall\ x \in \mathbb{F}_q$ represents all possible combinations in column i. Based on the column space, we define the diagonal space $\mathbb{D}_i$, the inverse-diagonal space $\mathbb{ID}_i$ and the mixed space $\mathbb{M}_i$ as $\mathbb{D}_i = SR^{-1}(\mathbb{C}_i)$, $\mathbb{ID}_i = SR(\mathbb{C}_i)$, $\mathbb{M}_i = MC(\mathbb{ID}_i)$ where $\mathbb{C}_0$, $\mathbb{D}_0$ and $\mathbb{ID}_0$ is represented as



and $\mathbb{M}_0$ on the other hand, can be represented as the symbolic matrix:

$$\mathbb{M}_0 = \begin{array}{|c|c|c|c|}\hline 2\cdot x_1 & x_4 & x_3 & 3\cdot x_2 \\ \hline x_1 & x_4 & 3\cdot x_3 & 2\cdot x_2 \\ \hline x_1 & 3\cdot x_4 & 2\cdot x_3 & x_2 \\ \hline 3\cdot x_1 & 2\cdot x_1 & x_3 & x_2 \\ \hline \end{array}\,.$$

Each of these subspaces have a dimension of $4\cdot|I|$ where $|I| \subseteq \{0,1,2,3\}$ and higher dimensions are made by adding them together. For a state $\alpha = (\alpha_0, \alpha_1, \alpha_2, \alpha_3)$ with activity pattern $(1,0,0,0)$, there can be generated a set $\mathcal{P}$ of 2^{32} other states in subspace $\mathbb{C}_0 = V_{1,0,0,0}$ which have the exact same zero difference as α. However, for a block state α where each byte in the state has a random non-zero value, there cannot be generated a set $\mathcal{P}$ in subspace $V_{1,0,0,0}$ with the same zero difference as α, but XOR-ing α with each state x in subspace $V_{1,0,0,0}$ defines a new subspace $(\alpha \oplus x)\ |x \in V_{1,0,0,0}$ where $(\alpha \oplus x)$ is not in $V_{1,0,0,0}$, but in its own particular subspace. The new subspace $V \oplus \alpha$ is then called a coset of the subspace V. This means that any state defines its own coset of a subspace according to another subspace.

We can also define a new subspace by applying any function F to the coset $(V \oplus \alpha)$. If $V \oplus \alpha$ spans the same subspace as $F(V \oplus \alpha)$, then $(V \oplus \alpha)$ is called an invariant coset of V for the function F. For instance, if we let F be the MixColumn function, then for any state $\alpha = (\alpha_0, \alpha_1, \alpha_2, \alpha_3)$ where all the bytes in α_0 are active and the rest is initialized to zero, we have that e.g. $\mathbb{C}_0 \oplus \alpha$ and $\mathbb{M}_0 \oplus \alpha$ are both invariant cosets of the subspace $\mathbb{C}_0$ for the MixColumn function since applying the MixColumn function on a column space only change the coset. If we rather apply the SubBytes function on a subspace, then, since SubBytes is a bijective mapping function that operates on individual bytes, it maps a coset of column and diagonal spaces to another coset of column and diagonal spaces. The ShiftRows function is used to change a coset of a diagonal space to a coset of a column space and from a coset of column space to a coset of an inverse-diagonal space.

Invariant subspace cryptanalysis is of particular interest when it comes to iterative key-alternating block ciphers like AES as adding a secret key K to a coset $V \oplus \alpha$ spans a key-dependent subspace. For two different block states α and α' that meets the condition $F(V \oplus \alpha) = V \oplus \alpha'$ and a key $K \in V \oplus (\alpha \oplus \alpha')$, it follows that we get an iterative invariant subspace $F(V \oplus \alpha) \oplus K = V \oplus \alpha$ dependent on the secret key. A set of subspaces $(V_1, V_2, ..., V_{r+1})$ that meets this condition is referred to as subspace trail.

Definition of subspace trail
Let $(V_1, V_2, ..., V_{r+1})$ denote a set of $r+1$ subspaces with dimension $dim(V_i) \leq dim(V_{i+1})$. For each $i = 1, ..., r$ and for each $\alpha_i \in V_i^{\perp}$ there exist $\alpha_{i+1} \in V_{i+1}^{\perp}$ such that

$$F_K(V_i \oplus \alpha_i) \subseteq V_{i+1} \oplus \alpha_{i+1},$$

then $(V_1, V_2, ..., V_{r+1})$) is subspace trail of length r for the function F_K. If all the previous relations hold with equality, the trail is called a constant-dimensional subspace trail.

If we divide the block state into several subsaces of unlike dimension, we can think of subspace trails as cosets of a plaintext subspaces that encrypts to proper subspaces over several rounds where the most typical scenario is that plaintexts within low dimention subspaces encrypts to higher dimension subspaces for each round. For example,

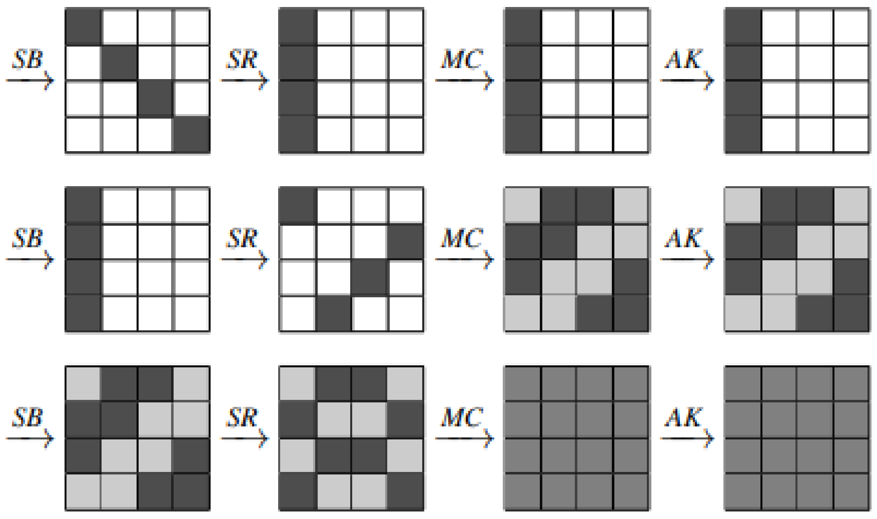


Figure 5.1: Subspace trails from $\mathbb{D}_0$ throughout the encryption process

a plaintext within the diagonal space $\mathbb{D}_i$ maps to a column space $\mathbb{C}_i$ after encryption of one round and a plaintext within the column space $\mathbb{C}_i$ maps to a mixed space $\mathbb{M}_i$ after encryption of one round. The trails are clearly seen by the active bytes in the intermediate ciphertext when encrypting a plaintext which is only active in one subspace. These properties combined with the exchange function becomes quite useful in differential cryptanalysis in the adaptive chosen plaintext/ciphertext setting as the exchange function preserves the differences in subspace trails between a set of plaintexts. Although the clear pattern seems to disappear after three rounds, we can use the exchange function on exchange invariant sets to combine two or more trails together to form longer trails that preserve a predictable structure.

In chapter 6 we will cover some round reduced techniques which takes advantage of particular subspace trails and exploits properties of the MixColumn function.

5.5 Different exchanges

In some of the attacks on AES that we will introduce, we use a simplified swapping technique called *SimpleSwap*, which simply swaps the first word that differs between two states and returns a new state. The following algorithm explains the *SimpleSwap* function:

Algorithm 1 Swaps the first different word between two states

```
Input: Two states $\alpha$ and $\beta$ where $\alpha \neq \beta$
Output: A new state $\alpha'$
function $SimpleSwap(\alpha, \beta)$
1: $\alpha' \leftarrow \beta$
2: for $i$ from 0 to 3 do
3:     if $(\alpha_i \neq \beta_i)$ then
4:         $\alpha'_i \leftarrow \alpha_i$
5:         return $\alpha'$
6:     end if
7: end for
end function
```

To make an exchange pair with the *SimpleSwap* function, wee need to apply it twice, once with input (α, β) to get α' and one with input (β, α) to get β'. There are also other variants of the Yoyo function with other swapping techniques that we will take a closer look at. So far we have only exchanged columns between block states to form a new pair called a yoyo pair. In theory we can do any swap operation between states to form a new pair, e.g swapping individual bytes between the states, or swapping rows between them, but there are certain exchange functions with some unique properties that we will use in our cryptanalysis. To better explain these exchange functions it is useful to define the column exchange difference first.

Definition of the column exchange difference
For a vector $v \in \mathbb{F}_2^4$ and a pair of states (α, β) we define the column exchange difference $\Delta_v^{\alpha,\beta} \in \mathbb{F}_2^{4x4}$ where the i-th column is defined by

$$(\Delta_v^{\alpha,\beta})i = (\alpha_i \oplus \beta_i)v_i$$

where α_i and β_i is the i-th columns of α and β.

Example: Let v = (1010) and

$$\alpha = \begin{array}{|c|c|c|c|}\hline 11 & 11 & 11 & 11 \\ \hline 11 & 11 & 11 & 11 \\ \hline 11 & 11 & 11 & 11 \\ \hline 11 & 11 & 11 & 11 \\ \hline \end{array}, \quad \beta = \begin{array}{|c|c|c|c|}\hline 22 & 22 & 22 & 22 \\ \hline 22 & 22 & 22 & 22 \\ \hline 22 & 22 & 22 & 22 \\ \hline 22 & 22 & 22 & 22 \\ \hline \end{array}, \quad \text{then } (\Delta_{1010}^{\alpha,\beta}) = \begin{array}{|c|c|c|c|}\hline 33 & 00 & 33 & 00 \\ \hline 33 & 00 & 33 & 00 \\ \hline 33 & 00 & 33 & 00 \\ \hline 33 & 00 & 33 & 00 \\ \hline \end{array}$$

The following formulas is used to calculate the column exchange difference $\Delta_v^{\alpha,\beta}$:

$(\Delta_v^{\alpha,\beta})_0 = (\alpha \oplus \beta)v_0$, $(\Delta_v^{\alpha,\beta})_1 = (\alpha \oplus \beta)v_1$, $(\Delta_v^{\alpha,\beta})_2 = (\alpha \oplus \beta)v_2$, $(\Delta_v^{\alpha,\beta})_3 = (\alpha \oplus \beta)v_3$. Notice that $\Delta_v^{\alpha,\beta} = \Delta_v^{\beta,\alpha}$ as the column exchange difference is just the sum between α and β where binary vector v is 1 and zero otherwise. If we let $\omega t_c(x)$ be the number of non-zero columns in x, there are $2^{\omega t_c(\alpha \oplus \beta)}$ possible column exchange differences for a pair of states. From the column exchange difference we define three different exchange operations: column exchange, diagonal exchange and mixed exchange.

Definition of the column exchange
For a vector $v \in \mathbb{F}_2^4$ and a pair of states (α, β) we define the column exchange

$$\rho_c^v(\alpha,\beta) = \alpha \oplus \Delta_v^{\alpha,\beta}, \;\; \rho_c^v(\beta,\alpha) = \beta \oplus \Delta_v^{\alpha,\beta}.$$

Notation wise, this is the most straight forward exchange as this can also be written as $\rho_c^v(\alpha,\beta)_i = \alpha_i v_i \oplus \beta_i(v_i \oplus 1)$, $\rho_c^v(\beta,\alpha)_i = \beta_i v_i \oplus \alpha_i(v_i \oplus 1)$. The column exchange is simply an exchange function where we form a new state by swapping individual columns among two states according to the value of the binary vector v. **Example**: Let v = (1000) and

$$\alpha = \begin{array}{|c|c|c|c|}\hline 11 & 11 & 11 & 11 \\ \hline 11 & 11 & 11 & 11 \\ \hline 11 & 11 & 11 & 11 \\ \hline 11 & 11 & 11 & 11 \\ \hline \end{array}, \;\; \beta = \begin{array}{|c|c|c|c|}\hline 22 & 22 & 22 & 22 \\ \hline 22 & 22 & 22 & 22 \\ \hline 22 & 22 & 22 & 22 \\ \hline 22 & 22 & 22 & 22 \\ \hline \end{array}, \;\; \text{then } (\rho_c^{1000}(\alpha,\beta) = \begin{array}{|c|c|c|c|}\hline 22 & 11 & 11 & 11 \\ \hline 22 & 11 & 11 & 11 \\ \hline 22 & 11 & 11 & 11 \\ \hline 22 & 11 & 11 & 11 \\ \hline \end{array}$$

Definition of the diagonal exchange
For a vector $v \in \mathbb{F}_2^4$ and a pair of states (α, β) we define the column exchange

$$\rho_d^v(\alpha,\beta) = \alpha \oplus SR^{-1}(\Delta_v^{SR(\alpha),SR(\beta)}), \;\; \rho_d^v(\beta,\alpha) = \beta \oplus SR^{-1}(\Delta_v^{SR(\alpha),SR(\beta)}).$$

Instead of swapping columns, the diagonal exchange is performed by applying the inverse ShiftRows function on the column exchange difference between $SR(\alpha)$ and $SR(\beta)$ and then adding one of the states. We can also write the diagonal exchange in terms of the column exchange like so: $\rho_d^v(\alpha,\beta) = SR^{-1}(p_c^v(SR(\alpha),SR(\beta)))$. This is equivalent to swapping individual diagonals among two states according to the value of the binary vector v. **Example**: Let v = (1000) and

$$\alpha = \begin{array}{|c|c|c|c|}\hline 11 & 11 & 11 & 11 \\ \hline 11 & 11 & 11 & 11 \\ \hline 11 & 11 & 11 & 11 \\ \hline 11 & 11 & 11 & 11 \\ \hline \end{array}, \;\; \beta = \begin{array}{|c|c|c|c|}\hline 22 & 22 & 22 & 22 \\ \hline 22 & 22 & 22 & 22 \\ \hline 22 & 22 & 22 & 22 \\ \hline 22 & 22 & 22 & 22 \\ \hline \end{array}, \;\; \text{then } (\rho_d^{1000}(\alpha,\beta) = \begin{array}{|c|c|c|c|}\hline 22 & 11 & 11 & 11 \\ \hline 11 & 22 & 11 & 11 \\ \hline 11 & 11 & 22 & 11 \\ \hline 11 & 11 & 11 & 22 \\ \hline \end{array}$$

Definition of the mixed exchange
For a vector $v \in \mathbb{F}_2^4$ and a pair of states (α, β) we define the column exchange

$$\rho_m^v(\alpha,\beta) = \alpha \oplus L(\Delta_v^{L^{-1}(\alpha)),L^{-1}(\beta}), \;\; \rho_m^v(\beta,\alpha) = \beta \oplus L(\Delta_v^{L^{-1}(\alpha)),L^{-1}(\beta}))$$

where $L = MC \circ SR$ *and* $L^{-1} = SR^{-1} \circ MC^{-1}$

We can also write $\rho_m^v(\alpha,\beta) = \alpha \oplus MC \circ SR(\Delta_v^{SR^{-1} \circ MC^{-1}(\alpha)),SR^{-1} \circ MC^{-1}(\beta}))$ in terms of the column exchange: $p_m^v(\alpha,\beta) = MC \circ SR(\rho_c^v(SR^{-1} \circ MC^{-1}(\alpha), SR^{-1} \circ MC^{-1}(\beta))$. This exchange operation is not as straight forward as the previous two as we can not just swap individual columns or diagonals among the states because the MixColumns function will change the bytes in both block states. Instead of forming a new state which looks like a direct combinations of two other states, the mixed exchange looks a little more randomized.
Example: Let v = (1000) and

$$\alpha = \begin{array}{|c|c|c|c|}\hline 11 & 11 & 11 & 11 \\ \hline 11 & 11 & 11 & 11 \\ \hline 11 & 11 & 11 & 11 \\ \hline 11 & 11 & 11 & 11 \\ \hline \end{array}, \quad \beta = \begin{array}{|c|c|c|c|}\hline 22 & 22 & 22 & 22 \\ \hline 22 & 22 & 22 & 22 \\ \hline 22 & 22 & 22 & 22 \\ \hline 22 & 22 & 22 & 22 \\ \hline \end{array}, \quad \text{then } (\rho_m^{1000}(\alpha,\beta) = \begin{array}{|c|c|c|c|}\hline 77 & \colorbox{lightgray}{22} & \colorbox{lightgray}{22} & 44 \\ \hline \colorbox{lightgray}{22} & \colorbox{lightgray}{22} & 44 & 77 \\ \hline \colorbox{lightgray}{22} & 44 & 77 & \colorbox{lightgray}{22} \\ \hline 44 & 77 & \colorbox{lightgray}{22} & \colorbox{lightgray}{22} \\ \hline \end{array}$$

Common for all the exchange operations is that the difference between α and β does not change when we apply any of the exchange functions them, i.g $\alpha \oplus \beta$

$$\begin{aligned} &= \rho_c^v(\alpha,\beta) \oplus \rho_c^v(\beta,\alpha) \\ &= \rho_d^v(\alpha,\beta) \oplus \rho_c^v(\beta,\alpha) \\ &= \rho_m^v(\alpha,\beta) \oplus \rho_c^v(\beta,\alpha) \end{aligned}$$

The relation between these exchange operations is that a diagonal exchange between two states after 1 encryption round equals a column exchange after 2 rounds which equals a mixed exchange after 3 rounds. We will use this principle in the attacks on AES in chapter 6

Chapter 6

Cryptanalysis of AES

In this chapter, we will cover in detail some cryptanalysis attacks on round reduced AES and how they work in practice. The attacks in the section 6.1 and 6.2 are self-made attacks based on own analysis and section 6.3 and 6.4 covers attacks that are newly discovered. The techniques used in the attacks are based on the material covered in Chapter 5. All the attacks presented in this chapter are practical attacks that are easy to implement and have low data and computational complexity.

6.1 A key recovery on 1 round

In this section we will cover a self-made attack based on own cryptanalysis. We start by introducing a key recovery attack on the first round of AES that only needs 1 non-adaptive plaintext / ciphertext pair.

A general round in AES is a composition $AK \circ MC \circ SR \circ SB$ where the round key addition is the only function that makes the round unique. We can therefore merge $MC \circ SR \circ SB$ together to one big function F since these are known functions. Let K be any key value and α an intermediate state in the block cipher where β is the result of the encryption of α through one round function so that $\beta = F(\alpha) \oplus K$. From this relation, it becomes easy to extract the key value for known values of α and β, as the key can be calculated directly: $K = F(\alpha) \oplus \beta$. A cipher that is designed only by this principle is clearly vulnerable to known plaintext/cipher attacks, but it would still be difficult to extract the key if we only knew the plaintext or the ciphertext, since we would then have two unknowns in the equation. So if we introduce a second key in the cipher so that $\beta = F(\alpha) \oplus K_1 \oplus K_2$ where K_1 and K_2 are different, it becomes much harder to extract the key values from known values of α and β. We know that in an equation with two unknowns that are independent of each other, we need at least two equations to find the right value for the two. If not, there would be as many unique solutions to the equation as the whole key space. In AES, an XOR operation between the plaintext and the key is added to the cipher as prewhitening before the round functions starts, thus a more accurate expression of the first round is $AK \circ MC \circ SR \circ SB \circ AK$. Because we have two unknown keys here, we can apply the same logic and say that we need at least two pairs of known plaintext and ciphertext in order to find the right key combination. While this is not entirely accurate since the value of the two keys depends on each other for the key schedule and we can brute-force the solution, it is at least much easier to find the key with a pair of plaintexts and ciphertext that we can observe the differences between.

The attack method is now as follows: generate two random plaintexts (p^0, p^1) and request the encryption of them (c^0, c^1) through the first AES round with an unknown key so that

$$c^0 = enc_1(p^0, K) = rk^1 \oplus MC \circ SR \circ SB(p^0 \oplus rk^0)))$$
$$c^1 = enc_1(p^1, K) = rk^1 \oplus MC \circ SR \circ SB(p^1 \oplus rk^0))).$$

When we solve the equation set with respect to the first round key we get the following:

$$rk^0 = p^0 \oplus SB^{-1} \circ SR^{-1} \circ MC^{-1}(c^0 \oplus rk^1)))$$
$$rk^0 = p^1 \oplus SB^{-1} \circ SR^{-1} \circ MC^{-1}(c^1 \oplus rk^1)))$$

and when the right round key rk^1 is plotted in, the relation

$$p^0 \oplus SB^{-1} \circ SR^{-1} \circ MC^{-1}(c^0 \oplus rk^1))) = p^1 \oplus SB^{-1} \circ SR^{-1} \circ MC^{-1}(c^1 \oplus rk^1)))$$

must hold. Let the left hand side of the equation be α and the right hand side be β. If we now plug in a candidate key *cand* for rk^1 which is only correct in one column, it then follows directly from the equation that $SR(\alpha \oplus \beta)$ must be zero in the exact same column. However, for a set of 2^{32} possible values, there might exist many false positives that fulfills this requirement. The number of false positives in the column space is ranging from 2^4 to 2^{11} dependent on the values in the other columns in the round key. Thus, we have to store the value in a list when we get a hit. We can now begin to attack each column individually by iterating through all possible values for the candidate key and storing all the hits in separate lists for the respective columns. This process has a computational complexity of $4 \cdot 2^{32} = 2^{34}$. After this process, we combine the hits in the list to form a whole candidate key for rk^1 for which we use the reverse key schedule algorithm on and test an encryption with one of the plaintext to see if it match its corresponding ciphertext. The computational complexity of this combination process is at least 2^{16} considering the lowest amount of 16 hits for each column which makes the total complexity of the attack 2^{34}. The attack also works for attacking rk^0 directly instead of rk^1 so we don't need to perform the reverse key schedule for each key guess where the relation we then need to balance is

$$c^0 \oplus MC \circ SR \circ SB(p^0 \oplus rk^0) = c^1 \oplus MC \circ SR \circ SB(p^1 \oplus rk^0)$$

However, since the reverse key schedule only need to be performed 2^{16} times in the second phase of the attack, this does not affect the total computational complexity of 2^{34} for the attack which is dominated by guessing the right sub-keys in every column. Although this complexity is a bit high for a key recovery in just one round, the data complexity outweighs this by only $2\ CP/CC$. The algorithm for this attack is described in **Algorithm 2**.

Algorithm 2 Key recovery attack on One AES round

Input: A pair of plaintexts (p^0, p^1) and its corresponding ciphertexts (c^0, c^1) after 1 round
Output: The secret key k

```
1: Make an empty set $\mathbb{S} = \{\mathbb{S}_0, \mathbb{S}_1, \mathbb{S}_2, \mathbb{S}_3\}$
2: for $i$ from 0 to 3 do
3:     for $k_0$ from 0 to $2^8 - 1$ do
4:         for $k_1$ from 0 to $2^8 - 1$ do
5:             for $k_2$ from 0 to $2^8 - 1$ do
6:                 for $k_3$ from 0 to $2^8 - 1$ do
7:                     $cand_i = (k_0, k_1, k_2, k_3)$
8:                     $\alpha \leftarrow p^0 \oplus SB^{-1} \circ SR{-}1 \circ MC{-}1(c^0 \oplus cand_i)$
9:                     $\beta \leftarrow p^0 \oplus SB^{-1} \circ SR{-}1 \circ MC{-}1(c^1 \oplus cand_i)$
10:                    $x \leftarrow SR(\alpha \oplus \beta)$
11:                    if $x_i = (0,0,0,0)$ then
12:                        $\mathbb{S}_i \leftarrow cand_i$
13:                    end if
14:                end for
15:            end for
16:        end for
17:    end for
18:    for all combinations of hits in $\mathbb{S}_0, \mathbb{S}_1, \mathbb{S}_2, \mathbb{S}_3$ do
19:        $candRK_0 \leftarrow \mathbb{S}_0$, $candRK_1 \leftarrow \mathbb{S}_1$, $candRK_2 \leftarrow \mathbb{S}_2$, $candRK_3 \leftarrow \mathbb{S}_3$
20:        $k \leftarrow reverseKeySchedule(candRK)$
21:        if $enc_1(p^0, k) = c^0$ then
22:            Return $k$
23:        end if
24:    end for
25: end for
```

6.2 Exchange attack for 1-3 rounds

Here we will cover a key recovery attack that works in the adaptively chosen plaintext / ciphertext setting and have the same data complexity and computational complexity for 1, 2 and 3 rounds using the fact that a diagonal space maps to a column space and a column space to a mixed space. Let's start by analyzing some simple facts. We choose two plaintext p^0 and p^1 that are only different in the first byte

$$p^0 = \begin{array}{|c|c|c|c|}\hline * & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \\ \hline\end{array} \quad p^1 = \begin{array}{|c|c|c|c|}\hline * & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \\ \hline\end{array}$$

and encrypt them 1 round using a random key to form c^0 and c^1. We observe that the difference between c^0 and c^1 only lies in the first column space after the encryption of

1 round. If we then do a column exchange between c_0^0 and c_0^1 to form

$$c'^0 = \rho_c^v(p^0, p^1), c'^1 = \rho_c^v(p^1, p^0)$$

where $v = (1,0,0,0)$ and decrypt c'^0 and c'^1 with the same key to form p'^0 and p'^1, we will actually end up with the same pair of plaintext as we started with where $p^0 = p'^1$ and $p^1 = p'^1 0$. This is because the column exchange does not create any unique pairs when we swap the only column that differ between two state and it simply makes $c'^0 = c^1$ and $c'^1 = c^0$. So if we rather do a diagonal exchange between the ciphertexts

$$c'^0 = \rho_d^v(p^0, p^1), c'^1 = \rho_d^v(p^1, p^0),$$

we get a unique ciphertext pair where $(c'^0, c'^1) \neq (c^0, c^1)$, but the zero difference remains the same: $\nu(c'^0 \oplus c'^1) = \nu(c^0 \oplus c^1)$. We observe that the difference in $c^0 \oplus c^1 = c'^0 \oplus c'^1$ is the same whether we do a column exchange or a diagonal exchange

$$\rho_c^v(c^0, c^1) \oplus \rho_c^v(c^1, c^0) = \rho_d^v(c^0, c^1) \oplus \rho_d^v(c^1, c^0).$$

This is due to the fact that a diagonal exchange between two states that differs only in the first column only affects the first byte between them which makes $c'^0_{0,0} = c^1_{0,0}$ and $c'^1_{0,0} = c^0_{0,0}$ and the rest equal. If we now decrypt the ciphertext pair to p'^0 and p'^1, we end up with a plaintext pair where both of the plaintexts are active just in the first diagonal and will only differ in the first byte, the same byte position that p^0 and p^1 differs. This is true for any key we use and the value of $p'^0 \oplus p'^1$ depends only on the first byte in the key in this scenario. This means that for a given pair (p^0, p^1) that differs only in the first byte and its corresponding plaintext pair (p'^0, p'^1) after encryption of one round, diagonal exchange and decryption, we only need 2^8 attempts to guess the first byte of the correct key. We can then initialize a candidate key *cand* that is zero in all bytes and do the following operation

$$\alpha = dec^1_{cand}(\rho_d^v(enc^1_{cand}(p^0), enc_1(p^1)))$$
$$\beta = dec^1_{cand}(\rho_d^v(enc^1_{cand}(p^1), enc_1(p^0)))$$

for all $i \in \mathbb{F}_{2^8}$ where $cand_{0,0} = i$. Let $\Delta = (\alpha \oplus \beta) \oplus (p'^0 \oplus p'^1)$, then if $\Delta = 0$, we have the correct key guess. In fact, there is a direct linear correlation between each individual byte in $(p^0 \oplus p^1)$, Δ and the correct key value if $(p^0 \oplus p^1)$ only differs in one byte, meaning that if $(p^0 \oplus p^1)$ differs only in byte position x, then Δ has one active byte in the same position x if the key guess is wrong which becomes zero if the key guess for that byte position is correct. In other words, if $(p^0 \oplus p^1)_{i,j} = w$ where w is any random value and zero in the rest and $cand_{i,j} = k_{i,j}$ where k is the original key, then $\Delta_{i,j} = 0$. There are of course some false positives where $\Delta_{i,j}$ is zero even if the candidate key is not correct in $cand_{i,j}$, which can happen 2, 4 or 6 times for all 2^8 values, but the most common scenario is that we have 2 byte values where $\Delta_{i,j} = 0$. We show the relation between $(p'^0 \oplus p'^1)$ and $(\alpha \oplus \beta)$ in Figure 6.1 using the original key to get $(p'^0 \oplus p'^1)$ and the candidate key to get $(\alpha \oplus \beta)$ when we have guessed the correct value of $cand_{i,j}$.

To find the whole key, we choose a plaintext p^0 that is zero in all bytes that we will use as our initial p^0 value and request its corresponding ciphertext c^0 after encryption of 1 round with an unknown key. Then we prepare a set $\mathbb{S}$ of $i = 16$ plaintexts (since we have 16 bytes in the block state) where plaintext i is only active in the i-th byte and zero in the rest. We use plaintext i in $\mathbb{S}$ as our temporary p^1 value. Then for all p^1 in $\mathbb{S}$ we request their corresponding plaintexts (p'^0, p'^1) using our initial p^0 value. We now have 16 $(p'^0 \oplus p'^1)$ values that has an active byte only in the same position i as our plaintesxt in $\mathbb{S}$. We can then brute-force each individual byte of the key by letting our candidate key iterate through each byte value in the i-th position and compare $(p'^0, p'^1)_i$ to $(\alpha \oplus \beta)_i$ which will be our Δ_i value. When Δ_i becomes zero, we store the value of the candidate key in a separate list for that byte position. We need 16 lists in total to hold all the possible candidate key values and the computational complexity of this process is only $16 \cdot 2^8 = 2^{12}$. When we have iterated through all the bytes in the candidate key, we end up with 16 list that holds 2, 4 or 6 values where one of them is the correct key value. We then simply combine the hits in the lists to make a whole key guess and use it on our initial p^0 value to see if it encrypts to its corresponding c^0 value. The computational complexity of this combination process is the amount of hits in each list multiplied with each other, which makes the lowest theoretical computational complexity at 2^{16} if we only have 2 hits in each list and the highest theoretical computational complexity at $2^{41.36}$ if we have 6 hits in each list, but since it is a much higher probability that each list holds only 2 values, the typical complexity of this is around 2^{20} to 2^{24}. We emphasize that this depends on the value of the only active byte we choose in the 16 plaintexts in $\mathbb{S}$ and the computational complexity of the whole attack will be dominated by this. This key recovery attack on 1 round only needs 1 initial chosen plaintext and its

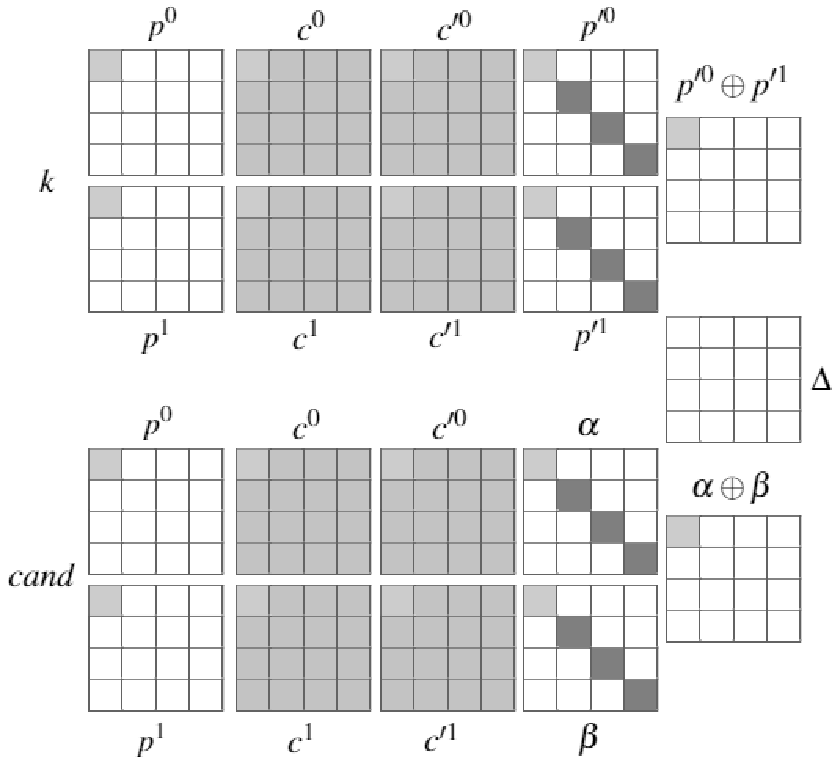


Figure 6.1: The Δ relation using the original key and a candidate key that are correct in 1 byte

corresponding ciphertext + 16 chosen chosen ciphertexts from our 16 chosen plaintexts in the set $\mathbb{S}$ + their 16 corresponding adaptively chosen plaintext when they are diagonal exchanged with c^0 which make the total data complexity of 1 CP/CC + 32 ACP/ACC, but since we only need information from the plaintexts in the attack, we actually only need 1 CP/CC + 16 ACP.

We will now demonstrate how to perform this attack with only 2 ACP/ACC. We start by choosing two random plaintexts (p^0, p^1) that can be any random value and requests their corresponding ciphertexts (c^0, c^1) after 1 round with an unknown key k, perform a diagonal exchange on (c^0, c^1) to make (c'^0, c'^1) and ask for the decryption of (c'^0, c'^1) with the same key to get (p'^0, p'^1). We can now do the same procedure using a candidate key instead of k to end up with two plaintexts α and β. Since (p^0, p^1) consist of random values where all the bytes are active, the linear correlation between each individual byte in $(p'^0 \oplus p'^1)$, Δ and k has now disappeared, but instead of being a direct correlation between each individual byte, there is now a correlation between the diagonal spaces in $(p'^0 \oplus p'^1)$, Δ and k, meaning that if we only change the bytes in diagonal space $\mathbb{D}_i$ of the candidate key, that change will only affect the same diagonal space in Δ. It then follows that if we do a correct key guess for a whole diagonal space $\mathbb{D}_i$ in the candidate key, then $\mathbb{D}_i$ will be zero in Δ as well. We can therefore brute-force each diagonal space $\mathbb{D}_i$ in the candidate key separately to check if the corresponding diagonal space in Δ is zero, making the computational complexity of this attack equal to the attack on one round in section 6.1.1.

This attack method can easily be extended to cover two and three rounds by using column exchange and mixed exchange respectively instead of a diagonal exchange. This is due to the fact that a diagonal exchange after one rounds maps to a column exchange after two rounds, which maps to a mixed exchange after three rounds. Using this relation, we can make the same plaintext pair (p'^0, p'^0) from an initial plaintext pair (p^0, p^0) by performing different exchange functions in different rounds. For a plaintext pair (p^0, p^0), we have the following relation:

$$(p'^0, p'^1)_1 = (R^{-1}(\rho_d^v(R^1(p^0), R^1(p^1))), R^{-1}(\rho_d^v(R^1(p^1), R^1(p^0)))$$
$$(p'^0, p'^1)_2 = (R^{-2}(\rho_c^v(R^2(p^0), R^2(p^1))), R^{-2}(\rho_c^v(R^2(p^1), R^2(p^0)))$$
$$(p'^0, p'^1)_3 = (R^{-3}(\rho_m^v(R^3(p^0), R^3(p^1))), R^{-3}(\rho_m^v(R^3(p^1), R^3(p^0)))$$

where $(p'^0, p'^1) = (p'^0, p'^1)_1 = (p'^0, p'^1)_2 = (p'^0, p'^1)_3$. We can then do the same attack method on two and three rounds of AES that have the exact same data and computational complexity as for one round. The attack method applied on three rounds is described in **Algorithm 3**.

There seems to be a pattern where for a given plaintext pair (p^0, p^1), we can produce the exact same pair (p'^0, p'^1) by encrypting (p^0, p^1) n number of rounds to form (c^0, c^1), do an exchange function F to form (c'^0, c'^1) and then decrypt (c'^0, c'^1), but for rounds higher than 3, there seems to be no independent exchange function that will leave us with the exact same (p'^0, p'^1) pair. This is because a column exchange and a diagonal exchange operates directly on the words between two states and the mixed exchange decrypts two states through $SR^{-1} \circ MC^{-1}$ and then performs a column exchange directly between the two intermediate states. In this process, the mixed exchange actually skips the AddRoundKey function before MC^{-1} since it has no effect on the difference between the two states. We end up with the same (p'^0, p'^1) pair when we perform an exchange function on two states after n rounds that exchange values in the n-th degree subspace and after three rounds, these subspaces becomes key dependent. This means that to make the same (p'^0, p'^1) pair from an initial pair (p^0, p^1) after encryption of $n > 3$ rounds, the exchange function also needs to be key dependent. For example, to do an exchange function on two states (α, β) after four rounds of encryption where we end up with the same (p'^0, p'^1) pair, we can do the following exchange method

$$\rho_4^v(\alpha,\beta) = \rho_m^v(SB^{-1} \circ SR^{-1} \circ MC^{-1}(\alpha \oplus rk^4), SB^{-1} \circ SR^{-1} \circ MC^{-1}(\beta \oplus rk^4))$$

where rk^4 is the fourth round key. Similarly, we can apply this method to infinitely rounds so that

$$\rho_n^v(\alpha,\beta) = \rho_m^v(R^{-(n-3)}(\alpha), R^{-(n-3)}(\beta))$$

where we need $n-3$ round keys to make the same (p'^0, p'^1) pair and when $n = 10$, we remove the first MC^{-1} function in the exchange function. So in order to extend this attack to more than three rounds, it requires that we know some of the correct round keys, which we do not have. But if we theoretically could request the encryption of any pair (p^0, p^1) through n rounds and ask for the decryption of the key dependent exchange function between the two ciphertexts to get (p'^0, p'^1), we could break any number of rounds only by this information. In this way, this method becomes a only chosen plaitext attack where we choose a pair (p^0, p^1) and ask for its corresponding pair (p'^0, p'^1), but since this method does not appeal to any real-world scenarios, it is merely a theoretical attack and cannot be considered a real key recovery attack since it depends on known round keys.

Also an interesting property to note is by encrypting a pair (p^0, p^1) n rounds, perform a n degree exchange function between the corresponding ciphertexts and decrypt them with the same key to get (p'^0, p'^1), if we then repeat the same process with (p^0, p'^0), we end up with a loop where $(p^0, p'^0) = (p^1, p'^1)$. This means that every plaintext p^0 have a corresponding plaintext p'^0 for a given key that ends up being the same pair after the encryption, swapping and decryption process. The reason for this is because the two plaintexts have a direct collision of 2^{96} bits after encryption of 1 round and we only exchange the 32-bit word that differs. The knowledge of such a plaintext pair for a given key is enough to break the key with a complexity of $4 \cdot 2^{32}$, and given 16 such pairs, the complexity becomes only $16 \cdot 2^8$, but to do this for more than 3 rounds, the exchange equivalent swapping functions becomes dependent on round-keys which we do not have access to, and thus the attack only becomes theoretical. We attach an executable GUI program in this submission that can generate such pairs by performing an n-degree key-dependent exchange function on their corresponding ciphertexts.

Algorithm 3 3 round key recovery attack

Input: 1 CP/CC pair, 16 plaintexts with zero difference in 1 byte and their corresponding encryption/decryption when mixed exchanged with the ciphertext from the initial plaintext
Output: Secret key

```
$\mathbb{S} \leftarrow$16 lists
1: for $i$ from 0 to 15 do
2:     $p^0 \leftarrow$randomState(), $p^1 \leftarrow p^0$, $p^1_i$ = randomByte()
3:     $c^0 \leftarrow enc_k(p^0,3)$, $c^1 \leftarrow enc_k(p^1,3)$
4:     $c'^0 \leftarrow \rho^v_m(c^0,c^1)$, $c'^1 \leftarrow \rho^v_m(c^1,c^0)$,
5:     $p^0 \leftarrow dec_k(c'^0,3)$, $p^1 \leftarrow enc_k(c'^1,3)$
6:     $x \leftarrow (p^0 \oplus p^1)$
7:     $cand$ = allZeroes()
8:     for $j$ from 0 to 255 do
9:         $cand_i \leftarrow j$
10:        $c^0 \leftarrow enc_{cand}(p^0,3)$, $c^1 \leftarrow enc_{cand}(p^1,3)$
11:        $c'^0 \leftarrow \rho^v_m(c^0,c^1)$, $c'^1 \leftarrow \rho^v_m(c^1,c^0)$,
12:        $p^0 \leftarrow dec_{cand}(c'^0,3)$, $p^1 \leftarrow enc_{cand}(c'^1,3)$
13:        $y \leftarrow (p^0 \oplus p^1)$
14:        if $(x_i = y_i)$ then
15:            $\mathbb{S}_i \leftarrow j$
16:        end if
17:    end for
18: end for
19: for all combinations in $\mathbb{S}$ do
20:    $recoveredKey \leftarrow \mathbb{S}_{i,j}$
21:    $testC \leftarrow enc_{recoveredKey}(p^0,3)$
22:    if $(c^0 = testC)$ then
23:        return $recoveredKey$
24:    end if
25: end for
```

6.3 Distinguishing attack for 5 rounds

In this section we will review a distinguisher for 5 rounds that was presented in (Yoyo Tricks with AES) [23] with a record breaking complexity of approximately $2^{25,8}$ adaptive chosen ciphertexts. The visual representation of the thought process behind distinguisher is depicted in Figure 6.2. The distinguisher works by exploiting a fixed property

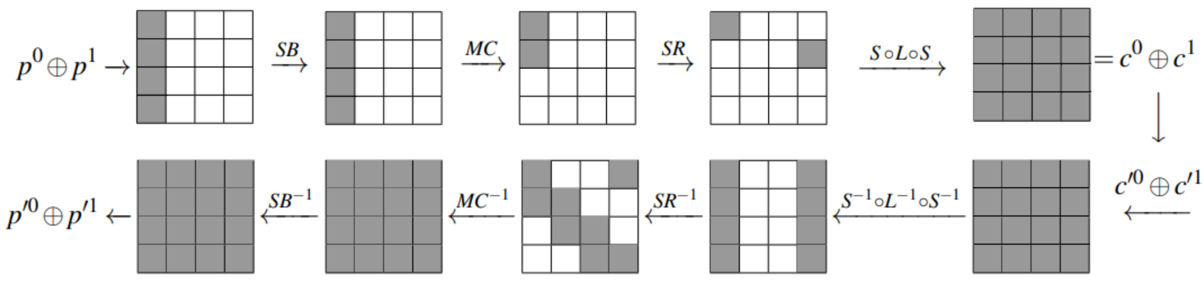


Figure 6.2: 5 rounds distinguishing attack pattern

of the MixColumn function. The main idea here is that if the difference between two plaintexts after the first round is zero in t out of 4 words, we apply the exvhange function and return a new plaintext pairs that are zero in exactly the same word after one round.

We begin by generating a plaintext pair (p^0, p^1) that has a zero difference pattern with Hamming weight $\omega t(p^0 \oplus p^1) = 3$ where only the first column is active. Then we apply the SubBytes transformation on p^0 and p^1 and see that the Hamming weight of the zero difference pattern is preserved, that is $\omega t(\nu(p^0 \oplus p^1)) = \omega t(\nu(SB(p^0) \oplus SB(p^1))) = 3$. This also applies to the MixColumn function since it operates on the columns independently and thus $\omega t(\nu(p^0 \oplus p^1)) = \omega t(\nu(MC \circ SB(p^0) \oplus MC \circ SB(p^1))) = 3$. If we now have that the two states are different in all bytes in the first column after the MixColumn layer, applying the ShiftRows transformation on them will destroy the Hemming weight of their zero difference pattern, making all the columns active. If we rather imagine that the two states are different in only 2 out of 4 bytes after MixColumns, then after applying the ShiftRows transformation they will have a Hemming weight of 2 where only 1 byte is active in the fist column and 1 byte is active in the last column while the two columns in the middle is the zero set. This is the scenario we want after $Q' = SR \circ MC \circ SB$ for the best performance of the attack. The probability that a pair (p^0, p^1) with $\omega t(\nu(p^0 \oplus p^1)) = 3$ encrypts trough Q' to a pair of states with $\omega t(\nu(Q'(p^0) \oplus Q'(p^1))) = t$ can be approximated by

$$P_b(t) = \binom{4}{t} \cdot q^{-t}$$

where $q = 2^8$. For $t = 3$, this probability is about $2 \cdot 10^{-7}$ (or roughly 2^{-22}).

So if a pair of states (p^0, p^1) encrypts one round trough Q', forming a pair of states $(Q'(a), Q'(b))$, that has a zero difference pattern of weight t (only $4 - t$ out of 4 words are active), then the probability that any $4 - t$ bytes are simultaneously zero in a word in the difference $a \oplus b$ is q^{t-4}. When this happens, all bytes in the difference are zero.

We now define the key dependent subspace trail $T_{z,a} = Q(\alpha \oplus x) \mid x \in V_z$ and let H_i denote the image of the i-th word in $SR(\alpha \oplus x) \mid x \in V_z$ such that $|H_i| = q^{4-t}$. It then follows that $T_i^{z,\alpha} = SB \circ MC(H_i)$ such that $T_{z,\alpha}$ has $T_i^{z,a}$ as its i-th word. Then all the pairs where $\omega t(\nu(Q'(p^0) \oplus Q'(p^1))) = 3$ belongs to the same set $T_{z,a} = T_0^{z,a} \times T_1^{z,a} \times T_2^{z,a} \times T_3^{z,a}$ where each set $T_i^{z,a} \subset \mathbb{F}_q^4$ has size $q = q^{4-\omega t(z)}$. This set will be different for each secret key that's used in the encryption since it depends on the unknown round keys XOR-ed between the layers. Although we do not know the set $T_{z,\alpha}$ because of this, we know their size and we should expect one of them to be among 2^{-22} random pairs. If we rather let $t = 2$, then the probability increases to around $9 \cdot 10^{-5}$, reducing the run time of the distinguisher from 2^{-22} to $2^{-13.4}$.

So if we generate a set of around 2^{13} random pairs, we will expect that at least one of the pairs will have a zero difference pattern of weight 2 after Q'. Lets imagine that we have found such a pair and continue to encrypt the states 4 rounds trough $S \circ L \circ S$ to form c^0 and c^1. The zero difference pattern is now destroyed, but it is here the exchange function comes in handy. We make an exchange pair $(\rho^{\nu}(c^0, c^1), \rho^{\nu}(c^1, c^0)) = (c'^0, c'^1)$ and have that $\nu(c^0, c^1) = \nu(c'^0, c'^1)$. If we now decrypt c'^0 and c'^1 4 rounds trough $S^{-1} \circ L^{-1} \circ S^{-1}$ to get $Q'(p'^0)$ and $Q'(p'^0)$ we see that $\nu(Q'(p^0) \oplus Q'(p^1)) = \nu(Q'(p'^0) \oplus Q'(p'^1))$ because the zero difference pattern between an original pair and an exchange pair is preserved trough $S^{-1} \circ L^{-1} \circ S^{-1}$. Here $Q'(p'^0) \oplus Q'(p'^1)$ is active in all bytes in the first and last column, while the two columns in the middle is the zero set. We keep in mind that this only happens if we assume that we have found a pair (p^0, p^1) with a zero difference pattern of weight 2 in their difference after Q'. For two random states α and β, the probability that $4-t$ bytes are zero in the difference $\alpha_i \oplus \beta_i$ is $P_b(4-t) = q^{t-4}$. The probability that $4-t$ bytes are zero in the difference $\alpha \oplus \beta$ in any of the 4 words is

$$4P_b(4-t) = 4 \cdot \binom{4}{t} \cdot q^{t-4}$$

thus, the probability when $t = 2$ is around $2^{-11,4}$.

Now we need to decrypt the exchange pair trough $Q'^{-1} = SB^{-1} \circ MC^{-1} \circ SR^{-1}$. When we first go trough SR^{-1}, the difference between the two states looks like a stair shape where each column has 2 active bytes and 2 zero bytes. When we decrypt further through MC^{-1} we will then have a predictable pattern that follows from the properties of the MixColumn matrix M where $x \cdot M$ can not contain t or more zeros if x has $4-t$ zeros. For AES, we can not have more than two zeroes in each word after MC^{-1} if the state before MC^{-1} has exactly two bytes in each word, but if it was a random cipher, we would have that one of the 4 columns have 2 or more zeroes if we try 2^{11} pars. This is how we distinguish weather this is AES or a random permutation.

A brief explanation of the algorithm
We make 2^{13} pairs (p^0, p^1), then test each of them by making 2^{11} exchange pairs and check if they have 2 or more zeroes in $p'^0 \oplus p'^1$. If they do, we go to the next pair (p^0, p^1) and repeat the process of checking if its 2^{11} exchange pairs are 2 or more in $p'^0 \oplus p'^1$. If some of the 2^{13} survives the test of not having 2 or more zeroes in its 2^{11} exchange pairs, we can conclude that this is AES and terminate the process. Every time we get 2 zeroes, we discard the pair and check next pair. Continue this for 2^{13} pairs. If this is AES, we have at least one pair that will survive the test.

When performing this attack on the simplified representation of 5 rounds where $R^5 = S \circ L \circ S \circ Q'$, we use the column exchanges on plaintext pairs and ciphertext pairs, but in order for this attack to work on the full 5 round, we perform a mixed exchange between ciphertexts and the diagonal exchange between plaintexts.

6.4 Key recovery for 5 rounds

In this section we will cover a key recovery attack on 5 rounds AES developed presented in [23]. This attack works in the adaptive chosen plaintext/ciphertext (ACP/ACC) scenario where we are allowed to generate new plaintexs and ciphertexts to encrypt/decrypt with the real key. The mechanics behind this attack is that the attacker queries plaintext pairs to encrypt with the real key and uses a swap operation on the obtained ciphertext pairs to generate new cipertext pairs to decrypt. The attacker can then observe differences in the new pairs and continue to generate new pairs with the same zero difference until one of them meets a certain zero difference. When this condition is met, the attacker takes advantage of impossible zero differentials made by the Mix-Column layer to attack individual sub-keys by generating yoyo pairs whose difference after a partial encryption is guaranteed to be non-zero only in the first word.

The attack had a record breaking complexity both in terms of data usage and computation where the amount of data needed is roughly $2^{11.3}$ adaptively chosen plaintexts with a computational complexity of 2^{31}. Up until now, this has been the fastest key recovery attack on 5 round AES. There have recently been studies where others have taken the techniques from this attack and implemented it on other AES-based SPN network like for example AESQ, but as far as we know, no one have improved the attack on pure AES.

A major part of this thesis is to study and implement the attack to see whether it is room for improvement since the attack method is quite new. I have implemented the attack in C# and experimented with it where I came up with some new and interesting ideas for different attack methods by combining the exchange functions with techniques used in the square attack which led to the attack covered in 6.2. Although no improvements have been made on the theoretical computational complexity of this attack, I have made some small adjustments in the implementation of the attack which improved the overall data complexity from $2^{11.3}$ ACC to roughly $2^{10.59}$ ACC in one setting and to 2^{10} ACC in another setting by requesting 3 or 2 exchange pairs per sub-key guess instead of 5. Running this attack on a laptop, the time complexity varies between the different settings depending on secret key, but after many tests it turns out that when we choose 3 pairs instead of 5 it runs faster on average.

We now take a deeper dive into the theory of this attack. We will use the same representation of 5 rounds like in the exchange distinguisher for 5 rounds where we shift out the SR-layer with $MC \circ SB$ so that $R^5 = S \circ L \circ S \circ Q'$ and $Q' = SR \circ MC \circ SB$. The goal of the attack is to find the first round-key before Q', which also is the secret key k. The attack starts by picking a pair of plaintexts p^0 and p^1 with a certain zero difference, where the first words are given by $p^0_0 = (0, i, 0, 0)$ and $p^1_0 = (z, z \oplus i, 0, 0)$ where z is a random non-zero element of $\mathbb{F}_q$ and the tree other words are equal as depicted here:

$$p^0 = \begin{array}{|c|c|c|c|} \hline 00 & a & e & i \\ \hline i & b & f & j \\ \hline 00 & c & g & k \\ \hline 00 & d & h & l \\ \hline \end{array} \qquad p^1 = \begin{array}{|c|c|c|c|} \hline z & a & e & i \\ \hline z \oplus i & b & f & j \\ \hline 00 & c & g & k \\ \hline 00 & d & h & l \\ \hline \end{array}$$

The property for this setup is that we always have the same difference in $p^0_0 \oplus p^1_0 = (z,z,0,0)$ for all values of $i \in \mathbb{F}_q$ while the difference in the three other words is always zero. This also means that $AK(p^0) \oplus AK(p^1) = (z,z,0,0)$ will always hold. We now let $k_0 = (k_{0,0}, k_{0,1}, k_{0,2}, k_{0,3})$ be the first 4 bytes in round-key 0 XORed with the first word of the plaintexts so that

$$\begin{aligned} k_{0,0} &= rk^0_{0,0} \oplus p^0_{0,0} \oplus p^1_{0,0} \\ k_{0,1} &= rk^0_{1,0} \oplus p^0_{1,0} \oplus p^1_{1,0} \\ k_{0,2} &= rk^0_{2,0} \oplus p^0_{2,0} \oplus p^1_{2,0} \\ k_{0,3} &= rk^0_{3,0} \oplus p^0_{3,0} \oplus p^1_{3,0} \end{aligned}$$

where rk^0 is the first round key. Since $p^0_0 \oplus p^1_0 = (z,z,0,0)$ for all $i \in \mathbb{F}_q$, the last two bytes in k_0 will always be equal to the last two bytes in rk^0_0. We also have that the difference between the two first bytes in k_0 will always be equal to the difference between the two first bytes in rk^0_0, i.e. $k_{0,0} \oplus k_{0,0} = rk^0_{0,0} \oplus rk^0_{1,0}$. Since we know $z = p^0_{0,0} \oplus p^1_{0,0} = p^0_{1,0} \oplus p^1_{1,0}$ is the same for all values of $i \in \mathbb{F}_q$ we have that $k_0 = (z \oplus rk^0_{0,0}, z \oplus rk^0_{1,0}, rk^0_{2,0}, rk^0_{3,0})$ for all $i \in \mathbb{F}_q$. Although we don't know the 4 bytes in rk^0_0, we can use k_0 as a candidate key to find them based on this relation. Lets look at the first word in the partial encryption of p^0 and p^1 trough $SB \circ AK$:

$$\begin{aligned} SB \circ AK(p^0_0) &= (s(p^0_{0,0} \oplus k_{0,0}), s(p^0_{1,0} \oplus k_{0,1}), s(p^0_{2,0} \oplus k_{0,2}), s(p^0_{3,0} \oplus k_{0,2})) \\ SB \circ AK(p^1_0) &= (s(p^1_{0,0} \oplus k_{0,0}), s(p^1_{1,0} \oplus k_{0,1}), s(p^1_{2,0} \oplus k_{0,2}), s(p^1_{3,0} \oplus k_{0,2})) \end{aligned}$$

where $s(x)$ is the representation of one AES s-box. When we set the value of $p^0_0 = (0,i,0,0)$ and $p^1_0 = (z, z \oplus i, 0, 0)$, they get canceled out when they are zero, thus we are left with

$$\begin{aligned} SB \circ AK(p^0_0) &= (s(k_{0,0}), s(k_{0,1} \oplus i), s(k_{0,2}), s(k_{0,2})) \\ SB \circ AK(p^1_0) &= (s(k_{0,0} \oplus z), s(k_{0,1} \oplus z \oplus i), s(k_{0,2}), s(k_{0,2})) \end{aligned}$$

where we see that there is only a difference in the first two bytes. We define $b = (b_0, b_1, b_2, b_3)$ to be the difference in $SB \circ AK(p^0_0) \oplus SB \circ AK(p^1_0)$ so that

$$\begin{aligned} b_0 &= s(k_{0,0}) \oplus s(z \oplus k_{0,0}) \\ b_1 &= s(k_{0,1} \oplus i) \oplus s(k_{0,1} \oplus z \oplus i) \end{aligned}$$

where b_2 and b_3 is always zero and b_0 is the only constant since it is independent of i. We continue the partial encryption and define $y = (y_0, y_1, y_2, y3)$ to be the difference between the first words after the partial encryption of the two plaintexts through MC $\circ$ SB $\circ$ AK so that $(y_0, y_1, y_2, y_3) = M \circ s^4(p_0^0 \oplus rk_0^0) \oplus M \circ s^4(p_0^1 \oplus rk_0^0)$ where M is the MixColumns matrix and s^4 is the concatination of 4 parallel s-boxes. Since the columns in M is defined $M = (\alpha, 1, 1, \alpha \oplus 1), (\alpha \oplus 1, \alpha, 1, 1), (1, \alpha \oplus 1, \alpha, 1), (1, 1, \alpha \oplus 1, \alpha)$, we can calculate y this way:

$$\begin{aligned}
y_0 &= \alpha b_0 \oplus (\alpha \oplus 1) b_1 \\
y_1 &= b_0 \oplus \alpha b_1 \\
y_2 &= b_0 \oplus b_1 \\
y_3 &= (\alpha \oplus 1) b_0 \oplus b_1
\end{aligned}$$

where $\alpha = 2$. We know from the properties of M that the total number of zeroes before and after M cannot be more then 3 and if t bytes in a word x are zero, then $x \cdot M$ or $x \cdot M^{-1}$ cannot contain $4 - t$ or more zeros unless all the bytes in x are zero. If we run through all values of $i \in \mathbb{F}_q$ with $z = 0$, we see that $(y_0, y_1, y_2, y_3) = (0, 0, 0, 0)$ because then $p_0^0 = p_0^1$ for all values of i and thus all the bytes before M in $SB \circ AK(p_0^0) \oplus SB \circ AK(p_0^1)$ are zero, but if we pick another constant $z \neq 0$, we see that y_2 becomes zero for $b_0 = b_1$. This happens at least 2 times for all $i \in \mathbb{F}_q$. If we expand the equations like so:

$$\begin{aligned}
y_0 &= \alpha(s(k_{0,0}) \oplus s(z \oplus k_{0,0})) \oplus (\alpha \oplus 1)(s(k_{0,1} \oplus z \oplus i) \oplus s(k_{0,1} \oplus i)) \\
y_1 &= (s(k_{0,0}) \oplus s(z \oplus k_{0,0})) \oplus \alpha(s(k_{0,1} \oplus z \oplus i) \oplus s(k_{0,1} \oplus i)) \\
y_2 &= (s(k_{0,0}) \oplus s(z \oplus k_{0,0})) \oplus (s(k_{0,1} \oplus z \oplus i) \oplus s(k_{0,1} \oplus i)) \\
y_3 &= (\alpha \oplus 1)(s(k_{0,0}) \oplus s(z \oplus k_{0,0})) \oplus (s(k_{0,1} \oplus z \oplus i) \oplus s(k_{0,1} \oplus i))
\end{aligned}$$

it is easier to see that y_2 becomes zero when $s(k_{0,0}) \oplus s(z \oplus k_{0,0}) = s(k_{0,1} \oplus i) \oplus s(k_{0,1} \oplus z \oplus i)$, which happens when $i \in \{k_{0,0} \oplus k_{0,1}, z \oplus k_{0,0} \oplus k_{0,1}\}$. When this condition is met and $z \neq 0$, we then know that $y = (*, *, 0, *)$ where $*$ is any other random value in $\mathbb{F}_q$. Since we have that $k_0 = (z \oplus rk_{0,0}^0, z \oplus rk_{1,0}^0, rk_{2,0}^0, rk_{3,0}^0)$ where $k_{0,0} \oplus k_{0,1} = rk_{0,0}^0 \oplus rk_{1,0}^0$, we see that the first two bytes in k_0 depends on the value of z we choose and since we know the value of i and z, we can then know the difference in the first two bytes in rk_0^0. So for each time $k_{0,0} \oplus k_{0,1}$ is equal to i or $i \oplus z$, we know that $b_0 = b_1$ and thus $y_2 = 0$. In a scenario where $rk_{0,1}$ is known and we run through all values of i and set $k_{0,0} = i$, there are at least two possible solutions of $k_{0,0}$ that makes $y_2 = 0$ where one of them is the correct one. A simple trick in this attack is to assume that $k_{0,0} = k_{0,1} \oplus i$ when we run trough all values of i. This way we can focus on bruteforcing the other 2^{24} bytes in k_0.

We now know that we have at least 2 plaintext pairs (p^0, p^1) in the set $\mathbb{F}_q$ where $p_0^1 = (0, i, 0, 0)$ and $p_0^1 = (z, z \oplus i, 0, 0)$ that makes $y = (*, *, 0, *)$ and the three other words are zero, hence the Hemming weight of the whole state is 3 with a zero difference pattern of $(0, 1, 1, 1)$. If we now apply the ShiftRows function to make Q' complete, we get a Hemming weight of only 1 with the zero difference pattern of $(0, 0, 1, 0)$. This follows directly from the well-known properties of the ShiftRows function. With a Hemming weight of 1 in the difference of two states $\alpha \oplus \beta$ before the $S \circ L \circ S$ stage, we know from the definition of the yoyo function that if we apply $S \circ L \circ S$ on the two states to get (c^0, c^1), make a yoyo pair $c'^0 = \rho^v(c^0, c^1), c'^1 = \rho^v(c^1, c^0)$ and apply $S^{-1} \circ L^{-1} \circ S^{-1}$ on (c'^0, c'^1) to get (α', β'), the zero difference pattern is preserved throughout this process,

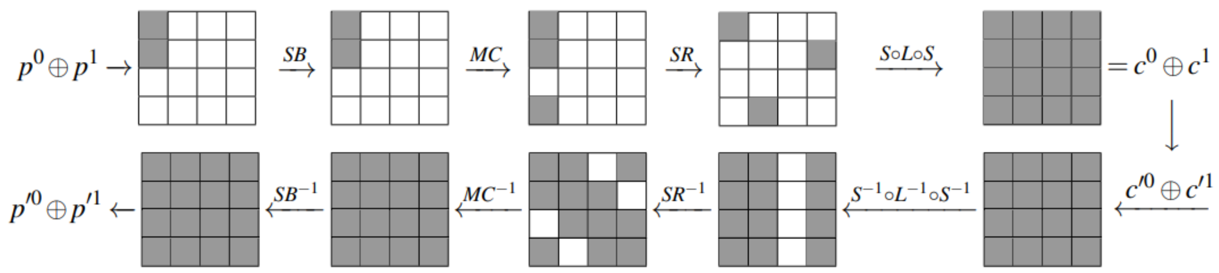


Figure 6.3: 5 rounds key recovery pattern

i.e. $\nu(\alpha,\beta) = \nu(\alpha',\beta')$. This process is depicted in Figure 4.3 where we can see that the third columns in $Q'(p^0 \oplus p^1)$ and $Q'(p'^0 \oplus p'^1)$ are zero and this is one the main properties of AES that we will exploit in this attack.

When we generate a plaintext pair $(p_0^0, p_0^1) = (0,i,0,0),(z,z\oplus i,0,0)$ where the three other words are equal, we know that independent on which secret key that's is used, $Q'(p^0) \oplus Q'(p^1)$ are zero in the third word for at least 2 values of i. If we then use a value for i where this condition is met to encrypt (p^0, p^1) to (c^0, c^1) with any random key, applying the swap function to get (c'^0, c'^1) and returning the corresponding (p'^0, p'^1), we also know that $\nu(Q(p'^0) \oplus Q'(p^1)) = \nu(Q'(p'^0) \oplus Q'(p'^1))$. We can then use (p'^0, p'^1) as our new plaintext pair and repeat this process. In fact, we can go back and forth with this process as many times as we want where all the returned plaintexts will have the same zero difference pattern.

It is easy to check whether $\nu(Q'(p'^0) \oplus Q'(p'^1))$ is zero in the third word by just doing the partial encryption of (p'^0, p'^1) through $SR \circ MC \circ SB$ and then return the sum in the third word in their difference, but because of the ShiftRows layer, we need information from the whole block state throughout this process, even though we only need to check one word. Since $\nu(Q'(p'^0) \oplus Q'(p'^1))$ is only zero in the third word for around 2 values of $i \in \mathbb{F}_q$ in our controlled plaintexts setup and this only occurs when when $y_2 = 0$, we can rather check y_2 directly, skipping the ShiftRows step. This way we only need information in one word throughout the process since MC and SB operates on individual words. Not only do we save $2 \cdot 12$ s-box lookups, $2 \cdot 3$ M-functions and the whole ShiftRows operation for the computational time, but if we look at Q' included with the individual round-keys: $Q' = SR \circ AK(rk^1) \circ MC \circ SB \circ AK(rk^0)$, we also see that we remove the need for a second round-key. This means that we only need information from the first 4 bytes in the first round-key to test our condition.
For our candidate key $k_0 \in \mathbb{F}_q^4 = (k_{0,0}, k_{0,1}, k_{0,2}, k_{0,3},)$ we can now try all 2^{32} possible combinations to calculate $b = (p_0^0 \oplus k_0) \oplus s(p_0^1 \oplus k_0)$ by the following computation:

$$
\begin{aligned}
b_0 &= s(p_{0,0}^0 \oplus k_{0,0}) \oplus s(p_{0,0}^1 \oplus k_{0,0}) \\
b_1 &= s(p_{1,0}^0 \oplus k_{0,1}) \oplus s(p_{1,0}^1 \oplus k_{0,1}) \\
b_2 &= s(p_{2,0}^0 \oplus k_{0,2}) \oplus s(p_{2,0}^1 \oplus k_{0,2}) \\
b_3 &= s(p_{3,0}^0 \oplus k_{0,3}) \oplus s(p_{3,0}^1 \oplus k_{0,3})
\end{aligned}
$$

and then calculate $y_2 = b_0 \oplus b_1 \oplus 2 \cdot b_2 \cdot 3 \cdot b_3$ to see if it is zero or not. The probability that $y_2 = 0$ for any state is 2^{-8}, meaning that in a set of 256 random plaintexts, this will probably occur, but with our controlled setup of plaintext pairs where $p_0^0 = (0, i, 0, 0)$ and $p_0^1 = (z, z \oplus i, 0, 0)$ and the three other words are equal, there is a guarantee that $y_2 = 0$ occurs at least 2 times when $i \in \{k_{0,0} \oplus k_{0,1}, z \oplus k_{0,0} \oplus k_{0,1}\}$. If we run through all values of $i \in \mathbb{F}_q$ with our controlled plaintexts pair setup and set $k_{0,0} = k_{0,1} \oplus i$ and ask for the encryption of every pair with a random key, then making a yoyo pair from the two ciphertexts and returning the corresponding (p'^0, p'^0), we can test the rest of the 2^{24} bytes $k_{0,1}, k_{0,2}, k_{0,3}$ for each i to see if any of the plaintexts pairs returned by the yoyo function have the same value in y_2 when we parital encrypt them with k_0. Because of the probability for having $y_2 = 0$ in any state is 2^{-8}, we will of course in this case get around 2^{16} hits in every value of i and around 2^{24} hits in total, which does not help us to find the real sub-key. If we rather ask for encryption and decryption of 2 pairs per value i, the probability that $y_2 = 0$ in both pairs at the same time becomes $2^{-8 \cdot 2} = 2^{-16}$, which is still not good enough to recover the key because we would then have around 2^{16} hits in total in our test. Increasing the number of plaintext pairs to 3 for each i leaves us with around 2^8 matching sub-keys, but with 4 pairs we will only get around 2 or more matches when $i \in \{k_{0,0} \oplus k_{0,1}, z \oplus k_{0,0} \oplus k_{0,1}\}$ where one of them is the correct sub-key. If we now use 5 pairs instead, the probability that all 5 pairs have $y_2 = 0$ at the same time is $2^{-8 \cdot 5} = 2^{-40}$, which means that when we get a hit in $i \in \{k_{0,0} \oplus k_{0,1}, z \oplus k_{0,0} \oplus k_{0,1}\}$, we can with high probability say that we have found the correct sub-key. A false positive might of course occur for this probability, but in practice we can do some extra tests by generating a few more pairs when the test succeeds on the first 5 pairs to see if it still holds. This will not effect the total data complexity of the attack since this happens rarely. Thus, we need a total of $2 \cdot 2^8 \cdot 5 = 2^{11.3}$ adaptively chosen plaintexts to find the first correct sub-key. The algorithm for finding the first sub-key k_0 is presented here:

Depending on how we implement this attack, we can actually get away with only 2^{10} plaintexts by choosing only 2 pairs per sub-key guess, as we will take a look at in a moment, but since that leaves us with 2^{16} hits for every guess, we will need to do more computation on guessing the remaining 2^{94} sub-keys. So for the sake of clarity, let's imagine that we use a set of 5 pairs which leaves us with a comfortable margin for our test and that we already have found the correct sub-key k_0. Now we need a method for finding the rest of the sub-keys: k_1, k_2, k_3. It turns out that there is a fairly trivial and quick way to do this without just bruteforcing them, which would take an unimaginable amount of time.
When we have found the correct sub-key k_0, we generate a new plaintex pair (p^0, p^1) that differ only in their first byte of the first word and the rest is initialized to only zeroes like this:

$$p^0 = \begin{array}{|c|c|c|c|}\hline * & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \\ \hline\end{array} \qquad p^1 = \begin{array}{|c|c|c|c|}\hline 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \\ \hline\end{array}$$

where we will call the first word in p^0 and p^1 for a'_0 and b'_0 respectively. It does not

Algorithm 4 Recovering the first sub-key k_0 for 5 rounds of AES

Input: Set $\mathcal{P}$ contains 2^8 plaintext pairs (p^0, p^1) where $p^0_0 = (0, i, 0, 0)$ and $p^1_0 = (z, z \oplus i, 0, 0)$ for $i = 0, ..., 2^8 - 1$ where z is any element in $\mathbb{F}_q$ and $p^0_j = p^1_j = 0$ for $j = 1, 2, 3$
Output: The first sub-key k_0

```
1: // r-round AES enc/dec without first $SR$ and last $SR \circ MC$
2: for $i$ from 0 to $2^8 - 1$ do
3:     $p^0 \leftarrow 0, p^1 \leftarrow 0, p^0_0 = (0, i, 0, 0), p^1_0 = (z, z \oplus i, 0, 0)$
4:     $\mathbb{S} \leftarrow \{(p^0, p^1)\}$
5:     while $len(\mathbb{S}) < 5$ do
6:         $c^0 \leftarrow enc_k(p^0, 5), c^1 \leftarrow enc_k(p^1, 5)$
7:         $c'^0 \leftarrow \text{SimpleSWAP}(c^0, c^1), c'^1 \leftarrow \text{SimpleSWAP}(c^1, c^0)$
8:         $p'^0 \leftarrow dec_k(c'^0, 5), p'^1 \leftarrow dec_k(c'^1, 5)$
9:         $p^0 \leftarrow \text{SimpleSWAP}(p'^0, p'^1), p^1 \leftarrow \text{SimpleSWAP}(p'^1, p'^0)$
10:        $\mathbb{S} \leftarrow \mathbb{S} \cup \{(p^0, p^1)\}$
11:    end while
12:    for all $2^{24}$ remaining candidates $k_0$ do
13:        for all $(p^0, p^1) \in \mathbb{S}$ do
14:            if $l_3(s^4(p^0_0 \oplus k_0) \oplus s^4(p^1_0 \oplus k_0)) \neq 0$ then
15:                break and jump to next key
16:            end if
17:        end for
18:    end for
19: end for
```

matter which value we choose for $a'_{0,0}$ and $b'_{0,0}$ as long as they are not the same so that $a'_{0,0} \oplus b'_{0,0} \neq 0$. We now want to partial decrypt p^0 and p^1 through $Q'^{-1} = SB^{-1} \circ MC^{-1} \circ SR^{-1}$ with our candidate key k_0 to form a new pair. If we take a closer look at Q'^{-1} included with the round keys: $Q'^{-1} = AK(rk^0) \circ SB^{-1} \circ MC^{-1} \circ AK(rk^1) \circ SR$, we see that Q'^{-1} depends on two round-keys, rk^1, which we do not know and rk^0, which we only know the first 4 bytes of, but now we are only interested in the difference between the first words after Q'^{-1} so we can actually skip $AK(rk^1) \circ SR^{-1}$ in this case as this step would not effect that difference since p^0 and p^1 only differ in $p^0_{0,0}$ and $p^1_{0,0}$. This leaves us with only operations that operates independently on each word. So we apply the following transformation on a'_0 and b'_0 to form a_0 and b_0:

$$a_0 = s^{-1^4}(a'_0 \oplus k_0) \circ M^{-1}$$
$$b_0 = s^{-1^4}(b'_0 \oplus k_0) \circ M^{-1}$$

where s^{-1^4} is the concatenation of 4 parallel inverse s-boxes and $M^{-1} = B \cdot x^3 + D \cdot x^2 + 9 \cdot x + E$ is the inverse MixColumns matrix. We now have a new pair $p^0 = (a_0, 0, 0, 0), p^1 = (b_0, 0, 0, 0)$ whose difference after $SR \circ MC \circ SB \circ AK$ is guaranteed to be non-zero in only the first word independent on which round-key that is used. This means that if we remove the ShiftRows layer in Q', they will only be different in the first diagonal, which is the property we will take advantage of. We can now use the same technique on p^0 and p^1 to find the remaining sub-keys as we did when we found the first sub-key k_0. Now, this pair in itself is actually useless since they are equal in the three last words, as this would not give us any information about the three remaining sub-keys, but if we request 5 more yoyo pairs from p^0 and p^1 with the real key, we know with a high probability that the new pairs (p'^0, p'^1) will be different in the last three words and also have the same zero difference after $SR \circ MC \circ SB \circ AK$ as our original pair. This means that if we partial encrypt p'^0 and p'^1 through $MC \circ SB \circ AK$ with the real key, then $p'^0 \oplus p'^1$ must be active only in the first diagonal after the encryption, thus we have the following relations:

$$\begin{aligned}
M \circ s^4(p'^0_0 \oplus k_0) \oplus M \circ s^4(p'^1_0 \oplus k_0) &= (w, 0, 0, 0) \\
M \circ s^4(p'^0_1 \oplus k_1) \oplus M \circ s^4(p'^1_1 \oplus k_1) &= (0, w, 0, 0) \\
M \circ s^4(p'^0_2 \oplus k_2) \oplus M \circ s^4(p'^1_2 \oplus k_2) &= (0, 0, w, 0) \\
M \circ s^4(p'^0_3 \oplus k_3) \oplus M \circ s^4(p'^1_3 \oplus k_3) &= (0, 0, 0, w)
\end{aligned}$$

where $k = (k_0, k_1, k_2, k_3)$ is the right secret key and w can be any value in $\mathbb{F}_q$ depending on the plaintext. We see that for each of this relations to hold, the right sub-key must be used. In theory we could now bruteforce k_1, k_2, k_3 individually and check which individual sub-key that fulfills one of this relations. Since we already know k_0, this would have a complexity of $3 \cdot 2^{32}$, which is doable on a laptop in a short time, but we can do even better by recovering all the remaining sub-keys at once with some simple algebra. We let Δ be the difference in $SB \circ AK(p'^0) \oplus SB \circ AK(p'^1)$ so that

$$\begin{aligned}
\Delta_0 &= s^4(p'^0_0 \oplus k_0) \oplus s^4(p'^1_0 \oplus k_0) \\
\Delta_1 &= s^4(p'^0_1 \oplus k_1) \oplus s^4(p'^1_1 \oplus k_1) \\
\Delta_2 &= s^4(p'^0_2 \oplus k_2) \oplus s^4(p'^1_2 \oplus k_2) \\
\Delta_3 &= s^4(p'^0_3 \oplus 3_1) \oplus s^4(p'^1_3 \oplus k_3).
\end{aligned}$$

We know that $MC \circ SB \circ AK(p'^0) \oplus MC \circ SB \circ AK(p'^1)$ is active only in the first diagonal and if we remove the MixColumns step we get Δ. This means that if we apply the inverse MixColumns matrix M^{-1} on our yoyo pairs, we know that the relations:

$$\begin{aligned}
M^{-1} \cdot (w, 0, 0, 0) = w \cdot (E, 9, D, B) &= (\Delta_{0,0}, \Delta_{0,1}, \Delta_{0,2}, \Delta_{0,3}) \\
M^{-1} \cdot (0, w, 0, 0) = w \cdot (B, E, 9, D) &= (\Delta_{1,0}, \Delta_{1,1}, \Delta_{1,2}, \Delta_{1,3}) \\
M^{-1} \cdot (0, 0, w, 0) = w \cdot (D, B, E, 9) &= (\Delta_{2,0}, \Delta_{2,1}, \Delta_{2,2}, \Delta_{2,3}) \\
M^{-1} \cdot (0, 0, 0, w) = w \cdot (9, D, B, E) &= (\Delta_{3,0}, \Delta_{3,1}, \Delta_{3,2}, \Delta_{3,3})
\end{aligned}$$

must hold for known values of M^{-1}. If we take a look at the second equation for example and assume that we know the first byte $k_{1,0}$ in the second sub-key, we can find the second byte in k_1 by balancing the following equation according to $k_{1,1}$:

$$B^{-1} \cdot (s(p^0_{0,1} \oplus k_{1,0}) \oplus (p^1_{0,1} \oplus k_{1,0})) = E^{-1} \cdot (s(p^0_{1,1} \oplus k_{1,1}) \oplus (p^1_{1,1} \oplus k_{1,1}))$$

where B^{-1} and E^{-1} is the AES defined GF(2^8) inversion of B and E. Similar relations also applies to the other bytes in each sub-key, thus if we only know one byte of the sub-key, we can calculate the other one by testing all 256 values of each byte individually

sub-key k_1	sub-key k_2	sub-key k_3
$B^{-1} \cdot \Delta_{1,0} = E^{-1} \cdot \Delta_{1,1}$	$D^{-1} \cdot \Delta_{2,0} = B^{-1} \cdot \Delta_{2,1}$	$9^{-1} \cdot \Delta_{3,0} = D^{-1} \cdot \Delta_{3,1}$
$B^{-1} \cdot \Delta_{1,0} = 9^{-1} \cdot \Delta_{1,2}$	$D^{-1} \cdot \Delta_{2,0} = E^{-1} \cdot \Delta_{2,2}$	$9^{-1} \cdot \Delta_{3,0} = B^{-1} \cdot \Delta_{3,2}$
$B^{-1} \cdot \Delta_{1,0} = D^{-1} \cdot \Delta_{1,3}$	$D^{-1} \cdot \Delta_{2,0} = 9^{-1} \cdot \Delta_{2,3}$	$9^{-1} \cdot \Delta_{3,0} = E^{-1} \cdot \Delta_{3,3}$

Table 6.1: Equations for recovering individual bytes in sub-keys

and see which byte that meets the condition. The equations we calculate for sub-key k_1, k_2 and k_3 is listed in table 4.2.

The attack method is now straight forward. We can now start to attack each individual sub-key by assuming we know the first byte in the sub-key and see if the equations in table 4.2 holds for all the 5 pairs returned by the yoyo function according to the other bytes that we have to guess. If they doesn't hold, we change the first byte and repeat the process until we get a hit. Thus, we have to spend at most $3 \cdot 2^8$ guesses in total to find all the correct first bytes of the sub-keys. A more detailed explanation of this process can be seen in Algorithm 3.

This whole process goes very fast on a computer, so the complexity of this 5-round key recovery attack is dominated by guessing the first sub-key k_0 with a complexity of 2^{31} and the rest is done under a second. The speed of guessing the first sub-key is also dependent on how many adaptively chosen plaintext/ciphertext pairs we choose. The more pairs we use, the more computation we need to do in the first stage for every byte guess, which we guess in the order from 0 to 255 and set $k_{0,0} = k_{0,1} \oplus i$ so depending on the value of the secret key, the speed of the attack will vary. But on the other hand, if we choose less pairs, we will then, of course, have more false positives on the way and go to the next stage for attacking the three remaining sub-keys more often. We can easily check if we have the wrong key by simply comparing an encryption of one of the plaintexts with the real key and the candidate key and see if they have the exact same ciphertext. Since the second stage runs pretty fast, we can afford having no hits here and continue attacking the first sub-key again. By choosing 3 yoyo pairs insted of 5 in the first stage, we will iterate through all the values quicker

Algorithm 5 Recovering full key

```
Input: Secret key $k$, $cand \leftarrow allZeroes$, $cand_0 \leftarrow k_0$
Output: Secret key
$\alpha \leftarrow [b, e, 9, d]$,   $con \leftarrow 0$
$p^0 \leftarrow allZeroes$, $p^1 \leftarrow p^0$, $p^0_{0,0} \leftarrow 1$, $p^0 = MC^{-1} \circ SR^{-1}(p^0 \oplus k)$, $p^1 = MC^{-1} \circ SR^{-1}(p^1 \oplus k)$
Construct 5 yoyo pairs $(p'^0, p'^1)$ from $p^0$ and $p^1$ with $k$ and put them in a set $\mathbb{S}$
 1: for $i$ from 1 to 3 do
 2:     for $k_0$ from 0 to $2^8 - 1$ do
 3:         for $k_1$ from 0 to $2^8 - 1$ do
 4:             while $len(\mathbb{S}) < 5$ do
 5:                 $b_0 \leftarrow s(p'^0_{0,i} \oplus k_0) \oplus s(p'^1_{0,i} \oplus k_0), b_1 \leftarrow s(p'^0_{1,i} \oplus k_1) \oplus s(p'^1_{1,i} \oplus k_1)$
 6:                 if $\alpha[-(i-1) \mod 4]^{-1} \cdot b_0 = \alpha[-(i-1)+1 \mod 4]^{-1} \cdot b_1$ then
 7:                     $con \leftarrow con + 1$
 8:                 end if
 9:             end while
10:             if $con = 5$ then
11:                 $cand_{1,i} \leftarrow k_1$,   $con \leftarrow 0$
12:             end if
13:         end for
14:         for $k_2$ from 0 to $2^8 - 1$ do
15:             while $len(\mathbb{S}) < 5$ do
16:                 $b_0 \leftarrow s(p'^0_{0,i} \oplus k_0) \oplus s(p'^1_{0,i} \oplus k_0), b_2 \leftarrow s(p'^0_{2,i} \oplus k_2) \oplus s(p'^1_{2,i} \oplus k_2)$
17:                 if $\alpha[-(i-1) \mod 4]^{-1} \cdot b_0 = \alpha[-(i-1)+2 \mod 4]^{-1} \cdot b_2$ then
18:                     $con \leftarrow con + 1$
19:                 end if
20:             end while
21:             if $con = 5$ then
22:                 $cand_{2,i} \leftarrow k_2$,   $con \leftarrow 0$
23:             end if
24:         end for
25:         for $k_3$ from 0 to $2^8 - 1$ do
26:             while $len(\mathbb{S}) < 5$ do
27:                 $b_0 \leftarrow s(p'^0_{0,i} \oplus k_0) \oplus s(p'^1_{0,i} \oplus k_0), b_3 \leftarrow s(p'^0_{3,i} \oplus k_3) \oplus s(p'^1_{3,i} \oplus k_3)$
28:                 if $\alpha[-(i-1) \mod 4]^{-1} \cdot b_0 = \alpha[-(i-1)+3 \mod 4]^{-1} \cdot b_3$ then
29:                     $con \leftarrow con + 1$
30:                 end if
31:             end while
32:             if $con = 5$ then
33:                 $cand_{3,i} \leftarrow k_3$,   $con \leftarrow 0$
34:             end if
35:         end for
36:     end for
37: end for
```

Chapter 7

Summary

Here we go through a summary of the main conclusions that we can draw from our cryptanalisis of AES and point to some new suggestions for further research

7.1 Comparisons and conclusions

In this theses we have done an analysis of AES and looked at certain properties in the round functions that makes AES a strong cipher and demonstrated how the full scale version of AES provides a high level of security. We covered in detail how the AES algorithm works in chapter 3 and how to exploit some interesting properties in the algorithm. In chapter 4 we did a comparison review over existing attacks on different rounds and concluded that attacks on over 6 rounds has a complexity that is unpractical given the time they need to finish.

In chapter 6, we have reviewed some cryptanalysis attacks on round reduced AES in the adaptive chosen plaintext/ciphertext setting where the attacks on 1-3 round was self-made based on own cryptanalysis and the two other attacks on 5 rounds was already developed. Although no improvement on the theoretical computational complexity of the 5 round attacks was made, we were able to improve the run time of the implementation of the 5 round key recovery attack by choosing 3 ACP/ACC insted of 5 ACP/ACC, making the iteration process of all k_0 value faster and lowered the overall data complexity from $2^{11.3}$ ACC to roughly $2^{10.59}$ ACC. All the attacks covered in chapter 6 was implemented using *C#*.

In chapter 5, we have showed how the dimensions of subspace trails expands throughout the different encryption rounds using truncated differential cryptanalysis on adaptivery chosen plaintexts pairs, where different exchange functions can be used to make more plaintext/ciphertext pairs that inherits the same properties from an original pair. This method was also used to make the exact same plaintext pair by encrypting the pair n rounds, perform a certain key-dependent exchange function on the ciphertexts and decrypt the new ciphertext pair.

7.2 Further research

We have demonstrated in section 6.2 that any number of rounds on AES can be broken by only two plaintexts (p^0, p^1) that share the common property that when encrypted through n rounds to a pair (c^0, c^1) and a key dependent exchange operation is performed between (c^0, c^1) to make (c'^0, c'^1), the decrypted pair (p'^0, p'^1) becomes equal to (p^0, p^1). When two plaintexts shares this property, doing the key-dependent exchange function on the encrypted pair swaps all the bytes between them so that $(c^0, c^1) = (c'^0, c'^1)$ because of collision of 2^{96} bits after encryption of 1 round. The zero difference between two pairs of plaintexts after applying a column exchange after 3 rounds with different keys is the same as applying a mixed exchange after 4 rounds if the keys that are used is the same in byte position 0, 1, 2, 3, 5, 10, 12, 13, 14 and 15. Although the key-dependent exchange function is not possible for 4 rounds without the knowledge of the fourth round-key, it would be interesting if there exists any method to exploit this by approaching the the value of the forth round key in some way or if any other key-dependent subspace exchange between the ciphertexts could reveal any information of the secret key.

Bibliography

[1] Onur Aci, cmez1, Werner Schindler, and Cÿetin K. Ko c. Cache based remote timing attack on the aes. 2007. 4

[2] C Ashokkumar, Bholanath Roy, M Bhargav Sri Venkatesh, , and Bernard L.Menezes. s-box implementation of aes is not side channel resistant. 2018. 4

[3] Navid Ghaedi Bardeh and Sondre Rønjom. Practical attacks on reduced-round aes. 2019. 4.2

[4] Eli Biham, Alex Biryukov, Orr Dunkelman, and Eran Richardson. Cryptanalysis of skipjack-4xor. pages 5–6, jun 1998. 5.1

[5] Eli Biham and Nathan Keller. Cryptanalysis of reduced variants of rijndael. 2000. 4.1

[6] Alex Biryukov. The boomerang attack on 5 and 6-round reduced aes. 2004. 4.1, 4.2, 5.1

[7] Alex Biryukov and Dmitry Khovratovich. Two new techniques of side-channel cryptanalysis. pages 195–208, 09 2007. 4.1

[8] Andrey Bogdanov, Dmitry Khovratovich, and Christian Rechberger. Biclique cryptanalysis of the full aes. 2011. 4.3.2

[9] Joan Daemen, Lars Knudsen, and Vincent Rijmen. The block cipher square. *Lecture Notes in Computer Science*, 1267, 10 1998. 4.1

[10] Joan Daemen and Vincent Rijmen. *The Design of Rijndael, AES The Advanced Encryption Standard.* Springer-Verlag, 2 edition, November 2001. 3.1

[11] Joan Daemen and Vincent Rijmen. The design of rijndael: Aes - theadvanced encryption standard. 2002. 4.2

[12] Patrick Derbe. Meet-in-the-middle attacks on aes. 2013. 4.2

[13] William Diehl, Abubakr Abdulgadir, Farnoud Farahmand, Jens-Peter Kaps, and Kris Gaj. Comparison of cost of protection against differential power analysis of selected authenticated ciphers. 2018. 4

[14] Jack Dongarra. Report on the sunway taihulight system. 2016. 4.2

[15] Bc. Jana Ernekerova. Analysis and detection of krack attack aganinst wifi infrastructure. 2019. 4

[16] Lorenzo Grassi. Mixture differential cryptanalysis: a newapproach to distinguishers and attacks onround-reduced aes. 2018. 4.1, 4.2

[17] Lorenzo Grassi and Christian Rechberger. Rigorous analysis of truncated differentials for 5-round aes. 2018. 4.1

[18] Lorenzo Grassi, Christian Rechberger, and Sondre Rønjom. Subspace trail cryptanalysis and its applications to aes. 2016. 4.1

[19] Lorenzo Grassi, Christian Rechberger, and Sondre Rønjom. A new structural-differential property of 5-round aes. 2017. 4.1

[20] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, , and Berk Sunar. Wait a minute! a fast, cross-vm attack on aes. 2014. 4

[21] Alan Kaminsky and B. Thomas Golisano. Enigma 2000: An authenticated encryption algorithm for human-to-human communication. 2019. 4.2

[22] Jiqiang Lu1, Orr Dunkelman, Nathan Keller, and Jongsung Ki. New impossible differential attacks on aes. 2007. 4.1

[23] Sondre Rønjom, Navid Ghaedi Bardeh, and Tor Helleseth. Yoyo tricks with aes. jun 2017. 4.1, 6.3, 6.4

[24] Bing Sun, Meicheng Liu, Jian Guo, Longjiang Qu, and Vincent Rijmen. New insights on aes-like spn ciphers. 2016. 4.1

[25] Tyge Tiessen. Improved partial sums"-based square attack on aes. 2012. 4.2

[26] Tyge Tiessen. Polytopic cryptanalysi. 2016. 4.2

[27] Michael Tunstall. Improved partial sums-based square attack on aes. 2012. 4.2

[28] Harshali Zodpe, Prakash Wani, and Rakesh Mehta. Design and implementation of algorithm for des cryptanalysis. pages 278–282, 12 2012. 3