Master Thesis

Analysis of Recent Attacks on AES

David Gstir
david@nod.at

Assessor: Univ.-Prof. Dr. Vincent Rijmen
Advisor: DI Dr. Martin Schläffer

Institute for Applied Information
Processing and Communications (IAIK)
Graz University of Technology
A-8010 Graz, Austria

Graz, 15 October 2012

Abstract

The advanced encryption standard (AES) is one of todays most widely used block ciphers. Although it was introduced in 2001, no attack on the cipher has been found until now that would threaten its practical use. In recent years, a lot of research has been done in the area of single-key attacks. Biclique attacks, for example, are the first attacks on AES that apply to all full variants of AES and are faster than brute force attacks. Furthermore, various improvements to existing round-reduced attacks have been made. One of these improved attacks is the multiset attack which applies to 7-round AES and has a time complexity of 2^{103} encryptions.

In this thesis we present the three most promising of the recently published single-key attacks: the multiset attack, low data complexity attacks and biclique attacks. To find new and improved attacks, we investigate possible combinations of their attack techniques. Since the biclique attacks on AES are only marginally faster than a brute force attack, we further take a closer look at the performance of the biclique attack on AES-128. For this purpose, we created highly optimized software implementations of the biclique attack and a brute force attack using Intel's AES-NI technology. Overall, our implementation of the biclique attack is on average 10.5% faster than the brute force attack. Thus, we were able to verify the claimed advantage of the biclique attack when implemented in software.

Zusammenfassung

Der Advanced Encryption Standard (AES) ist eine der heutzutage am weitesten verbreiteten Blockchiffren. Obwohl der Algorithmus bereits 2001 veröffentlicht wurde, gibt es bis heute noch keine Attacke, die die praktische Anwendung von AES bedenklich machen würde. Vor allem in den letzten Jahren gab es zahlreiche Publikationen im Bereich von "single-key" Attacken auf AES. Ein Beispiel dafür sind Biclique Attacken, die zum erstem Mal eine Möglichkeit bieten, AES mit der vollen Anzahl an Runden, schneller als eine Brute-force Attacke, anzugreifen. Weiters wurden kürzlich auch Verbesserungen von bestehenden Attacken veröffentlicht. Eine davon ist die Multiset Attacke, welche für AES mit sieben Runden eine Zeitkomplexität von 2^{103} Verschlüsselungen aufweist.

In dieser Arbeit werden die drei prominentesten Angriffe von kürzlich veröffentlichten Attacken behandelt: die Multiset Attacke, Attacken mit minimaler Datenkomplexität und Biclique Attacken. Um neue Angriffe zu finden, werden die Möglichkeiten einzelne Techniken dieser neuen Attacken zu kombinieren, analysiert und bewertet. Da Biclique Attacken generell nur geringfügig schneller sind als eine normal Brute-force Attacke, wird weiters die Biclique Attacke auf AES-128 im Detail betrachtet. Dazu werden mit Intel AES-NI Technologie entwickelte, ins Detail optimierte Software-Implementierungen dieser beiden Angriffe vorgestellt. Im Schnitt ist die Implementierung der Biclique Attacke 10,5% schneller als die Brute-force Attacke, was die publizierten Vorteile der Biclique Attacken bestätigt.

Acknowledgments

First of all, I would like to thank my supervisors Vincent Rijmen and Martin Schläffer, who encouraged me to focus my master's thesis in the area of cryptography. Both provided great support in finding the right direction and had at any time an open ear for all my questions and concerns. Especially, I would like to thank Martin for his great comments and endless hours of correcting early drafts of this thesis.

Secondly, special thanks go to my girlfriend Maria and my family for supporting my throughout my studies and providing, encouragement whenever I required it and simply being there when I needed them. Last but not least, I would like to thank Christian Lumper and my cousin Verena Gstir for their advice concerning English grammar and for revising this thesis.

Contents

1. Introduction 1

I. Preliminaries 5

2. The Advanced Encryption Standard 7
- 2.1. The Structure of AES . . . 7
- 2.2. Finite Fields . . . 9
- 2.3. The Round Transformations . . . 10
  - 2.3.1. SubBytes . . . 11
  - 2.3.2. ShiftRows . . . 12
  - 2.3.3. MixColumns . . . 12
  - 2.3.4. AddRoundKey . . . 13
- 2.4. Key Schedule . . . 13
- 2.5. Decryption . . . 14
- 2.6. Notation . . . 15

3. Background on Cryptanalysis of Block Ciphers 17
- 3.1. Complexity of Attacks . . . 17
- 3.2. Linear Cryptanalysis . . . 18
- 3.3. Differential Cryptanalysis . . . 21
  - 3.3.1. Differentials . . . 22
  - 3.3.2. Extracting Key Information . . . 23
  - 3.3.3. Differential Trails . . . 24
- 3.4. Truncated Differentials . . . 25
- 3.5. Impossible Differentials . . . 25

4. Security Properties of AES 27
- 4.1. Design Aspects of AES . . . 27
  - 4.1.1. AES S-Box . . . 27
  - 4.1.2. Proper Diffusion . . . 28
  - 4.1.3. Number of Rounds . . . 28
  - 4.1.4. Key Schedule . . . 29
- 4.2. The Saturation Attack . . . 29
  - 4.2.1. Balanced Property . . . 30
  - 4.2.2. Basic Attack on 4 Rounds . . . 31

4.2.3. Attacking 5 Rounds . . . 32
4.2.4. Extension to 6 Rounds . . . 32
4.2.5. Improvement using Partial Sums . . . 34

II. Recent Attacks on AES **37**

5. Introduction **39**

6. The Multiset Attack **41**
6.1. The 3-Round Distinguisher by Gilbert and Minier . . . 41
6.2. 4-Round Distinguisher by Demirci and Selçuk . . . 42
6.3. Multiset Tabulation . . . 43
6.4. Differential Enumeration . . . 45
6.5. The Multiset Attack on 7-Round AES . . . 48
6.5.1. Performance of the Attack . . . 49
6.6. Further Improvements and Trade-offs . . . 49

7. Low Data Complexity Attacks **51**
7.1. Theoretical Basics . . . 53
7.2. Low Data Complexity Attacks . . . 55
7.2.1. Attacking 2-Round AES-128 without Last `MixColumns` . . . 56
7.2.2. An Attack on 1 Round with 1 Known Plaintext . . . 58
7.2.3. Attack on 2 and 3 Rounds Using 2 Chosen Plaintexts . . . 59
7.2.4. Attack on 3 Rounds with 9 Known Plaintexts . . . 61
7.3. Known-Plaintext Attack on 6-Round AES-128 . . . 62

8. Biclique Attacks on AES **65**
8.1. The Basics of Biclique Cryptanalysis . . . 65
8.1.1. Bicliques . . . 66
8.1.2. Concept of a Biclique Attack . . . 66
8.2. Biclique Construction for Block Ciphers . . . 67
8.2.1. Independent Related-Key Differentials . . . 67
8.2.2. Interleaving Related-Key Differentials . . . 69
8.3. Methods for Key Recovery . . . 69
8.3.1. Long-Biclique . . . 70
8.3.2. Matching with Precomputations . . . 71
8.3.3. Independent-Biclique . . . 72
8.4. Biclique Attack on the Full AES-128 . . . 72
8.4.1. Key Recovery . . . 73
8.4.2. Complexity Analysis . . . 75
8.5. Overview of other Biclique Attacks on AES . . . 76

9. Summary **77**

III. Analysis and Implementation **79**

10. Analysis of Single-Key Attacks **81**
10.1. Combining LDC Attacks with Square-like Distinguishers . . . 81
10.1.1. Balanced Property . . . 82
10.1.2. Multiset Tabulation . . . 82
10.1.3. Summary . . . 82
10.2. Attacking 7-Round AES-128 Using Differential Enumeration and LDC Attacks 83
10.2.1. Differential Enumeration . . . 83
10.2.2. Adapting the LDC Attack . . . 84
10.2.3. The Full Attack . . . 84
10.2.4. Improving the Attack . . . 86

11. Implementation of the Biclique Attack on AES-128 **87**
11.1. Hardware Implementation by Bogdanov et al. . . . 87
11.1.1. Theoretical Attack . . . 88
11.1.2. Hardware Implementation . . . 93
11.2. Background on AES-NI and Optimization . . . 94
11.2.1. Intel AES Instruction Set . . . 94
11.2.2. Optimizing for Performance . . . 95
11.2.3. AES-NI Specific Improvements . . . 97
11.3. Software Implementation . . . 100
11.3.1. Test Environment . . . 100
11.3.2. Brute Force Reference Implementation . . . 101
11.3.3. Biclique Implementation . . . 102
11.4. Performance Results . . . 104

12. Conclusions **107**

A. AES Key Schedule **109**
A.1. Pseudo Code for Key Schedule Algorithm . . . 109
A.2. AES-NI Implementation . . . 110

B. Measuring CPU Cycles on x86 CPUs **113**
B.1. The Time Stamp Counter . . . 113
B.2. Measurement Process . . . 114

Bibliography **123**

1. Introduction

Encryption is the process of transforming information into a format that is unreadable for anybody without the knowledge of a secret piece of information called *key*. With its origins dating back to the ancient Egyptians, who used unknown hieroglyphs for secret communication, cryptography has today evolved into a large field of research with a wide variety of different algorithms. Since the beginning of the computer era, it has also become more and more present in many areas of our lives. Whenever we use an electronic device, encryption is always involved at some step. Whether it is while buying a train ticket, withdrawing cash from an ATM, or making a phone call, encryption is used in some form to ensure confidentiality of information.

But not everything is perfect and encryption methods can have weaknesses too. Such flaws make encryption algorithms prone to attacks and easier to break. History has shown in many occasions that it is often a race between those who use cryptographic algorithms and those who break them. This has not changed today, when researchers try different methods of cryptanalysis to find vulnerabilities in algorithms and their areas of use.

Additionally the continuous increase in computing power made exhaustive search for keys, so-called *brute force attacks*, faster over time. Therefore, ancient encryption algorithms can be broken within seconds using today's computers. Even the Data Encryption Standard (DES), which was developed in the early 1970s can be broken today. Back then, when it was computationally infeasible to try all keys of its 56-bit key space. Years later, in 1998, the Electronic Frontier Foundation (EFF) designed and built custom hardware for just US $250,000 which is able to try all possible keys and decrypt any DES ciphertext within a few days [Fou98].

These reasons lead to the continuous invention of new algorithms and encryption techniques which are commonly categorized into 2 areas: symmetric and asymmetric ciphers. Symmetric-key algorithms use the same key for encryption and decryption, asymmetric ciphers use different keys for encryption and decryption. Symmetric-key algorithms are further split into block ciphers and stream ciphers. Where stream ciphers work by combining variable length plaintext with a stream of key material of equal length, block ciphers split the plaintext into sequences of fixed length called *blocks* and encrypt them using a key of fixed length.

Nowadays, one of the most commonly used block ciphers is the Advanced Encryption Standard (AES), which is – as the name already indicates – a standardized cipher. Announced in November 2001 by the National Institute of Standards and Technology (NIST) as the successor to DES, AES is now used worldwide on many different devices ranging from smart cards with 8 bit processors to highly parallel server systems. AES has been used for more than 10 years and there have been many attempts to fully break it. However, up until

today there is no publicly known attack which enables an attacker to extract information from AES-encrypted ciphertexts within feasible amount of time[1]. Nevertheless, there have been multiple publications which discovered weaknesses in the algorithm, although, many of these weaknesses require very special circumstances in order the exploit them. Thus, AES can still be used in practice without worrying about its security. However, since 2008 new and improved attacks on AES surfaced which attempt to reduce the theoretical security margin of AES. A prime example for this are the biclique attacks on all variants of AES [BKR11]. These attacks are the first to apply to the unmodified versions (full number of rounds) of AES. However, due to their time complexity (for example $2^{126.15}$ on AES-128), they are only marginally faster than exhaustive key search. Another recent attack is the multiset attack [DKS10]. This attack is basically an improvement of the saturation attack on up to six AES rounds [DR02] which was published together with the specification of AES. Although, AES was specifically designed with enough security margin to withstand the saturation attack, the multiset attack applies to all variants of AES up to seven rounds and has a time complexity of about 2^{103}. A completely different approach was taken for low data complexity attacks on AES [BDD$^+$10]. These attacks aim for minimal data requirements instead of attacking as many rounds as possible. One example is the low data complexity attack on two AES rounds which requires only two chosen plaintexts to recover the full encryption key within at most 2^8 encryptions. Such attacks alone are no threat to AES, however, they can be used as building block for more complex attacks on more rounds.

The main goals of this thesis are to analyze these recent attacks, verify certain techniques, and look for possible improvements of those attacks. Thus, we first give a thorough description of each attack and the techniques used to construct this attack. To find improvements on these attacks, we investigate the possibilities of combining techniques from multiple recent attack. Here, we focus primarily on using low data complexity attacks as building block for more complex attacks. Consequently, our attempts center on single-key attacks on AES-128. Another major part of this thesis is an in-depth analysis of the biclique attack on the full AES-128. This is of interest because this attack has only a marginal advantage over exhaustive key search. At first glance it seems that this marginal advantage is lost by the overhead of an implementation which would render this attack completely useless. However, with our software implementations of the biclique attack and a standard brute force attack, we confirm that this advantage still exists in practice, but is even smaller. As these implementations use Intel's AES-NI technology and were specifically optimized for high performance, we also provide a exhaustive description of how we achieve these high-speed implementations.

We introduce the Advanced Encryption Standard in Chapter 2, followed by Chapter 3 with background information on cryptanalysis of block ciphers. Part I concludes in Chapter 4 on the basic security properties of AES. In Part II we turn to the above mentioned recent attacks and present them in full detail. Beginning with Multiset attack in Chapter 6, we continue with a collection of attacks with particularly low data complexity introduced by

[1] We refer here to full versions of AES and also do not include specific implementation or side channel attacks, because those use vulnerabilities of the implementation and not weaknesses of the theoretical algorithm itself. A feasible amount of time means a time complexity below about 2^{64} AES encryptions.

Bouillaguet et al. in Chapter 7, and end this part with Chapter 8, where we describe the Biclique attack by Bogdanov et al. Part III covers our contributions where we construct hypothetical improvements by combining the techniques from the presented recent attacks, and verify or disproof the applicability of these improvements in Chapter 10. In Chapter 11 we concentrate on the biclique attack on the full AES-128 and verify its claimed complexities in practice by implementing it in software. Finally, we conclude this thesis in Chapter 12 with a summary of our findings.

Part I.

Preliminaries

2. The Advanced Encryption Standard

AES was developed by the Belgian researchers Joan Daemen and Vincent Rijmen and was one of the submissions to a competition by the NIST in 1997. The goal of this competition was to find a successor to DES. Initially, the algorithm was called Rijndael, a wordplay with the last names of the designers. In 2000 Rijndael was selected as the winner of the competition and published as U.S. FIPS PUB 197 [NIS01]. Due to the wide adoption of Rijndael, it is nowadays simply known as AES.

The most important design goals of AES are its simple design, adequate security properties, and the possibility to create fast and compact soft- and hardware implementations [DR02]. In this chapter we describe the basic operations which achieve these goals.

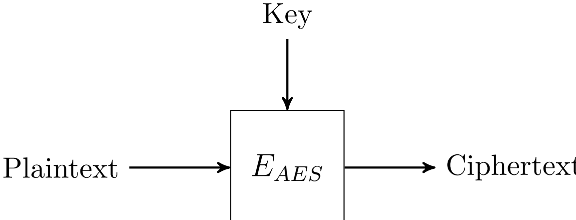


Figure 2.1.: The AES encryption function, which has a fixed length input (plaintext) and output of the same length (ciphertext). The encryption key is the same for encryption and decryption.

2.1. The Structure of AES

Similar to many other block ciphers, AES works by repeating a round transformation a defined number of times on a given input block. For AES, this input has a fixed length of 128 bits and represents the plaintext to be encrypted. The result of each transformation during encryption and decryption is called *state* and has a fixed length of 128 bits. Since AES operates only at byte level, plaintext, ciphertext and intermediate states are commonly represented as a 4×4 matrix of 16 bytes. Figure 2.2 depicts the order of these 16 bytes within a state.

The simplistic design of AES is based on a substitution-permutation network (SP-network) [MVO01, p. 251]. Such ciphers iterate the same round transformation, which is made up of a substitution followed by a permutation operation. At the end of each round, a round key is added to the current state. The repeated execution of the round transformation on the plaintext produces the ciphertext. In contrast to AES, DES is based on a Feistel network [MVO01, p. 251]. The advantage of AES over DES is that the round transformation

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

Figure 2.2.: Representation of the AES state as 4×4 matrix. The numbers indicate the position of the corresponding byte in the cipher's input array.

operates on *all* state bytes. This allows full diffusion within only two rounds, where Feistel networks need at best three rounds and in practice normally four rounds. The importance of diffusion in ciphers was also noted by Shannon in 1949 [Sha49] and describes the grade of dependence between input and output bits. For a secure cipher the diffusion should be high. Full diffusion means that every bit of the state depends on *all* input state bits from n rounds before. In the case of AES, $n = 2$ [DR02, p. 41]. Another advantage of AES is that it allows encryption and decryption to be similar, thus making implementations easier and smaller in terms of code complexity and code size.

In AES each round key is derived from the initial key. The *key schedule* takes the initial key (given as input to the algorithm) and outputs a separate round key for each round transformation. This is also called *key expansion*.

AES offers 3 different flavors, which only differ in their respective key length and the number of rounds. Compared to the original Rijndael proposal, AES only allows a single block size of 128 bits and three key lengths of 128, 192, and 256 bits, where Rijndael offers variable key and block lengths. Rijndael allows key and block length to be set independently to a multiple of 32 bits between 128 bits and 256 bits. Apart from that, both specifications are equivalent. The three versions of AES are commonly referred to as *AES-128*, *AES-192*, and *AES-256*, where the number indicates the key size. For AES, the number of rounds is defined by the key size: AES-128 has 10 rounds, AES-192 12 rounds and AES-256 14 rounds.

The high-level outline of AES is described in pseudo code in Listing 2.1 and roughly works as follows:

1. Combine plaintext with initial key using `XOR`
2. Apply $n - 1$ round transformations on the current state, where n is the number of rounds defined by the key size.
3. Apply last round

```
1 RijndaelEncrypt(State, CipherKey)
2 {
3     KeyExpansion(CipherKey, ExpandedKey);
4     AddRoundKey(State, ExpandedKey[0]);
5     for (i=1; i<Nr; i++) Round(State, ExpandedKey[i]);
6     FinalRound(State, ExpandedKey[Nr]);
7 }
```

Listing 2.1: High level description of AES given in C-pseudo code as depicted in [DR02], where `Nr` is the number of rounds.

2.2. Finite Fields

The operations used in the transformations of AES are finite field calculations over $GF(2^8)$. Here, we give a short introduction to the basics and refer the interested reader to [DR02, pp. 9-29], which covers all the mathematical background needed to understand the inner workings of AES in more detail.

Definition 2.1 (Abelian Group)**.** A set G and an operation $+ : G \times G \to G, (a, b) \mapsto a + b$, denoted by $< G, + >$, is called Abelian group if the operation fulfills the following properties in G:

- Associative: $\forall a, b, c \in G : (a + b) + c = a + (b + c)$
- Commutative: $\forall a, b \in G : a + b = b + a$
- Closed: $\forall a, b \in G : a + b \in G$
- Neutral element: $\exists \mathbf{0} \in G : a + \mathbf{0} = a$
- Inverse element: $\forall a \in G, \exists b \in G : a + b = \mathbf{0}$

A common example for an Abelian group is $< \mathbb{Z}, + >$ with the set of integers $\mathbb{Z}$ and addition as operation. Another example with a set containing only a finite number of elements is $< \mathbb{Z}_{\ltimes}, + >$ where $\mathbb{Z}_{\ltimes}$ contains integers from 0 to $n - 1$ and $+$ is the addition modulo n.

Definition 2.2 (Field)**.** $< S, +, \cdot >$ is called a field, if:

- $< S, + >$ is an Abelian group
- $< S \setminus \{\mathbf{0}\}, \cdot >$ is an Abelian group
- $+$ is distributive with respect to $\cdot$ in S

One example for a field is the set of all real numbers, with addition and multiplication.

A special subset of fields are *finite fields*, which contain only a finite number of elements. For a set with m elements there exists a finite field if and only if (iff) $m = p^n$, where p is prime and n a positive integer [DR02]. Such fields are also called *Galois fields* and are

denoted by $GF(p^n)$. The simplest form of a Galois field is $GF(p)$, which can be represented as integers from 0 to $p-1$. Consequently, the operations are then addition modulo p and multiplication modulo p. In case $p=2$, the field is called *binary field*. For $n>1$, $GF(p^n)$ can be represented as polynomials over $GF(p)$. In this case, the addition in the finite field is performed by adding coefficients of equal power. For example if $p=2$ and the polynomials are $x^6+x^4+x^2+x+1$ and x^7+x+1, we can add them as follows:

$$(x^6+x^4+x^2+x+1)+(x^7+x+1)=x^7+x^6+x^4+x^2.$$

Multiplication in $GF(p^n)$ is performed as the algebraic product of both polynomials modulo an *irreducible polynomial* $m(x)$ of degree n. An irreducible polynomial is just a polynomial that cannot be factored into smaller polynomials. More formally:

Definition 2.3 (Irreducible Polynomial)**.** Let $a(x), b(x), c(x)$ polynomials in $GF(p)$ and $a(x), b(x)$ with degree >0. $c(x)$ is irreducable iff there exists no $a(x), b(x)$ such that $c(x)=a(x)\cdot b(x)$.

For AES, the irreducible polynomials $m(x)=x^4+1$ and $m(x)=x^8+x^4+x^3+x+1$ are used.

2.3. The Round Transformations

One round in AES consists of four round transformations. The order and purpose of each transformation is given as follows (see also Figure 2.4):

1. `SubBytes`: A byte-wise substitution (*S-Box*) that provides non-linearity
2. `ShiftRows`: Permutation of bytes for diffusion within each row
3. `MixColumns`: Linear operation that provides diffusion within columns
4. `AddRoundKey`: `XOR` with round key

The last round differs from a standard (full) round by leaving out the `MixColumns` operation. Moreover, before the application of first round, the initial key is added[1] to the plaintext. This initial key is often called *whitening key*.

In order to decrypt a ciphertext, each one of the above operations is invertible, and has an inverse counterpart. These operations are: `InvSubBytes`, `InvShiftRows` and `InvMixColumns`. Note that we do not require an inverse operation for `AddRoundKey` since the $x\oplus x=0$. Thus, the inverse operation to `AddRoundKey` is also simply the `XOR` operation with the round key.

In the following sections, we will introduce each round transformation of an AES round and explain its purpose and functionality in more detail.

[1] In a binary field, bitwise `XOR` of two bytes is the same as bitwise addition modulo 2, so when we write addition, we perform an `XOR` operation. Note that the same is true for subtraction modulo 2.

		y															
		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
x	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 2.1.: The AES S-Box table. The values are given in hexadecimal notation and a cell (x, y) holds S-Box(xy).

2.3.1. SubBytes

Compared to DES, AES has only one S-Box that is applied to all state bytes. `SubBytes` represents the substitution stage of an SP-network and applies the S-Box to each byte separately. The S-Box is a bijective mapping that adds non-linearity to an AES round. Non-linearity is an important factor for a block cipher to provide strength against linear and differential cryptanalysis despite the rather low number of rounds. The mapping has to be bijective to make the S-Box invertible for the decryption process.

The design of the AES S-Box is based on the multiplicative inverse in $GF(2^8)$ modulo the irreducible polynomial $m(x) = x^8 + x^4 + x^3 + x + 1$. Additionally, an invertible affine transformation is applied. This affine transformation is specifically designed to remove fixed points ($SB(x) \oplus x = 0$) and yields a more complex algebraic expression for the full S-Box equation when combined with the multiplicative inverse.

Especially for software implementations, the S-Box itself is realized by a simple lookup table (see Table 2.1) since it only takes 256 bytes to store. Tables of this size fit nicely into the CPU cache and make lookup operations very fast. For hardware implementations, it is useful to calculate the S-Box on-demand[2] because such designs can be made quite compact and need less space in terms of gates [BP09, Can05]. The downside of lookup tables used in software implementations are side channel attacks. Since the time of a lookup operation takes depends on the value that is looked up, such implementations are prone to timing attacks. In a cache timing attack, an adversary uses the time required to look up an S-Box

[2]The full equations of the S-Box are computed for each value.

value from the table stored in the CPU cache to retrieve the actual value that is looked up. This allows to conclude the value of the secret encryption key. The first attack of this kind on AES was presented by Bernstein in [Ber05]. To avoid this weakness, so called *bitslice* implementations calculate the S-Box on-demand, which requires the same amount of time for every possible value.

2.3.2. ShiftRows

`ShiftRows` is a simple transposition operation that shuffles the bytes of the current state. This transposition is a byte-wise rotation (cyclical shift) of each row of the state. To provide optimal diffusion, each row is shifted by a different offset. The first row is not rotated, the second is rotated by one, the third by two and the fourth by three bytes. `ShiftRows` was further designed to maximize resistance against truncated differential attacks and saturation attacks like the Square attack described in section 4.2 [DR02, p. 37].

2.3.3. MixColumns

`MixColumns` is another permutation operation with the purpose to provide high diffusion within columns of the state. It is a linear transformation that mixes all four bytes of a column with each other. This mixing is again based on finite field computations over $GF(2^8)$. Here, the columns are interpreted as polynomials over $GF(2^8)$ which are multiplied modulo $x^4 + 1$ with a constant polynomial $c(x) = 03 \cdot x^3 + 01 \cdot x^2 + 01 \cdot x + 02$. The fixed polynomials are chosen such that it is possible to write fast `MixColumns` implementations while still having proper diffusion properties. Its calculation is commonly written as multiplication of each column (4-byte vector $[a_0, \ldots, a_3]^T$) with a constant matrix:

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} \texttt{02} & \texttt{03} & \texttt{01} & \texttt{01} \\ \texttt{01} & \texttt{02} & \texttt{03} & \texttt{01} \\ \texttt{01} & \texttt{01} & \texttt{02} & \texttt{03} \\ \texttt{03} & \texttt{01} & \texttt{01} & \texttt{02} \end{bmatrix} \times \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}.$$

Like all other round transformations, it is designed to be invertible for the decryption process. For the inverse operation `InvMixColumns` it is only necessary to change the coefficient matrix, resulting in:

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} \texttt{0E} & \texttt{0B} & \texttt{0D} & \texttt{09} \\ \texttt{09} & \texttt{0E} & \texttt{0B} & \texttt{0D} \\ \texttt{0D} & \texttt{09} & \texttt{0E} & \texttt{0B} \\ \texttt{0B} & \texttt{0D} & \texttt{09} & \texttt{0E} \end{bmatrix} \times \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}.$$

Instead of evaluating the above equation for each `MixColumns` operation, software implementations commonly use lookup tables. These tables can be combined with the lookup table for `SubBytes` which yields 4 tables of 1024 bytes each. This is a common implementation on 32-bit platforms where all tables take up only 4 kB of memory and again fit nicely into most 32-bit CPU caches to yield fast table lookups [DR02, pp. 53-62].

2.3.4. AddRoundKey

Each round in AES requires a 128-bit round key. This key is added to the current state using a byte-wise `XOR` operation. To revert the `AddRoundKey` operation, we just add the same round key again. Thus, `AddRoundKey` does not require a special inverse operation for the decryption algorithm.

2.4. Key Schedule

The key schedule provides the keys for each round. It derives each round key through a recursive algorithm from the previous round key, starting with the initial cipher key as the whitening key. The key schedule outputs `Nr+1` round keys where `Nr` denotes the number of rounds. The AES key schedule is designed for high performance with low memory usage while still providing efficient diffusion of the cipher key differences and appropriate non-linearity for security.

The key schedule operates on 4-byte blocks (words) w_i and is slightly different for each AES variant. As mentioned before, the input to the key schedule is the initial cipher key. Depending on the AES variant, the initial key is 128, 192, or 256 bits in length, which is equal to $N_k = 4, 6$, or 8 words respectively. The initial key is also directly used as the first N_k output words of the key schedule (the whitening key). The remaining output words $w_i, i > N_k$ are defined as:

$$w_i = \begin{cases} w_{i-Nk} \oplus v(w_{i-1}) & \text{if } i \bmod N_k = 0 \\ w_{i-Nk} \oplus \texttt{SubWord}(w_{i-1}) & \text{if } i \bmod N_k = 4 \text{ and } N_k > 6 \\ w_{i-Nk} \oplus w_{i-1} & \text{otherwise} \end{cases}$$

Where:

- $v(x)$ is defined as $v(x) = \texttt{SubWord}(\texttt{RotWord}(x)) \oplus \texttt{Rcon[}i/N_k\texttt{]}$.
- `RotWord()` rotates four bytes to the left by one, such that `[a0,a1,a2,a3]` becomes `[a1,a2,a3,a0]`.
- `SubWord()` applies the AES S-Box to four bytes.
- `Rcon` is the round constant, defined in $GF(2^8)$ by:

$$\begin{array}{lcl} \texttt{Rcon[1]} & = & x^0 \\ \texttt{Rcon[i]} & = & x^{i-1}, i > 1 \end{array} .$$

The special case for $i \bmod N_k = 4$ and $N_k > 6$ is only used in the key schedule of AES-256. Due to the larger key size, this case is required to introduce additional non-linearity. For a more detailed description of the key schedule algorithm and a implementation in pseudo-code see Appendix A.

Figure 2.3 shows how to retrieve the individual round keys from the output words w_i of the key schedule. Since the round key of each round has a fixed size of 128 bits for all variants of AES, the j-th round key rk_j consists simply of $w_{j \cdot 4}, w_{(j \cdot 4)+1}, w_{(j \cdot 4)+2}, w_{(j \cdot 4)+3}$.

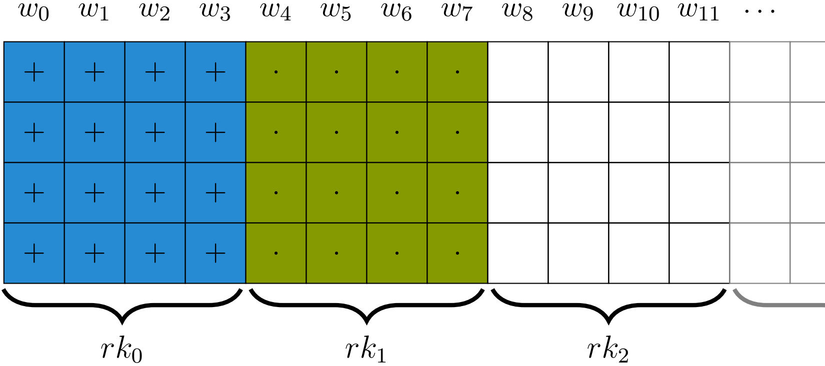


Figure 2.3.: Derivation of round keys from the key schedule's output words w_i (columns).

2.5. Decryption

For the decryption of a ciphertext produced by AES, we have to revert all the operations that were applied during the encryption process. To do this, we execute the round transformations in reverse order on the ciphertext, and also invert each round transformation (`SubBytes`, `ShiftRows` and `MixColumns`) itself.

`ShiftRows` is very simple to invert since it is only a rotation of each row by a constant factor. To get the inverse operation `InvShiftRows`, we just invert the direction of the rotation. Since `SubBytes` is a bijective mapping, we can just create a second lookup table that performs inverse substitution to get `InvSubBytes`. As already mentioned above, the `InvMixColumns` step is performed by multiplying each column with the inverse coefficient matrix. The only step left is to revert the `AddRoundKey` step. Since $x \oplus x = 0$, we can just add the same round key again, to undo the key addition. The order of operations within one full decryption round is then:

1. `AddRoundKey`
2. `InvMixColumns`
3. `InvShiftRows`
4. `InvSubBytes`

Note that in the last round of the encryption algorithm the `MixColumns` transformation is omitted. Hence, for the decryption we have to omit `InvMixColumns` of the first inverse round. A similar statement can be made about the initial `AddRoundKey` with the whitening key.

This primitive inversion of the encryption process would require us to create new code for the whole decryption process. This is not very advantageous, since it would be simpler to keep the round transformations in the same order as for the encryption process. This

would save space in hardware implementations, and would allow us to reuse code parts in software implementations. To make decryption more similar to encryption, certain algebraic properties of the basic operations of AES can be used. The resulting process is called *equivalent decryption algorithm* [DR02, pp. 45-50], and uses the following two properties:

1. The order in which `InvSubBytes` and `InvShiftRows` are executed does not matter since those steps work on each byte separately and the same S-Box is used for all bytes.

2. The order of `AddRoundKey` and `InvMixColumns` can be switched if `InvMixColumns` is applied to the round key before adding it to the current state.

Using these two properties, we get the same high-level process as for the encryption algorithm shown in Listing 2.1:

1. Run the key schedule.

2. `AddRoundKey` to ciphertext.

3. Apply `Nr-1` full (inverse) rounds.

4. Apply the last (inverse) round.

The only differences are that each round transformation is switched with its inverse counterpart, and for each `AddRoundKey` operation, we first apply `InvMixColumns` to this round key. This improvement makes decryption very similar to encryption, and allows equally efficient implementations.

2.6. Notation

Throughout this thesis we will use the notations shown in Figure 2.4. The rounds are numbered from `1` for the first round to `Nr` for the last round where `Nr` is 10, 12, or 14 for AES-128, AES-192, or AES-256, respectively. The round keys are numbered from 0 to `Nr`. Thus, in round i we use round key i denoted by rk_i. For `SubBytes`, `ShiftRows`, `MixColumns`, and `AddRoundKey` we use the abbreviations `SB`, `SR`, `MC`, and `ARK`.

We assign each state a name which specifies its exact position within the algorithm. The input state of each round is denoted by S_i where i is the round number. For the states within a round, we use the naming scheme $S_{i,\texttt{<input-of>}}$. The placeholder <`input-of`> is replaced with the abbreviation of the next operation, i.e. `SR`, `MC`, or `ARK`. For example, the state after `SubBytes` and before `ShiftRows` of round 4 is named $S_{4,\texttt{SR}}$. The next state after $S_{4,\texttt{SR}}$ is called $S_{4,\texttt{MC}}$, and so on. Further, to denote single bytes of a state, we use an array-like notation. So, the j-th byte (zero-based count) of a state S_i is written as $S_i[j]$. The order of the bytes within a state is shown in Figure 2.2.

In case there are multiple plaintexts and/or ciphertexts, we use a zero-base number in the superscript to distinguish them. For example, S_i^l denotes the l-th state S_i, and $S_i^l[j]$ denotes

the byte $S_i[j]$ of the l-th state. Moreover, $S^3_{4,\mathtt{SR}}[0]$ is the first byte in state $S_{4,\mathtt{SR}}$ of plaintext 3.

We denote consecutive round keys for rounds $i, i+1, \ldots$ as $rk_i, rk_{i+1}, \ldots$, and represent them as a 4×4 state matrix equal to an internal AES state shown in Figure 2.2. We distinguish the four columns of rk_i by $rk_{i,0}, rk_{i,1}, rk_{i,2}, rk_{i,3}$, and individual bytes of that key as $rk_i[0], rk_i[1], \ldots rk_i[15]$. For the key schedule of AES-128, we also define $v_i = \mathtt{SubWord}(\mathtt{RotWord}(rk_{i,3})) \oplus \mathtt{Rcon}[i+1]$.

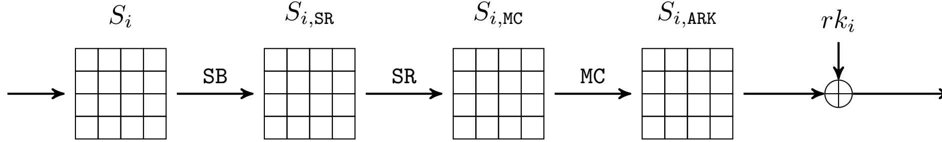


Figure 2.4.: The round transformation of AES with all steps and notations used throughout this thesis. rk_i denotes the i-th round key.

3. Background on Cryptanalysis of Block Ciphers

The theoretical security of an encryption algorithm is primarily concerned with two main aspects:

- Make exhaustive key search infeasible.
- Avoid short-cute attacks or make them infeasible.

Since exhaustive key search is easily conquered with large enough key sizes, we will concentrate on the second aspect. The goal of such attacks is to find certain properties within the internals of the cipher that allow to retrieve information on key bits. Such attacks basically exploit the internal structure of the cipher to be faster than exhaustive key search. For the designers of a cipher, it is practically impossible to make the cipher resistant against every imaginable attack since the internal structure differs from cipher to cipher. Moreover, there could still be attacks that have not been discovered yet. Still, many ciphers share common basic concepts like SP-networks (AES), or the Feistel structure (DES). Such concepts are common between multiple ciphers and thus, ciphers based on these concepts also share possible vulnerabilities that stem from them.

The two most prominent forms of cryptanalysis today are *linear cryptanalysis* and *differential cryptanalysis*. Both were initially applied to DES but are applicable to many other algorithms with similar structure. During the AES competition, many candidates provided proof that such attacks do not threaten their security. Nowadays, resistance against linear and differential cryptanalysis is a de-facto standard for each encryption algorithm to be accepted as *good* cipher.

This chapter provides the necessary background knowledge on linear and differential cryptanalysis as well as some other related methods such as truncated and impossible differentials. It is mainly based on [Kho10, Sch11] with additional information from [DR02, BS91, Mat93]. Readers already familiar with those concepts can safely skip this chapter.

3.1. Complexity of Attacks

To compare different attacks on the same block cipher with each other, a simple measure has to be found. This allows us to identify how much memory is needed, how long it takes and what other requirements are needed. Throughout this thesis, we use the following standard measures which are commonly used for this purpose: time complexity, data complexity and memory complexity.

Memory complexity is simply the amount of physical storage needed to perform the attack. It is measured in blocks of the same size as the input of the cipher uses. So, for AES this is 128-bit blocks. The data complexity defines the amount of plaintexts and/or ciphertexts needed to perform the attack. Additionally, we often differentiate between *chosen plaintexts* and *known plaintexts*. We speak of chosen plaintexts if the attacker needs to choose the values of each plaintext used in the attack. With known plaintexts, the attacker has more freedom since he does not need to influence the plaintexts which are encrypted but only needs to know their values. The unit for data complexity is equal to memory complexity, so for AES, it is 128-bit blocks.

Time complexity is probably the most important measure since it allows to determine if an attack is feasible or not. In the context of attacks on ciphers, feasible means than an attacker is able to perform the attack in an acceptable amount of time (a few seconds, minutes, hours, days or even months). Infeasible attacks often take thousands of years and are thus merely theoretical attacks. Nevertheless, such attacks are still important to consider because hardware becomes faster and more affordable over time. Moreover, such attacks can often provide the basis for faster feasible attacks. There is no fixed boundary between feasible and infeasible attacks but normally, attacks below about 2^{60} - 2^{64} are assumed to be feasible. This boundary is subject to change because of new developments in hardware and the resulting faster implementations of algorithms. Thus, an attack on some block cipher with time complexity 2^{60} was infeasible ten years ago but might very well be feasible today.

The time complexity of an attack is also often compared to the time complexity of a simple brute force attack on the same cipher. In a *brute force* attack or *exhaustive search* for keys, an attacker tries all possible keys for a selected plaintext and its corresponding ciphertext until the matching key is found. Exhaustive search basically provides an upper bound for attacks since an attack with larger time complexity as exhaustive search makes no sense. For brute force attacks, the data and memory complexity is simply 2 because only one plaintext and its corresponding ciphertext are required.

The unit for time complexity is "equivalent encryptions" in the block cipher. So for AES, a hypothetical attack with time complexity of 2^{140} needs the same amount of time as encrypting 2^{140} plaintexts. It seems, an attack with such a high time complexity is useless because exhaustive search in AES-128 takes only 2^{128} encryptions. However, this hypothetical attack makes sense since there are 3 variants of AES with different key sizes and this attack would be faster than exhaustive search for AES-192 and AES-256. For attacks on round-reduced variants of AES, we measure the time complexity in encryptions of that reduced variant. Therefore, an attack on six rounds of AES with time complexity 2^{53} takes the same time as 2^{53} 6-round AES encryptions.

3.2. Linear Cryptanalysis

Linear cryptanalysis was invented by Matsui and first applied to DES [Mat93]. However, it is applicable to many other block ciphers with an iterative structure that rely on some non-linear transformation. Linear cryptanalysis is a *known-plaintext attack* which means the

attacker has to know the plaintext that gets encrypted and its corresponding ciphertext in order to apply it.

The general idea is to approximate the non-linear parts of a round transformation with a linear expression. By doing this, we make the whole round just a linear function which allows to calculate possible keys by just knowing input and output. It is common for block ciphers to use an S-Box as the only non-linear operation. Thus, we first concentrate on approximating the non-linear part without considering the other steps of the round. Later, we are going to extend this concept to a full round and finally to the full cipher.

To approximate the non-linear substitution S, we choose α and β such that the linear expression of the form

$$\alpha \cdot x = \beta \cdot y$$

is true for as many inputs x and outputs y as possible. The specific bits of the input and output are selected using the bit masks α and β. Furthermore, the operator $\cdot$ is the bitwise `AND`. There are multiple possibilities for these masks, in fact, for an S-Box with n input bits and m output bits, there are $(2^n - 1) \times (2^m - 1)$ possible approximations.

Each approximation works only for a subset of all possible input and output masks, hence, we assign each one a probability p. The probability that an expression holds is

$$p = \frac{x}{2^n},$$

where x is the number of times the linear expression holds for a particular input and output mask.

Definition 3.1 (Linear Probability Bias)**.** Let p be the probability of a linear expression. Then, the *linear probability bias* is defined as

$$\epsilon = p - \frac{1}{2}$$

and its magnitude represents the effectiveness of the linear expression.

The value of ϵ has the following meaning:

- $\epsilon = 0$: No gain in information with this expression.
- $\epsilon > 0$: The linear approximation $\alpha \cdot x = \beta \cdot y$ is good.
- $\epsilon < 0$: The approximation $\alpha \cdot x = \beta \cdot y \oplus 1$ is good.

To find the best linear approximation for our attack, we evaluate all possible masks and collect the resulting bias values in the *Linear Approximation Table*.

Definition 3.2 (Linear Approximation Table)**.** Let ϵ be the bias of a linear expression $\alpha \cdot x = \beta \cdot y$ that approximates a non-linear n to m bit function S. For all possible masks α and β, the linear approximation table contains the bias $\epsilon \cdot 2^n$.

Until here, we just concentrated on finding a linear approximation for the non-linear part of the round transformation. The linear approximation table provides the required knowledge to select a good approximation with a high enough bias. Since a round transformation normally includes a key addition step, and our goal is to get information on the key from just plaintext and ciphertext, we modify the linear expression to include the round key:

$$(\alpha \cdot x) \oplus (\beta \cdot y) = \gamma \cdot k,$$

where k is the round key and γ is its mask.

For extending the linear approximation to cover multiple rounds, we take approximations of a single round and combine them such that the final approximation only contains plaintext, ciphertext, and key bits. Such a combination is called *linear characteristic*.

Definition 3.3 (Piling-up Lemma [Mat93])**.** Let X_i be an independent boolean expression with probability $Pr(X_i = 0) = \frac{1}{2} + \epsilon$. Then, the probability of the linear characteristic $X_0 \oplus X_1 \oplus \cdots \oplus X_n = 0$ is defined as:

$$Pr(X_0 \oplus X_1 \oplus \cdots \oplus X_n = 0) = \frac{1}{2} + 2^{n-1} \prod_{i=1}^{n} \epsilon_i$$

Now, we have all the tools needed to construct a linear characteristic for a cipher. If the cipher is susceptible to linear cryptanalysis, we are able to find a characteristic with a high probability. This means that, given a set of plaintext-ciphertext pairs, it is likely that for some of those pairs the linear characteristic applies. This enables us to retrieve information about the key as follows:

In general, there are multiple ways to get key information using linear cryptanalysis. We will concentrate on Matsui's Algorithm 2 [Mat93], which is shown in Algorithm 1. Assume a cipher with `Nr` rounds and a corresponding linear approximation $(\alpha \cdot p) \oplus (\beta \cdot w) = \gamma \cdot k$ for the first `Nr-1` rounds that has a high enough probability. The approximation depends on two values: the plaintext p, and the intermediate state w which is the input to the last round or equally the state after `Nr-1` rounds. The mask β indicates which bits from the intermediate state are involved in the expression thus also defining the *active* S-Boxes of the last round.

For those active S-Boxes, we take all values k_i and a set of plaintext-ciphertext pairs (p_j, c_j). We decrypt the ciphertext through the last round (Line 7 in Algorithm 1) and calculate the value of the linear expression with the resulting values for the intermediate state and the plaintext. Since the linear approximation we chose has a high probability, it is likely that it holds for the correct key guess and does not hold for wrong key guesses. Thus, the key with the highest count of pairs for which the linear expression holds is the correct key.

It becomes clear that to withstand linear cryptanalysis, a cipher must make it hard for an attacker to create a linear approximation for the whole cipher. This is achieved by making the bias of the linear approximations as low as possible. Additionally the permutation should take care that the number of active S-Boxes is high.

Algorithm 1 Matsui's Algorithm 2

```
1:  function Algorithm2(L(p, c), F^{-1}(c, k))
2:      for all key guesses k_i do              ▷ Corresponding key bits for active S-Boxes
3:          T_0^{k_i} ← 0
4:          T_1^{k_i} ← 0
5:      end for
6:      for all plaintext-ciphertext pairs (p_j, c_j) and k_i do
7:          w ← F^{-1}(c_j, k_i)                ▷ inverse round transformation F^{-1}(c, k)
8:          a ← L(p_j, w)                       ▷ linear expression L(p, c)
9:          T_a^{k_i} ← T_a^{k_i} + 1
10:     end for
11:     return k_i with highest difference between T_0^{k_i} and T_1^{k_i}
12: end function
```

3.3. Differential Cryptanalysis

Differential cryptanalysis was found by Biham and Shamir and initially applied on DES and DES-like crypto-systems in the late 1980s [BS91]. The authors noted in their analysis that DES is remarkably resistant against this kind of attack. Later a member of the original DES team at IBM stated that they had been aware of this kind of attack before Biham and Shamir and designed DES specifically to be resistant against it.

Differential cryptanalysis is a *chosen-plaintext attack* that works with pairs of plaintexts (P, P^*) and corresponding ciphertext pairs (C, C^*). "Chosen-plaintext attack" means that the attacker is able to choose or at least influence the plaintext that gets encrypted. The pairs are formed using `XOR` so, the difference ΔP of two plaintexts P and P^* is defined as:

$$\Delta P = P \oplus P^*.$$

The idea is to follow such a difference through all rounds of the cipher and construct a relation between a plaintext difference ΔP and a ciphertext difference ΔC. This allows to calculate the values of certain key bits as we are going to explain in the course of this chapter.

In essence, differential cryptanalysis is, like linear cryptanalysis, an attempt to find an approximation for the non-linear operations in a cipher. However, for differential cryptanalysis we look at pairs of plaintexts and their difference ΔP. By tracing the difference ΔP through the cipher's operations, we analyze the changes of the differences. The influence of linear operations is predictable since it modifies the difference only in a deterministic and bijective way. It holds for all linear transformations that by knowing the input difference (Δx), we can compute the output difference (Δy). However, for non-linear transformations we are unable to conclude the output difference from a known input difference because a single Δx can be transformed into multiple output differences. Which specific output difference is determined by the actual values (x, x^*) of each pair we do not know since we only know the difference (Δx) between them. Differential cryptanalysis is an attempt to reduce the amount of possible output differences for a given input difference.

		Δy												
		00	01	02	03	04	05	06	07	08	09	0A	0B	0C	...
Δx	00	256	0	0	0	0	0	0	0	0	0	0	0	0	...
	01	0	2	0	0	2	0	2	0	2	2	2	2	2	...
	02	0	0	0	2	2	2	2	2	0	0	0	2	2	...
	03	0	0	2	0	2	2	0	0	2	0	2	2	2	...
	04	0	0	0	0	0	0	0	0	0	2	0	0	0	...
	05	0	0	0	0	2	0	0	0	4	2	0	0	2	...
	06	0	2	0	0	2	2	0	0	0	2	0	2	4	...
	07	0	2	0	0	0	0	2	0	2	2	2	0	0	...
	08	0	0	2	2	0	0	0	0	0	2	0	2	2	...
	09	0	0	2	2	0	0	2	2	0	0	0	0	2	...
	0A	0	0	2	2	4	0	2	2	0	2	2	0	2	...
	0B	0	2	0	0	0	0	0	2	2	2	0	0	2	...
	0C	0	0	2	2	0	0	2	2	2	2	2	0	0	...
	0D	0	2	2	0	0	0	2	2	2	2	2	2	0	...
	0E	0	0	2	2	2	2	0	2	2	0	2	2	0	...
	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...

Table 3.1.: The partial difference distribution table for the AES S-Box.

3.3.1. Differentials

In their original paper on differential cryptanalysis, Biham and Shamir discovered that a random input difference does not yield every possible output difference for the DES S-Boxes. This is also true for non-linear transformations of many other ciphers. Thus, some transitions from an input difference to an output difference are impossible and some are more likely than others. Such a transition $\Delta x \to \Delta y$ is called *differential*. Hence, for each possible differential we can construct the following table:

Definition 3.4 (Difference Distribution Table (DDT))**.** Let S be a non-linear transformation with n input bits and m output bits. Construct the $2^n \times 2^m$ difference distribution table by filling it with the *number of right pairs* (pairs yielding the differential transition)

$$N(\Delta x \to \Delta y) = \#\{(x, x^*) | \Delta x = x \oplus x^* \text{ and } S(x) \oplus S(x^*) = \Delta y\},$$

for all differences Δx, Δy [BS91].

An example for such a difference distribution table is the DDT for the AES S-Box as shown in Table 3.1. Since the AES S-Box operates on bytes, $n = m = 8$, and the table has a dimension of $2^8 \times 2^8$. The cells with value zero indicate impossible differential transitions. Note that the difference is only zero if both input values are equal. Since an S-Box is invertible, the output values are equal and thus, only a zero output difference is possible. Therefore, the first row and first column have a non-zero value only in $(0, 0)$. Further, note that each value always occurs an even number of times since for a pair (a, a^*), the pair (a^*, a) yields the same input and output difference.

The difference distribution table forms the basis of differential cryptanalysis. Given an input difference, only a fraction of all possible difference transitions can occur and the likelihood for each combination of input, output differences varies. We can extract this probability directly from the DDT.

Definition 3.5 (Difference Propagation Probability)**.** Let Δx be an arbitrary input difference and Δy an arbitrary output difference. The probability $p = Pr(\Delta x \to \Delta y)$ that the non-linear operation S transforms $\Delta x \xrightarrow{p} \Delta y$ is called *difference propagation probability* or *difference probability* [Kho10] and is calculated by

$$\begin{aligned} Pr(\Delta x \to \Delta y) &= \frac{1}{2^n} \cdot N(\Delta x \to \Delta y) \\ &= \frac{1}{2^n} \cdot \#\{(x, x^*) | \Delta x = x \oplus x^* \text{ and } S(x) \oplus S(x^*) = \Delta y\}, \end{aligned}$$

where the number of right pairs $N(\Delta x \to \Delta y)$ is given by the DDT.

3.3.2. Extracting Key Information

To explain the basic concept of how to find the key bits used for encryption, we concentrate on a single round first and extend this concept to multiple rounds in the next section. As we have already shown, the linear operations of a round influence the input differences only in a deterministic way, thus, by knowing one of the differences at either input or output, we are able to calculate the corresponding other one. So, we omit linear operations in the next example and concentrate on the substitution using the S-Box and the key addition.

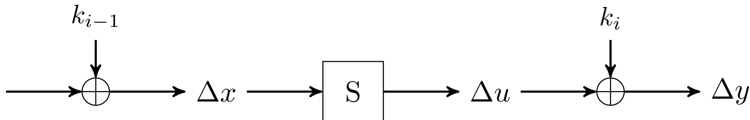


Figure 3.1.: A very simple round of an example cipher without any linear operations except the key addition.

Assume we only know the input difference Δx and both corresponding values at the output (y, y^*) of the round shown in Figure 3.1. From the output values we calculate the output difference Δy. Further, we can calculate $u = y \oplus k$ and $u^* = y^* \oplus k$. Since k_i is equal for u and u^*, the key addition does not change the output difference, and we get

$$\begin{aligned} \Delta y &= y \oplus y^* \\ &= (u \oplus k_i) \oplus (u^* \oplus k_i) \\ &= u \oplus u^* \\ &= \Delta u. \end{aligned}$$

Using this difference $\Delta u = \Delta y$ and the input difference Δx we turn to the difference distribution table, look up the number of possible output pairs, and retrieve the actual values (u_j, u_j^*) for each pair $j > 0$. Each pair yields one possible key value by calculating[1]:

$$k_i = y \oplus u_j.$$

This results in a small set of key guesses (for the AES S-Box it contains 2-4 guesses). To find the correct key, we have to repeat the procedure for more pairs with the same differences but different values. By collecting a large enough amount of such sets, the intersection of all sets should yield a single key guess that is common in all sets. This is the correct key, however, for this attack to work, we need non-zero input and output differences. Thus, S-Boxes with non-zero difference are called *active* S-Boxes.

3.3.3. Differential Trails

To use our method for retrieving key information on a standard cipher, we have to extend a differential to multiple rounds so that it spans `Nr-1` rounds. Thus, we get the input difference of the last round. Together with the actual output values we are able to retrieve the round key of the last round for all active S-Boxes.

A differential stretching over multiple rounds is called *differential characteristic* or *differential trail*. It is constructed by chaining multiple differentials. To accomplish this we have to find differentials $\Delta x_i \to \Delta y_i, i \geq 0$, such that $\Delta y_i = \Delta x_{i+1}$. Through this, we get a chain of differentials

$$\Delta x_0 \to \Delta x_1 \to \Delta x_2 \to \cdots \to \Delta x_{n-1} \to \Delta x_n,$$

where n is the number of rounds the differential trail should cover.

Each differential characteristic $\Delta x_0 \to \Delta x_n$ has a specific probability. This probability describes the likelihood that a randomly chosen plaintext difference Δx_0 yields the difference Δx_n after n rounds. A high probability indicates that we have to collect fewer plaintext-ciphertext pairs to find pairs matching the needed differences. Computing differential probability which depends on the number of right pairs is not easy. Since the input pairs of each differential are not independent, we are unable to exactly calculate the number of right pairs. Thus a reasonable approach is to approximate this probability [DR05].

Definition 3.6 (Approximate Differential Probability)**.** Assume the probabilities of each differential Δx_i in the differential trail $\Delta x_0 \to \Delta x_n$ to be independent. The approximate differential probability $Pr(\Delta x_0 \to \Delta x_n)$ [Sch11] is then calculated as:

$$\begin{aligned} Pr(\Delta x_0 \to \Delta x_n) &= Pr(\Delta x_0 \to \Delta x_1 \to \Delta x_2 \to \cdots \to \Delta x_{n-1} \to \Delta n_n) \\ &= Pr(\Delta x_0 \to \Delta x_1) \cdot Pr(\Delta x_1 \to \Delta x_2) \cdots Pr(\Delta x_{n-1} \to \Delta n_n) \\ &= \prod_{i=0}^{n-1} Pr(\Delta x_i \to \Delta x_{i+1}) \end{aligned}$$

[1] Since for each pair (a, a^*) the DDT yields also its reverse (a^*, a), it suffices to calculate only $y \oplus u_j$ and omit calculating $y \oplus u_j^*$.

If we choose a differential trail with high probability, we need a large enough set of plaintext pairs with the needed input difference in order to get enough pairs with the matching ciphertext difference of the trail. For example, if we assume a differential trail with probability 2^{-8}, we require a set containing at least 2^8 pairs with matching plaintext difference to get at least one pair that matches the full differential trail.

Similarly to linear cryptanalysis, the choice of the non-linear function has a large impact on the success of differential cryptanalysis. The goal is to design the S-Box such that the probability of the difference propagation is as low as possibly. Additionally, the number of active S-Boxes over multiple rounds should be as high as possible.

3.4. Truncated Differentials

In 1994 Knudsen published a new attack on block ciphers that uses truncated differentials [Knu94]. In its essence, this attack is a differential attack as presented above but modified such that ciphers which are secure against standard differential cryptanalysis could still be prone to truncated differential cryptanalysis.

In a standard differential cryptanalysis attack we look for differential trails that turn an input difference to some output difference with high probability. Truncated differential cryptanalysis differs, such that we only require a part of the output difference to conform to a differential trail. By ignoring part of the output difference, we are able to construct trails with larger probability since only a subset of the full state has to match a certain difference.

Countermeasures against truncated differentials are similar to those against standard differential cryptanalysis. Moreover, this form of attack is mostly efficient on ciphers that operate on aligned blocks of the state. AES is a good example for such a cipher, since all its operations are done at byte level rather than bit level. Nevertheless, AES is still resistant against this kind of attack as shown in [DR02].

3.5. Impossible Differentials

Impossible differentials were first noted in the initial paper on differential cryptanalysis by Biham and Shamir. The first applications as stand-alone attack were on IDEA [BBS99b] and Skipjack [BBD+98, BBS99a] by Biham, Biryukov and Shamir.

The basic idea is to use impossible difference transitions which are represented by entries with zero in the difference distribution table. These impossible transitions are used between two differential trails to filter key guesses made at a predefined round. The attacker first constructs a set of plaintext pairs that match a certain difference, and obtains the corresponding ciphertext pairs. Using those ciphertexts he then guesses some key bytes of the last round and partially decrypts the ciphertexts. With the obtained values of the intermediate state he than forms pairs that match a certain difference which results in an impossible differential at some state within the cipher. Together with the plaintexts, the attacker is then able to filter wrong key guesses since the impossible differential is likely to hold for correct guesses and unlikely to hold for wrong ones.

4. Security Properties of AES

AES has several aspects to ensure that extending short-cut attacks on round-reduced versions of AES to the full number of rounds makes them infeasible. Such attacks exploit certain properties of the cipher to be faster than exhaustive search. In Section 4.1, we cover the most important design aspects of AES that lead to its general security properties and security against standard cryptanalytic attacks. Then, in Section 4.2 we present the saturation attack, which is the most commonly known round-reduced attack on AES. This attack works on up to six rounds of AES but is infeasible for more than that.

4.1. Design Aspects of AES

As we have shown in the previous chapter, the properties of a cipher's non-linear transformations take an important role in withstanding linear and differential cryptanalysis. In AES this non-linear transformation is implemented as the AES S-Box which substitutes byte values. However, the S-Box is not the only important aspect of AES which makes it resistant against a wide variety of attacks. In the course of this section we introduce the most important techniques that ensure proper security.

4.1.1. AES S-Box

For resistance against linear and differential cryptanalysis, the AES S-Box was chosen to have minimal correlation between linear combinations of input bits and output bits. Additionally, the difference propagation probability was minimized to harden the cipher against differential cryptanalysis. For the AES S-Box this is achieved by the multiplicative inverse in $GF(2^8)$.

Another set of attacks that exploit properties of the S-Box are *algebraic attacks* [Bra09]. Those are effective on non-linear transformations with a simple algebraic expression. Since this is the case for the multiplicative inverse in $GF(2^8)$, the AES S-Box could be prone to such attacks. To achieve security against algebraic attacks, the S-Box also includes an affine transformation which is applied to the multiplicative inverse. This transformation does not influence the resistance against linear and differential cryptanalysis but makes the algebraic expression of the S-Box more complex and thus hardens the S-Box against this algebraic attacks.

When we analyze the DDT for the S-Box, we can see that 129/256 of the difference transitions are impossible, 126/256 of them have two possible pairs and for 1/256 of them there are four possible pairs. This is an important fact about the AES S-Box and is used in many considerations for attacks, especially attacks that use differentials in some way.

4.1.2. Proper Diffusion

Since the non-linear transformation operates at each byte of the state individually, it is important to mix these bytes properly. This results in high diffusion and thus, eliminates simple relations between plaintext and ciphertext bits. In the case of AES, the linear permutation operations `ShiftRows` and `MixColumns` are responsible for that. Both operations have their specific purpose: `MixColumns` mixes the bytes within each column and creates a dependency between them. `ShiftRows` shuffles the bytes in each row such that the dependencies within each column are spread out over the whole state. Together these operations attain full diffusion in just two rounds.

An important number that indicates the amount of mixing `MixColumns` provides is the *branch number*.

Definition 4.1 (Branch number, [DR02])**.** Let $P(v)$ be some linear transformation on a byte vector v and $W(v)$ be the number of non-zero bytes within such a vector. $W(v)$ is called *byte weight*. The branch number of $P(v)$ over all possible byte vectors $v \neq 0$ is defined as

$$\min_{v \neq 0}(W(v) + W(P(v)))$$

and describes the amount of diffusion introduced by the transformation.

The branch number of `MixColumns` is 5. Thus a linear relation between input and output bytes always involves at least *five different bytes*. A similar statement can be made for differentials: `MixColumns` transforms an input difference in only a single byte to a difference of four bytes of the same column. Or similarly, an input difference in two bytes leads to a difference in at least three bytes.

Since `MixColumns` operates on each column individually, the purpose of `ShiftRows` is to spread this diffusion over multiple columns. This is especially important since AES operates only at bytes and not individual bits and is thus potentially prone to truncated differential cryptanalysis. The rotation of each row by a different offset, as chosen for `ShiftRows`, is an optimal way to mix the bytes of different columns with each other and provide resistance against truncated differential attacks at the same time.

4.1.3. Number of Rounds

Another important factor for the security of a key iterating cipher is the number of times the round transformation is applied. Normally, it is a trade-off between security and performance. Increasing the number of rounds leads to higher security since it becomes harder to find differential trails with high enough probability or equally to construct linear characteristics. On the other hand, a lower number of rounds increases the performance. This is especially important if the cipher should also be usable on embedded systems or smart cards, as it is the case for AES.

For AES, Daemen and Rijmen chose the number of rounds that, in theory, it ensures a large enough security margin. This choice is mainly based on the best known attack at that time which was the *saturation attack* or *square attack* [DR02]. The saturation attack is a

short-cut attack on up to six rounds of AES. For AES-128, which has ten rounds in total, this gives four additional rounds of security margin. Since two full rounds for AES already provide full diffusion (a single bit of the state depends on all bits from two rounds before), these four rounds can be seen as adding full diffusion at the beginning and at the end. For larger key sizes the security margin is six and eight rounds, respectively. This higher security margin is needed because the block size stays the same while the key size increases. Therefore, the knowledge of part of the key influences more than one round. Additionally, as a larger key size results in more workload for an exhaustive key search, the workload for a shortcut attack should be higher too.

4.1.4. Key Schedule

The key schedule itself is also known to provide a target for attacks on block ciphers [KSW96, KS99]. AES counters this with a simple key schedule that provides enough diffusion within the round keys and is still fast on a wide range of processors. It uses the AES S-Box to introduce non-linearity to prohibit an attacker from deriving the full round key differences with the help of the cipher key differences. Further, AES has no known weak keys like DES [NIS81] and also no weak keys similar to IDEA [DGV93].

Since the round transformation of AES is exactly the same for each round, the cipher itself is fully symmetric. This symmetry could potentially lead to weaknesses of the algorithm. Therefore, the key schedule uses different round constants for each round key to break up this symmetry.

4.2. The Saturation Attack

The Saturation attack was initially developed by Knudsen for the block cipher *Square* and was part of the initial description of Square [DKR97]. Therefore it is known as *Square attack*. The attack itself exploits the byte-oriented structure of Square but is not bound to this specific cipher. Instead, it works on multiple other ciphers with a similar structure. Since the basic structure of AES is similar to that of Square, AES is also susceptible to this attack. However, the number of rounds of AES was chosen large enough to make this kind of attack infeasible. In the proposal for Rijndael, the authors analyze the security of AES and note that an attack on up to six rounds of AES is faster than exhaustive search [DR02].

The saturation attack on AES is a chosen plaintext attack which is based on a straight forward attack on four rounds. This basic attack can be extended to a practical attack on up to six rounds. The basic idea is to choose a set of plaintexts which are known to fulfill a certain property within three full rounds and use this property as a filter for key guesses. With the corresponding ciphertexts for the plaintexts, an attacker guesses certain bytes of the last round (rk_4) and partially decrypts the corresponding ciphertext bytes. By decrypting the same bytes of each ciphertext he can check the 3-round property and remove wrong key guesses with high probability.

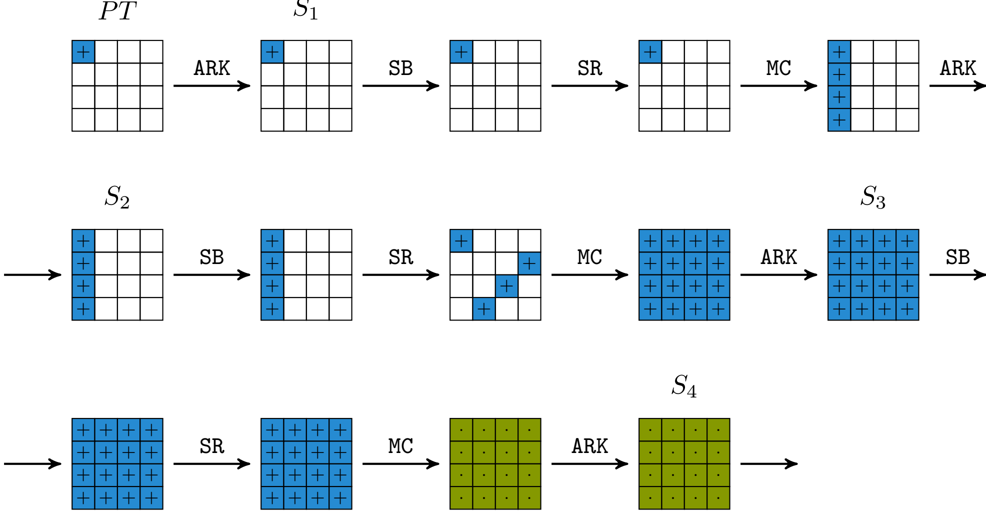


Figure 4.1.: Transformation of a Λ-set through three rounds of AES. ⊞ indicates active bytes of the Λ-set, ▪ indicates bytes for which only the balanced property holds and □ represents constant bytes.

4.2.1. Balanced Property

The heart of the Square attack is the 3-round *balanced property*. It requires the cipher's inputs to be chosen such that they form a so called Λ*-set*.

Definition 4.2 (Λ-set)**.** A Λ-set is formed by a set of 256 plaintexts that are equal in all bytes except a single byte. This byte is called *active* and varies over all possible 256 values.

Definition 4.3 (Balanced property)**.** Given an arbitrary set of n plaintexts, the balanced property for a byte $b_{i,j} = S^i[j]$ for some state S, over all plaintexts $1 \leq i \leq 256$ is defined as:

$$\bigoplus_{i=1}^{n} b_{i,j} = 0, \forall j,$$

where j defines the position in the state representation of the plaintext.

It is clear that the balanced property holds for all Λ-sets, since $x \oplus x = 0$ and $\bigoplus_{i=0}^{255} i = 0$. By tracing a Λ-set through the AES round transformation, we can see that `AddRoundKey` just creates another Λ-set since the same key is applied to all plaintexts. A similar statement can be made for `SubBytes`. Since it is a bijective function, it does not destroy the Λ-set as well. Moreover, as `ShiftRows` moves bytes, only the position of the active byte changes. `MixColumns` is the only operation that is able to destroy a Λ-set since it operates on four bytes and not on a single byte. As noted by Daemen and Rijmen in [DR02, p. 150], the output of `MixColumns` stays a Λ-set only in certain cases. For example, if the input had at most one active byte per column, this active byte is transformed to a column with all active bytes.

When we apply these observations to multiple rounds, we can observe the evolution of a Λ-set as shown in Figure 4.1. It is apparent from the explanation above that up to state $S_{3,\texttt{MC}}$ we still have a proper Λ-set. After applying the next `MixColumns` operation, the states do not form a Λ-set anymore. We denote the input of this `MixColumns` operation as $a_{i,j}$ and its output as $b_{i,j}$, then the balanced property holds for the output because $\forall j$:

$$\begin{aligned}
\bigoplus_i b_{i,j} &= \bigoplus_i \texttt{MixColumns}(a_{i,j}) \\
&= \bigoplus_i (\texttt{02} \cdot a_{i,j} \oplus \texttt{03} \cdot a_{i,j+1} \oplus \texttt{01} \cdot a_{i,j+2} \oplus \texttt{01} \cdot a_{i,j+3}) \\
&= \texttt{02} \cdot \bigoplus_i a_{i,j} \oplus \texttt{03} \cdot \bigoplus_i a_{i,j+1} \oplus \texttt{01} \cdot \bigoplus_i a_{i,j+2} \oplus \texttt{01} \cdot \bigoplus_i a_{i,j+3} \\
&= \texttt{02} \cdot 0 \oplus \texttt{03} \cdot 0 \oplus \texttt{01} \cdot 0 \oplus \texttt{01} \cdot 0 = 0
\end{aligned}$$

4.2.2. Basic Attack on 4 Rounds

First, we explain the basic attack on a round-reduced version of AES with four rounds (three full rounds plus the last round). We apply the balanced property on the first three rounds and use the last round to guess key bytes and filter them at the end of round 3 as shown in Figure 4.2. The whole attack goes as follows:

1. Choose a random plaintext and construct a Λ-set.
2. Retrieve the 256 ciphertexts by encrypting the plaintexts of the Λ-set.
3. We know that the balanced property holds at the end of round 3. Since the last round has no `MixColumns` operation, we have a one-to-one relation between each byte of ciphertext and the state at the end of round 3.
4. Choose an arbitrary byte from the ciphertext and guess the corresponding key byte of round key 4 (last round key).
5. For each key guess, partially decrypt this byte through the last round to the output state of round 3.
6. Verify the balanced property for this byte. For a correct guess, the balanced property must hold. For a wrong guess, the probability that it holds is 1/256 [DR02, p. 151].
7. Collect the guesses for which the balanced property holds.
8. Repeat the whole process for a different Λ-set to eliminate the false-positives from the set of key guesses.
9. Retrieve further key bytes by repeating the process until it is possible to find the remaining unknown key bytes using exhaustive search within a feasible amount of time.

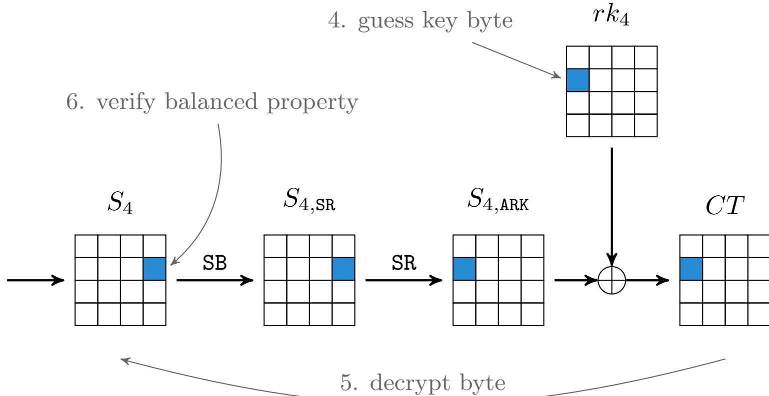


Figure 4.2.: The last round of the basic 4-round saturation attack on AES. The numbering corresponds to the attack description in Section 4.2.2.

4.2.3. Attacking 5 Rounds

The attack on four rounds can be extended to five rounds by adding one round at the end. As before, we apply the balanced property on the initial three rounds and guess key bytes on the remaining two rounds. To reach the output state of round 3, we now have to partially decrypt through the last *two rounds*. The idea is to partially decrypt the ciphertext through the last round, and then apply the same technique as for the attack on four rounds. To achieve this, we choose four bytes from the ciphertext such that they align in a single column after applying the inverse last round. By guessing the corresponding four bytes of the last round key (rk_5), we can partially decrypt these four state bytes.

To continue from that state, we would have to guess four bytes from rk_4 in order to apply the inverse of this round and reach round 3 to check the balanced property. Instead, we swap `MixColumns` and `AddRoundKey` in round 4 and apply `InvMixColumns` first to reduce the amount of key guesses and thus also the workload. This is possible as we described for the decryption process of AES in Section 2.5. The only consequence is that we have to guess bytes of the corresponding *equivalent* round key rk'_4, which is defined as:

$$rk'_i = \texttt{InvMixColumns}(rk_i).$$

The advantage is that at this stage we have the exact same situation as in the attack on four rounds. Since we already applied `InvMixColumns`, the remaining operations to revert for round 4 are equal to the last round. Thus, we only have to guess *one* byte of the equivalent round key. For each byte we verify at the end of round 3, we get a filter for four key bytes of the last round key and one of the equivalent round key 4. Figure 4.3 shows this process on the last two rounds.

4.2.4. Extension to 6 Rounds

To attack six rounds, we add an additional round at the beginning of the 5-round attack. This way, we use the balanced property starting from round 2 and apply the same attack

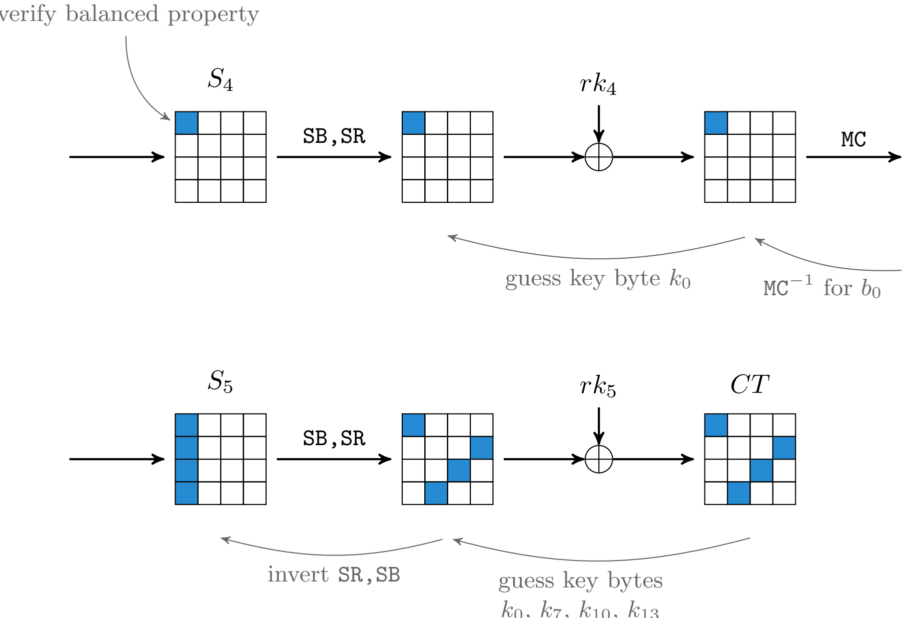


Figure 4.3.: The partial decryption and filtering stage of the saturation attack on the last two rounds.

on the last two rounds as before. To get a Λ-set at the beginning of round 2, we take a set of 2^{32} plaintexts instead of 2^8. These plaintexts are chosen such that four bytes vary over all possible 2^{32} values, and the remaining bytes are constant throughout all plaintexts.

The position of the four active bytes in the plaintext is selected so that all bytes are in the same column in state $S_{1,\mathtt{MC}}$. Instead of tracing a single Λ-set through the first round, we take all 2^{32} plaintexts since they form 2^{24} Λ-sets. As we have seen before, all operations except `MixColumns` keep the Λ-set intact. In this case, also `MixColumns` of round 1 does not destroy it either because all the active bytes align in one column and they take all possible values in the 2^{32} plaintexts.

Thus, we know that at state $S_{1,\mathtt{ARK}}$ we have 2^{24} Λ-sets but we do not know which plaintext belongs to which Λ-set. To select a single Λ-set, we would have to guess four bytes of the initial key and partially encrypt the active bytes to $S_{1,\mathtt{ARK}}$. Instead, we avoid this problem by taking all 2^{32} state values for the attack. This is possible because if the balanced property holds for one Λ-set, it also holds for a combination of them as should be evident from its definition. To filter wrong key guesses on the last two rounds, we just calculate the balanced property for all values and achieve the same effect.

The resulting final attack on six rounds needs 2^{32} chosen plaintexts and memory to store 2^{32} ciphertexts. Due to the fact that some wrong key guesses might survive the filtering, we have to repeat the attack on about four different sets of plaintexts to find the single right key [DR02, p. 151]. Each execution of this attack delivers five key bytes, thus the time complexity is calculated from 4 runs, 2^{32} plaintexts, 2^{40} key guesses, and 5 S-Box lookups

for partial decryption through the last two rounds. This equals to about 2^{72} 6-round-AES encryptions.

4.2.5. Improvement using Partial Sums

Ferguson et al. suggested an improvement on the 6-round attack using partial sums [FKL+00]. The basic attack stays the same as above, but the difference lies in the partial decryption of ciphertext bytes to verify the balanced property. When looking at this partial decryption in detail, we can omit `ShiftRows` since it is just a byte shuffle and does only influence the choice of bytes. We denote ciphertext bytes that are chosen for the attack by $c_{i,j}$, where i denotes the i-th ciphertext and $j, 0 \leq j \leq 3$ enumerates the four bytes we choose from it. The involved round key bytes k_i are numbered from 0 to 4, where k_0 to k_3 are from the last round key and k_4 is from round key rk_5. Then the verification of the balanced property for a single byte is represented by the equation:

$$\bigoplus_i \texttt{SB}^{-1}[T_0(c_{i,0} \oplus k_0) \oplus T_1(c_{i,1} \oplus k_1) \oplus T_2(c_{i,2} \oplus k_2) \oplus T_3(c_{i,3} \oplus k_3) \oplus k_4] \stackrel{?}{=} 0.$$

$T_j, 0 \leq j \leq 3$ denotes `InvSubBytes` followed by the multiplication with the corresponding field element of the `InvMixColumns` matrix. The $T_j(c_{i,j} \oplus k_j)$ partially decrypt the ciphertext bytes through the last round, and their `XOR` is the `InvMixColumns` for a single byte. k_4 is the single byte from the equivalent round key 5 we have to guess.

To improve this calculation and consequently reduce the workload of the attack, we reorganize the calculations by first constructing a triple $(T_0(c_{i,0} \oplus k_0) \oplus T_1(c_{i,1} \oplus k_1), c_{i,2}, c_{i,3})$ for all key guesses k_0 and k_1 and ciphertexts $1 \leq i \leq 256$. For convenience we define the partial sums x_l for $0 \leq l \leq 3$:

$$x_l = \bigoplus_{j=0}^{l} T_j(c_{i,j} \oplus k_j).$$

For $l > 0$ the partial sums can also be calculated recursively:

$$x_l = x_{l-1} \oplus T_l(c_{i,l} \oplus k_l).$$

From the values we get for the triple $(T_0(c_{i,0} \oplus k_0) \oplus T_1(c_{i,1} \oplus k_1), c_{i,2}, c_{i,3}) = (x_1, c_{i,2}, c_{i,3})$, we continue by calculating the tuple $(x_2, c_{i,3})$ for all possible key bytes k_2, and for each tuple we guess k_3 and calculate x_3. Finally, we verify the balanced property by guessing k_4 and calculate $\texttt{SB}^{-1}(x_3 \oplus k_4)$ for each x_3.

Instead of enumerating each triple, tuple and x_3, we only count their occurrences. Furthermore, if we encounter a triple, tuple or x_3 an odd number of times, we know that $x \oplus x = 0$ and thus, we can omit this value from the calculation since it has no influence on the result. Consequently, we count modulo 2 which requires only one bit for each value we count.

Despite this improvement, we still get false-positives for the key guesses and remove them by repeating the whole process about six times. To calculate the time complexity, we count the evaluations of the full equation from above. First, we guess two key bytes and calculate the triple $(x_1, c_{i,2}, c_{i,3})$ for all 2^{32} plaintexts. This yields at most 2^{24} values which we use

together with the 2^8 possible values for k_2 to compute $(x_2, c_{i,3})$. For x_3, this leaves 2^{16} possible values for the tuple and 2^8 possible values for k_3. Finally, we have at most 2^8 values for x_3 and 2^8 values for k_4 to check the full equation. This results in $(2^{32} \cdot 2^{16}) + (2^{24} \cdot 2^8) + (2^{16} \cdot 2^8) + (2^8 \cdot 2^8) \approx 2^{48}$ evaluations of the full equation. Since the equation contains five S-Boxes, and we have to repeat the process six times to eliminate all wrong key guesses, we get an equal of about 2^{53} S-Box applications. Using the rough estimate of 2^8 S-Box lookups for one AES encryption this yields a time complexity of about 2^{45}.

This improvement using partial sums allowed Ferguson et al. to extend the attack further to seven rounds by adding one more round at the end and guessing the full last round key. Naturally, this increases the workload drastically. Thus, this attack only applies to AES-192 and AES-256.

Part II.

Recent Attacks on AES

5. Introduction

This second part covers recent attacks on AES that form the foundation of this thesis. Generally, these attacks can be split into two categories: *single-key* attacks and *related-key* attacks. With a single-key attack, an adversary tries to find the secret encryption key of the system by using known or chosen ciphertexts and/or plaintexts to extract information on the key. In the related-key attack model, ciphertexts encrypted under multiple different keys are required. Those keys need to have a special relation to each other which has to be known or chosen by the adversary. In a chosen related-key attack she does not choose the value of the key directly, but influences the original encryption key such that it becomes a different key with the desired relation to the original one. An example where this is possible is a block cipher used in a hash function. Here, the key of the block cipher depends on the message which is hashed. The first appearances of related-key attacks were in [Knu92, Bih94].

Most recent attacks concentrate on the single-key attack scenario. Thus, we focused on three recent attacks on AES in the single-key attack model. The first is the multiset attack by Dunkelman et al., which is closely related to the saturation attack from Section 4.2. In Chapter 6 we give a detailed explanation of the multiset attack and both major techniques that were used to create it. Chapter 7 focuses on a different kind of attacks which aim for minimal data requirement. Such attacks alone do not threaten the security of any full AES variant, however, they provide a building block for more complex attacks like the 6-round known-plaintext attack on AES-128 presented in Section 7.3. The third recent publication we cover in this part is concerned with biclique attacks. Bicliques were originally used for cryptanalysis of hash functions, but have recently also been used for block cipher analysis by Bogdanov et al. In Chapter 8, we present the techniques used by Bogdanov et al. to create bicliques for block ciphers, specifically AES. Moreover, we also introduce their attacks on all full variants of AES and cover the biclique attack on 10-round AES-128 in full detail.

In addition to single-key attacks, there have also been two recent publications in the area of related-key attacks which are worth mentioning: The first related-key attacks on full AES-192 and full AES-256 were created by Biryukov and Khovratovich and published in [BK09]. These attacks are particularly interesting as they require only four related keys and have a time complexity of 2^{176} on AES-192 and $2^{99.5}$ on AES-256. The second recent related-key attack was introduced by Biryukov et al. and focused on round-reduced attacks for AES-256 with practical time complexity [BDK+10]. The best feasible attack they found works on 10-round AES-256 and has a time complexity of only 2^{45}.

A detailed listing of all attacks presented in this chapter, including the related-key attacks, and their complexities is shown in Table 5.1.

Attack	Kind	Rnds	Key	Time	Data	Memory	Reference	Section
LDC	KP	1	128	2^{32}	1	2^{24}	[BDD+10]	Sec. 7.2.2
	KP	2	128	2^{32}	2	1	[BDD+10]	Sec. 7.2.1
LDC	CP	2	128	2^{8}	2	2^{8}	[BDF12]	Sec. 7.2.3
	CP	3	128	2^{16}	2	2^{8}	[BDF12]	Sec. 7.2.3
LDC	KP	3	128	2^{40}	9	2^{31}	[BDD+10]	Sec. 7.2.4
LDC+diff.	KP	6	128	2^{120}	2^{109}	2^{88}	[BDD+10]	Sec. 7.3
Saturation	CP	6	all	2^{44}	$6 \cdot 2^{32}$	2^{32}	[FKL+00]	Sec. 4.2
Multiset	CP	7	all	2^{103+n}	2^{103+n}	2^{129-n}	[DKS10]	Sec. 6.5/6.6
Biclique	CC	10	128	$2^{126.15}$	2^{88}	2^{8}	[BKR11]	Sec. 8.4
	CP	10	128	$2^{127.34}$	2^{8}	2^{8}	[BKP+12]	Sec. 11.1
	CC	12	192	$2^{189.74}$	2^{80}	2^{8}	[BKR11]	
	CC	14	256	$2^{254.42}$	2^{40}	2^{8}	[BKR11]	
Related-key	CC	10	256	2^{45}	2^{44}	2^{33}	[BDK+10]	
	CP	12	192	2^{176}	2^{123}	2^{152}	[BK09]	
	CP	14	256	$2^{99.5}$	$2^{99.5}$	2^{77}	[BK09]	

Table 5.1.: A list of all recent attacks presented in this thesis. Additionally, three recent related-key attacks are also shown. CP, CC indicates *chosen* plaintext resp. ciphertext attacks and KP indicates *known* plaintext attacks.

6. The Multiset Attack

The attack by Dunkelman et al. [DKS10] represents the latest stage of multiple improvements on the square attack. It is based on the attack by Demirci and Selçuk from [DS08], which itself is an improvement of the attack by Gilbert and Minier from 2000 [GM00]. Due to its close relation to the original square attack, it is also a single-key attack using chosen plaintexts.

The general idea behind all these attacks is to find a distinguishing property or *distinguisher*. A good distinguisher allows us to differentiate a sequence of values, produced by encrypting a plaintext through a given number of AES rounds, from a random sequence. Such a sequence is constructed by encrypting multiple plaintexts and selecting the values a single ciphertext byte assumes. The balanced property itself is such a distinguisher. Given a sequence of 256 values, we can use it to verify if this sequence stems from encrypting a plaintext trough three rounds of AES or is just a random permutation. This is used in the square attack to verify key guesses made on the last two rounds. If the balanced property holds, it is very likely that the key guess is correct.

Since the theory of this attack is rather extensive, we first explain the 3-round distinguisher by Gilbert and Minier in Section 6.1, followed by the improved distinguisher by Demirci and Selçuk and their basic attack on the round-reduced variant of AES with seven rounds in Section 6.2. Afterwards, we continue with the new techniques added by Dunkelman et al., and explain their enhanced attack on seven rounds in full detail. We conclude our explanations of the multiset attack with some additional trade-offs which allowed Dunkelman et al. to further reduce the complexity of the attack.

6.1. The 3-Round Distinguisher by Gilbert and Minier

The basic idea of Gilbert and Minier is to take the balanced property and find a more performant way to distinguish a sequence of 256 values for which the balanced property holds from a sequence for which it does not hold. This is done by determining all possible sequences of values a byte can take when the balanced property holds. The distinguisher uses the following underlying observation on the balanced property:

Take a Λ-set and encrypt it through three full rounds. It is possible to introduce an order to the plaintexts of the set by sorting them according to the value of the active byte. For each byte of the output we thus get a well defined sequence of 256 values. For comparison, a random permutation of 256 bytes yields a total of $(2^8)^{256} = 2^{2048}$ possible sequences. Since we use a Λ-set, 15 bytes of the input (plaintext) are fixed and consequently equal. Thus, a Λ-set yields only $2^{256+15\cdot 8} = 2^{376}$ potential sequences. For AES, there are actually far fewer

possible sequences since it is possible to create a relation between plaintext, involved round keys and ciphertext.

In [GM00] Gilbert and Minier show that only nine bytes (2^{72} possible sequences) are necessary to uniquely define such a relation between the active byte of the Λ-set and one byte of the ciphertext. These nine bytes are not unique state or ciphertext bytes but rather a combination of them. Thus, we can enumerate these nine bytes and calculate the possible sequences, but we are not able to calculate the actual values of the involved key and state bytes. However, Gilbert and Minier used this observation to mount an attack on seven rounds for AES-192 and AES-256 with 2^{32} chosen plaintexts and a time complexity of 2^{140} encryptions. Additionally, they showed that this attack can be optimized for AES-128 to be marginally faster than exhaustive search.

6.2. 4-Round Distinguisher by Demirci and Selçuk

Demirci and Selçuk extended this distinguisher by one round. Their extension fully defines a sequence by just 25 bytes. This gives $2^{25 \cdot 8} = 2^{200}$ possible sequences for a single byte after decrypting a Λ-set through four rounds which is still less than the 2^{376} potentially possible sequences. Explaining the full process of how this distinguisher is constructed would exceed the scope of this thesis. Thus, we refer the reader to [DS08].

With this 4-round distinguisher they constructed a basic attack on seven rounds which provides the basis for the improvements by Dunkelman et al. described in the following sections. The basic attack is split into two phases: a precomputation phase and an online phase. In the precomputation phase we enumerate all 2^{200} possible sequences determined by the 25 bytes of the distinguisher and store them into a hash table. This enables us to take a random sequence and perform a fast lookup if it is contained in the table. This lookup determines if the sequence is a random permutation or stems from an AES encryption. The advantage of the precomputation phase is that it has to be performed only once per Λ-set. The online phase, on the other hand, has to be executed for every attack. The rough outline of the online phase is as follows:

1. Guess 4 bytes of first round key to construct a Λ-set starting from S_2.
2. Guess 1 key byte of rk_1 to sort elements of the Λ-set by the active byte.
3. Obtain the ciphertexts from the encryption oracle.
4. As in the square attack, guess the necessary 5 key bytes on the last two rounds to partially decrypt the ciphertext bytes.
5. This yields a sequence of 256 values for a decrypted byte in S_6.
6. If this sequence is in the hash table, keep the key guess, otherwise discard it.
7. To eliminate false-positives, repeat the process a few more times.

The time complexity of this attack is 2^{80} and requires only 2^{32} plaintexts. Unfortunately, the memory requirements for the hash table and the amount of time needed to prepare the table are so high (about 2^{200}) that it is slower than exhaustive search for AES-128 and AES-192. Therefore, Demirci and Selçuk introduced a trade-off between precomputation phase and online phase which makes the attack faster than exhaustive search for AES-192. For more details on this improvement we refer to [DS08].

6.3. Multiset Tabulation

For their improved attack on seven rounds Dunkelman et al. introduced two new techniques in [DKS10]. The *Multiset Tabulation* technique is the first one. The basic idea behind it is to modify the above 4-round distinguisher such that, instead of enumerating sequences of 256 bytes for a state byte, it represents the values using a multiset.

Definition 6.1 (Multiset)**.** A multiset [Knu98, p. 473] is a set of values where in addition, a counter is assigned to each value. This counter indicates how often an element was inserted into the set.

For the following explanation, we enumerate the states of those four rounds from S_1 through S_5, where S_1 is the plaintext and S_5 the resulting ciphertext after four full rounds. It is not necessary for this distinguisher to bring the plaintexts into a specific order as before. Thus, it suffices to just enumerate them arbitrarily from 0 to 255.

The goal of the 4-round distinguisher by Dunkelman et al. is to construct the difference vector

$$\mathbf{d}_{i,5} = (S_5^0[i] \oplus S_5^0[i], S_5^1[i] \oplus S_5^0[i], \ldots, S_5^{255}[i] \oplus S_5^0[i])$$

and collect the results in a multiset. This is done for each byte i of the ciphertext individually. In the remainder of this section, we outline how to construct this difference vector $\mathbf{d}_{i,5}$ from 24 parameters which are explicit key or state bytes.

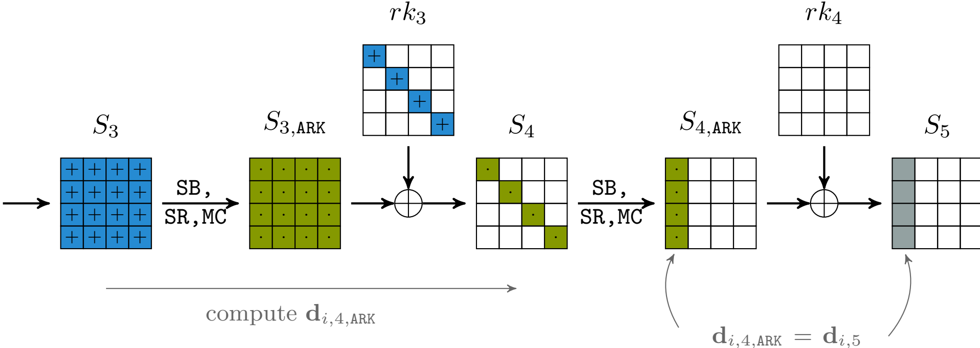


Figure 6.1.: First step in the multiset construction. [blue] indicates known bytes, [green] marks calculated values and [gray] marks calculated differences.

First, assume for each plaintext in the Λ-set that the full state S_3 is known. As shown in Figure 6.1, we can use these values to calculate $S_{3,\mathtt{ARK}}$ for each element of the Λ-set (indicated

by ■). By knowing four bytes (0, 5, 10 and 15) of the round key rk_3 (⊞), we can encrypt the four corresponding bytes of $S_{3,\texttt{ARK}}$ further. Thus, we get the values for the full first column of $S_{4,\texttt{ARK}}$. Using the values for these four bytes of $S_{4,\texttt{ARK}}$, we construct the difference vector

$$\mathbf{d}_{i,4,\texttt{ARK}} = (S^0_{4,\texttt{ARK}}[i] \oplus S^0_{4,\texttt{ARK}}[i], S^1_{4,\texttt{ARK}}[i] \oplus S^0_{4,\texttt{ARK}}[i], \ldots, S^{255}_{4,\texttt{ARK}}[i] \oplus S^0_{4,\texttt{ARK}}[i])$$

for $i \in \{0, 1, 2, 3\}$.

Since `AddRoundKey` is a linear operation and it does not change differences, this vector $\mathbf{d}_{i,4,\texttt{ARK}}$ is equal to $\mathbf{d}_{i,5}$ (indicated as ■ in Figure 6.1). This is exactly the difference vector we require for the construction of the multisets. Moreover, we can formulate a similar dependency for the other bytes ($i \in \{4, \ldots, 255\}$) of S_5 by choosing the appropriate bytes of rk_3.

Instead of knowing the full states S^l_3 for all $l \in \{0, \ldots, 255\}$, it suffices to know S^0_3 (S_3 for plaintext 0) and the difference vector

$$\mathbf{D}_3 = (S^0_3 \oplus S^0_3, S^1_3 \oplus S^0_3, \ldots, S^{255}_3 \oplus S^0_3)$$

since this vector enables us to construct the full states for all l from S^0_3. When we trace $\mathbf{D}_3$ backwards through the linear operations `AddRoundKey`, `MixColumns` and `ShiftRows` of round 2, it is clear we can also use the difference vector $\mathbf{D}_{2,\texttt{SR}}$ instead of $\mathbf{D}_3$. So far, to define the multiset we need to know:

- The full state S^0_3,
- four bytes of rk_3 and
- the differences $\mathbf{D}_{2,\texttt{SR}} = (S^0_{2,\texttt{SR}} \oplus S^0_{2,\texttt{SR}}, S^1_{2,\texttt{SR}} \oplus S^0_{2,\texttt{SR}}, \ldots, S^{255}_{2,\texttt{SR}} \oplus S^0_{2,\texttt{SR}})$.

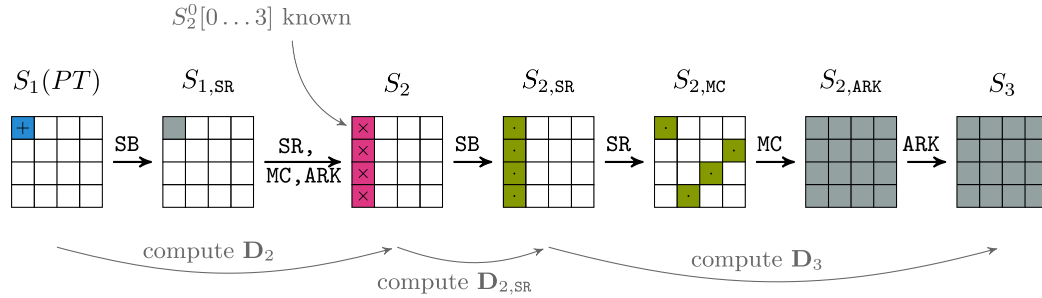


Figure 6.2.: Improved multiset construction by considering plaintexts. ⊠ marks bytes where we know the actual value for one plaintext only.

We can reduce these parameters further since we also know the 256 chosen plaintexts. Using the plaintexts we calculate the difference vector $\mathbf{D}_{1,\texttt{SR}}$. This is possible because after `SubBytes` all constant bytes remain equal and the active byte takes all possible values before and after `SubBytes`. Again, as shown in Figure 6.2, the remaining operations in the current round are all linear. Consequently, we also know the differences $\mathbf{D}_2$. Note that all four active

bytes in state S_2 are in the first column. Equally to the balanced property, all other columns contain constants (see Figure 4.1).

By knowing the value of the first column of S_2^0 (☒ in Figure 6.2), we are able to use the differences $\mathbf{D}_2$ to calculate the first column of S_2^l. Hence, we can now also compute the first column of $S_{2,\mathtt{SR}}^l$ for $l \in \{0, \ldots, 255\}$. Since we know the actual values of those four bytes, we are also able to calculate the differences $\mathbf{D}_{2,\mathtt{SR}}$. As the other bytes in columns 1 to 3 of S_2 and $S_{2,\mathtt{SR}}$ are constant and thus equal for all $l = \{0, \ldots, 255\}$, the differences in these bytes are also zero because

$$S_{2,\mathtt{SR}}^l[i] \oplus S_{2,\mathtt{SR}}^0[i] = 0.$$

As a result, the knowledge of bytes 0, 1, 2, 3 of state S_2^0 and the 256 chosen plaintexts enable us to calculate the full vector of differences $\mathbf{D}_{2,\mathtt{SR}}$ for *all* bytes. In the end, this leaves only 24 parameters to determine a multiset:

- The full state S_3^0,
- four bytes of rk_3 and
- four bytes of S_2^0 (bytes 0, 1, 2, 3 if active byte of Λ-set is byte 0).

This leaves 2^{192} possible multisets for these 24 parameters, compared to theoretically $\binom{510}{256} \approx 2^{505.2}$ possible multisets[1].

The advantage of this distinguisher over that presented by Demirci and Selçuk is that we do not have to sort the plaintexts of the Λ-set. Thus, it is not necessary to guess one byte of rk_1 in the attack to sort the plaintexts. Further, since there are 24 parameters, there are only 2^{192} possible multisets. For the distinguisher by Demirci and Selçuk there were 25 parameters and thus 2^{200} possible sequences. Furthermore, for the next improvement, it is also advantageous that the parameters are explicit state or key bytes. For the previous distinguisher by Gilbert and Minier, and also that by Demirci and Selçuk, this was not possible since some parameters depended on a combination of key and state bytes. Hence, we could only enumerate the parameters, but could not calculate the values of actual state or key bytes.

6.4. Differential Enumeration

To reduce the amount of possible multisets further, Dunkelman et al. introduced a technique called *differential enumeration*. Its goal is to fix as many of the 24 parameters as possible to previously known values. Some of those parameters are key bytes, which clearly cannot be fixed since their value depends on the unknown encryption key. However, the remaining parameters are state bytes, and it is possible to reduce the amount of possible values for them. Specifically, it is possible to reduce the 16 parameters of S_3^0 by a huge factor, as we describe in the remainder of this section.

[1] For a thorough explanation of the probability distribution of multisets see [DKS10].

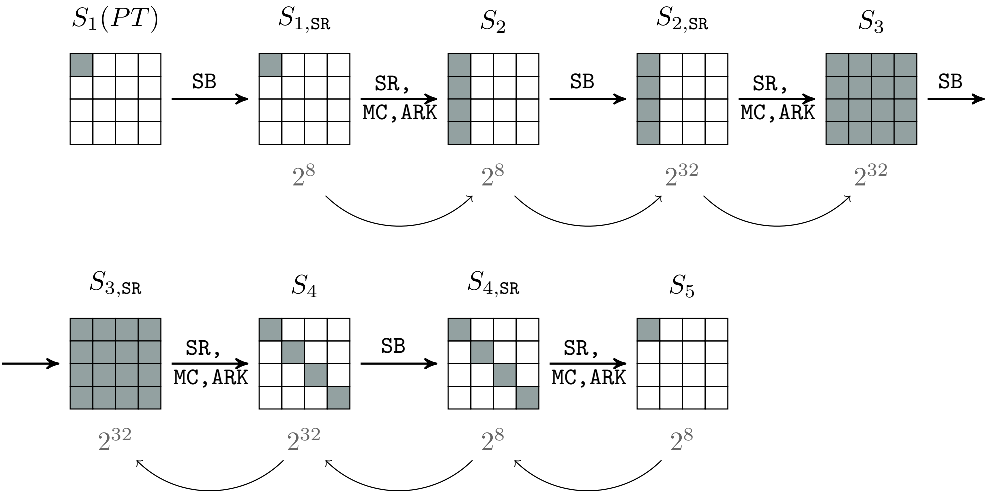


Figure 6.3.: The maximum possible amount of differences per state of the 4-round expected-probability differential.

To fix some parameters that define a multiset to known values, Dunkelman et al. use an *expected-probability* differential. This is basically a truncated differential trail with neither very high nor very low probability. The truncated differential chosen for this attack is depicted in Figure 6.3. It covers four full rounds of AES, and its input state and output state have only one active byte each. For our explanation of the multiset attack, we chose this active byte to be byte 0 of input and output state. However, it is also possible to choose different bytes as long as certain criteria are met to achieve the same effect.

For the input state of the differential trail (S_1 in Figure 6.3), we take all 2^8 possible differences for the single active byte. `MixColumns` of the first round spreads this differences to all four bytes of column 0. Nevertheless, there are still 2^8 possible differences in state S_2. Only, these differences are spread in some unknown way over the four active bytes of the first column. Since `SubBytes` of round 2 is a non-linear operation, and the differences in S_2 are unknown, we assume all 2^{32} possible differences for these four bytes in $S_{2,\mathtt{SR}}$. The remaining linear operations in this round (up to S_3) spread the differences to all 16 bytes such that all of them are active. Despite 16 active bytes, the amount of possible differences stays the same at 2^{32}. Note that these differences are completely independent from any round keys.

We perform a similar analysis from the end of those four rounds backwards. Beginning, in state S_5, we again take all 2^8 possible differences for the single active byte. The amount of possible differences remains unchanged until the output of `SubBytes` ($S_{4,\mathtt{SR}}$ in Figure 6.3). However, there are four active bytes at this state. For the same reason as before, to move backwards through `SubBytes`, we have to assume all possible differences for these four bytes. Thus, in state S_4 we take all 2^{32} possible differences for the active bytes 0, 5, 10 and 15. Up to state $S_{3,\mathtt{SR}}$, this amount stays the same, however, all bytes of this state are active due to `MixColumns` in this round.

At this point, we know that in states S_3 and $S_{3,\mathtt{SR}}$ there are 2^{32} possible differences each. Additionally, all bytes are active and the only operation left between those states is `SubBytes`.

By knowing input and output differences of the `SubBytes` operation, it is possible to deduce the actual values using the DDT of the AES S-Box. On average, there is only one possible pair of actual values for a random input and output difference (see also Chapter 4). In our case, we have 2^{32} differences for the input state and 2^{32} (most likely different) differences at the output of `SubBytes`. Consequently, there are only a total of $2^{32} \cdot 2^{32} = 2^{64}$ pairs of actual values for the input state. The same holds for the output state of `SubBytes` since knowing the actual values of the input state enables us to calculate the values of the output state.

Now, we use this differential to reduce the amount of possible multisets by overlaying the truncated differential on the 4-round distinguisher. Since we require the value for S_3^0 to construct the multisets from above, we can use this differential trail to reduce the amount of possible values for S_3^0 from 2^{128} to 2^{64}. By reducing the amount of possible values for S_3^0, we also reduce the overall amount of possible multisets for the distinguisher. To do this, we have to select plaintext 0 of the Λ-set from the right pair with respect to this truncated differential. If this is the case, then S_3^0 (state S_3 for plaintext 0) assumes at most 2^{64} possible values. As a right pair consists of two actual values for the plaintext, this leaves two choices for plaintext 0.

To select plaintext 0, we assume that four rounds of AES behave like a random permutation (which is commonly assumed). Hence, the probability of the above differential trail is 2^{-120}, which is the same as getting 120 bits to be equal. In essence, this means that within 2^{120} randomly chosen pairs of plaintexts, there is one pair that is right w.r.t. the differential. Furthermore, by using Λ-sets, each Λ-set already supports the construction of 2^{15} pairs with the correct input difference (only one active byte) to the differential. Consequently, it suffices to take only 2^{105} Λ-sets with byte 0 as active byte. From these 2^{105} Λ-sets there is then one pair which is a right pair w.r.t. the differential trail from Figure 6.3.

Compared to the attack by Demirci and Selçuk, this reduction of possible multisets results in a huge decrease of time and memory complexity for the precomputation phase. Since the differential trail is independent of any key bytes, the precomputation phase is independent of any key bytes too. Thus, we have to perform this phase only once. On the other hand, the data complexity of the whole attack is increased since this technique requires $2^{105} \cdot 2^8 = 2^{113}$ chosen plaintexts. Overall, the differential enumeration technique reduces the amount of possible multisets to $2^{192} \cdot 2^{-64} = 2^{128}$.

Summarizing the enhancements by Dunkelman et al., we can make the following statements:

- Multiset tabulation replaces explicit byte sequences for S_5 with multisets of difference vectors $\mathbf{d}_{i,5} = (S_5^0[i] \oplus S_5^0[i], S_5^1[i] \oplus S_5^0[i], \ldots, S_5^{255}[i] \oplus S_5^0[i])$.
- A multiset is fully determined by only 24 bytes.
- Differential enumeration reduces all possible values for S_3^0. Consequently, we get only 2^{128} possible multisets.
- For these improvements to work, plaintext 0 has to be chosen from the right pair for the differential.

- The data complexity is increased, but since it was low before, this provides a good trade-off for the following attack.

6.5. The Multiset Attack on 7-Round AES

The basic attack on seven rounds works for all variants of AES and is faster than exhaustive search for all of them. The round-reduced variant of AES used for this attack is comprised of six full rounds plus the last round. Further, this AES variant also includes the initial `AddRoundKey` operation with the whitening key before the first round.

As mentioned before, the attack is basically an improvement of the Demirci and Selçuk attack and thus, the 4-round distinguisher is also placed from S_2 to S_6. In addition to the distinguisher, we also place the truncated differential trail from S_2 to S_6. Similar to the Demirci and Selçuk attack, we split the attack into two phases: a precomputation phase and an online phase. The precomputation phase is straightforward because we simply calculate all 2^{128} multisets. Then, we store all these multisets in a hash table. The online phase is more complex, thus, we split it into two stages. The first stage finds the right pair w.r.t. the differential. The second phase constructs a Λ-set and performs similar key guessing operations as in the Demirci and Selçuk attack.

To find the right pair in the online phase, we require a large enough amount of plaintexts since on average there is only one right pair within 2^{120} randomly chosen plaintexts. As explained before, by choosing these plaintexts carefully we can lower their required amount. We achieve this by forming groups of 2^{32} plaintexts which differ only in four bytes. Those active bytes are byte 0, 5, 10 and 15 and they have to assume all possible values. The right pair is then found as follows:

1. Take 2^{81} groups of 2^{32} plaintexts as just described.

2. Encrypt all plaintexts under the unknown encryption key, and store the resulting 2^{113} ciphertexts in a hash table. Thus we get 2^{81} groups of 2^{32} plaintext-ciphertext pairs.

3. Using the ciphertexts of each group, form pairs that have a non-zero ciphertext difference only in bytes 0, 7, 10 and 13. All other differences have to be zero. For each plaintext-ciphertext group, there are $\binom{2^{32}}{2} = 2^{63}$ pairs with the correct plaintext difference for the truncated differential. Thus, there are $2^{81} \cdot 2^{63} = 2^{144}$ differential pairs for all groups. Since the differential pairs require 12 bytes or 96 bit of the ciphertexts to be equal, this removes 2^{96} pairs. Consequently, this leaves $2^{144} \cdot 2^{-96} = 2^{48}$ differential pairs with correct plaintext and ciphertext difference w.r.t. the truncated differential.

4. Next, guess key bytes 0, 5, 10 and 15 of rk_0 and partially encrypt the corresponding plaintext bytes up to $S_{1,\texttt{ARK}}$ for each key guess. This yields the values for column 0 of $S_{1,\texttt{ARK}}$. Consequently, we are also able to calculate the differences of the corresponding four bytes in S_2.

5. For each key guess, keep only pairs with non-zero difference in $S_2[0]$ and zero difference in $S_2[1]$, $S_2[2]$ and $S_2[3]$. This is a 24-bit filter, thus, only $2^{48} \cdot 2^{-24} = 2^{24}$ pairs *per key guess* are expected to remain.

6. Guess bytes 0, 7, 10 and 13 of rk_7, partially decrypt the corresponding ciphertext bytes, and select only pairs with non-zero difference in $S_6[0]$ and zero difference in all other bytes. This leaves about one pair per key guess which is the right pair w.r.t. the differential and the guessed key bytes.

At the end of this stage, we get 2^{64} possible "right pairs" since we made $2^{32} \cdot 2^{32} = 2^{64}$ key guesses and filtered all pairs to about one right pair per guess. For each of these pairs, we continue by choosing one element of the pair as plaintext 0. Together with the previously guessed bytes of rk_0, we can now create a Λ-set in state $S_{1,\texttt{ARK}}$ with byte 0 as active byte for each one of these possible "right pairs". Note that due to the key guesses, we only know the values for bytes 0 through 3 of $S_{1,\texttt{ARK}}$. To construct a Λ-set, we `XOR` byte 0 of $S_{1,\texttt{ARK}}$ with the values from 1 to 255. Further, we decrypt column 0 of $S_{1,\texttt{ARK}}$ up to the plaintext. We do this for each possible "right pair" to get one Λ-set per key guess. For each Λ-set, we guess byte 0 of the equivalent round key rk'_6, and use the already guessed key bytes of rk_7 to partially decrypt the corresponding ciphertext bytes up to byte 0 of S_6. With the actual value for this byte 0, we can compute the multiset for the difference vector

$$\mathbf{d}_{0,5} = (S_5^0[0] \oplus S_5^0[0], S_5^1[0] \oplus S_5^0[0], \ldots, S_5^{255}[0] \oplus S_5^0[0]).$$

To verify a key guess, we simply check if the obtained multiset is stored in the precomputed hash table. If it exists, we keep the key guess, otherwise we discard it. When we have only one key guess left, we calculate the remaining key bytes using other methods such as exhaustive search.

6.5.1. Performance of the Attack

The precomputation phase defines the overall memory complexity of the attack since all the computed multisets have to be stored in a hash table. Since a multiset can be represented by 512 bits, the memory complexity is 2^{130} 128-bit blocks. Calculating a single entry of the multiset is similar to about four full rounds. Since a multiset contains 256 elements, the time complexity is equivalent to $2^{128} \cdot 256 \cdot 2^{-3} = 2^{133}$ 7-round encryptions.

Calculating the time complexity of the online phase is rather simple: It is clear from the description of the attack above that the most time-consuming part is encrypting 2^{113} plaintexts. All other parts are far below this boundary, thus, the overall time complexity is about 2^{113} 7-round encryptions. The data complexity is also dominated by the 2^{113} chosen plaintexts.

6.6. Further Improvements and Trade-offs

To fine-tune their attack, Dunkelman et al. presented three trade-offs which allow to improve the complexities of the basic 7-round attack. These improvements mainly target the data

complexity because it is rather high. Moreover, the time complexity is solely determined by the amount of data to encrypt, thus, reducing the amount of data also improves the time complexity.

The first improvement reduces data and time complexity by 2^8 without changing the memory complexity. We achieve this by modifying the expected-probability differential to allow an additional byte with non-zero difference at the input state. However, this additional byte can only be one of bytes 5, 10 or 15, because the additional non-zero difference must not change the 2^{32} possible differences at S_3. The effect of this additional active byte is that we need fewer plaintexts. Thus, the data complexity is reduced to 2^{105}. However, we have to make additional key guesses (four bytes of rk_1) to filter these pairs with modified input difference. Nevertheless, the overall time complexity still decreases, because all other operations of the online phase have substantially lower time complexity compared to encrypting 2^{113} plaintexts. Consequently, the time complexity of the online phase is also 2^{105}.

The second improvement builds on top of the previous improvement in the sense that the active bytes of the truncated differential can be moved within a state without changing the amount of possible differences. Despite that, it is not possible to use all 256 possible combinations for one active input and output byte, because the active byte of the input state has to be the same as the active byte of the Λ-set. Hence, precomputations for multiple Λ-sets would be necessary. Nevertheless, a slight improvement can be achieved in combination with the additional non-zero byte of the previous improvement. Dunkelman et al. found out that it is possible to use up to five differentials in parallel, when the additional active byte for the input difference of each differential is one of 5, 10, 15. For the output difference, the active byte can be one of 1, 2, 3. Consequently, data and time complexities are both reduced by a factor of 5 to about 2^{103}.

The third improvement is merely a minor one and reduces the memory complexity. This can be accomplished by computing only part of the hash table in the precomputation phase. In online phase, this is compensated by running the attack multiple times on different Λ-sets. This represents a trade-off by a factor of 2^n. Hence, data and time complexities become 2^{103+n} and the memory complexity becomes 2^{129-n}. All complexities are equal at $n = 13$, which is 2^{116}.

Dunkelman et al. introduced further improvements to the basic 7-round attack which allowed them to extend the attack by one round. This improved attack applies only to 8-rounds AES-192 and AES-256. In addition to the techniques from above, this attack uses certain observations on the key schedule. However, as this would exceed the scope of this thesis, we refer the interested reader to [DKS10].

7. Low Data Complexity Attacks

One common problem of attacks is that they require a rather high amount of data. This is also true for the attacks we have covered so far. The Saturation attack requires 2^{32} plaintext-ciphertext pairs for 6-rounds AES-128, and the Multiset attack from above requires as many as 2^{113} plaintext-ciphertext pairs. In this section, we take a closer look at a different approach for attacks on AES which demand only a minimal amount of data. For this purpose, we present a number of attacks by Bouillaguet et al. which they call *low data complexity* (LDC) attacks. The content of this chapter is mainly based on [BDD$^+$10] by Bouillaguet et al. and the closely related paper [BDF12] by Bouillaguet, Derbez and Fouque from 2012 where the authors describe the tools used to automatically find new attacks with minimal data complexity.

In general, finding such attacks is hard because most ciphers and their key schedule need specific properties that can be exploited with a minimal amount of available data. Since most ciphers like AES are designed to have high diffusion over their full number of rounds, LDC attacks can be found for only a few rounds. At first, this would seem useless because this does not pose a threat to a cipher's security. Nevertheless, there are good reasons to consider these kinds of attacks: First of all, such attacks are more closely related to practical applications, because attaining large amounts of plaintext-ciphertext pairs often takes a lot of time and storage and makes the attack less practical. Furthermore, an attack on a few rounds with minimal data complexity provides a good building block for more complex attacks on more rounds. A list of concrete examples of how such attacks can be used is given in [BDD$^+$10]. The most important ones include:

Slide attacks: The goal of these kinds of attacks is to reduce the attack on the whole cipher to an attack on a single round with only 2 known plaintext-ciphertext pairs. The idea for slide attacks was first described by Wagner and Biryukov in [BW99] and for example used to attack the cipher Keeloq [CBW08].

Side channel attacks: For these kind of attacks, an attacker mostly has access to some internal state information and the amount of data is also often very low. Hence in such cases low data complexity attacks aid these kinds of attacks.

Building Blocks: LDC attacks can be used to construct more complex attacks. For instance, one attack in [BDD$^+$10] uses a 2-round low data complexity attack to construct a known-plaintext attack on six rounds of AES. At the end of this section we will cover this attack in detail.

Attacks on similar ciphers: Certain ciphers share the same basic operations. For example, there are other ciphers and hash functions which use similar basic operations as AES.

Low data complexity attacks can also provide the means to create new attacks on them.

As mentioned before, finding LDC attacks on many rounds is hard and these attacks are mostly infeasible. This is also true for AES, thus, Bouillaguet et al. concentrated on round-reduced versions of AES-128 from one to four rounds. Both papers considered here contain multiple different attacks ranging from one known plaintext to ten chosen plaintexts. All these attacks are based on two main concepts: differential properties and a meet-in-the-middle approach.

Differential properties use known differences before and after `SubBytes`, and the DDT to retrieve the actual values of those bytes. The idea of the meet-in-the-middle approach is to partially encrypt certain plaintext bytes and decrypt certain ciphertext bytes towards some intermediate state bytes. This is done by guessing a minimal amount of key bytes and using properties of the key schedule and the basic round operations. Since the key schedule is different for all three variants of AES, these attacks work only for AES-128 and are not easily adaptable for the other variants of AES. For completeness, a full list of the best attacks for each number of rounds and attack model (known or chosen plaintext) is given in Table 7.1. It can be seen from this table that it is possible to get fairly fast attacks with a minimal amount of chosen or even known plaintexts for up to four rounds.

Rounds	**Attack Model**	**Complexity**			**Reference**
		Data	Time	Memory	
1	known plaintext	1	2^{32}	2^{16}	[BDF12]
	known plaintext	2	2^{12}	1	[BDD$^+$10]
2	known plaintext	1	2^{64}	2^{48}	[BDF12]
	known plaintext	2	2^{32}	2^{24}	[BDF12]
	chosen plaintext	2	2^{8}	2^{8}	[BDF12], Sec. 7.2.3
3	known plaintext	1	2^{96}	2^{72}	[BDF12]
	known plaintext	9	2^{40}	2^{31}	[BDD$^+$10], Sec. 7.2.4
	chosen plaintext	2	2^{16}	2^{8}	[BDF12], Sec. 7.2.3
4	chosen plaintext	2	2^{80}	2^{80}	[BDF12]
	chosen plaintext	4	2^{32}	2^{24}	[BDF12]

Table 7.1.: A list of best low data complexity attacks from [BDD$^+$10] and [BDF12].

The remainder of this section is divided into three parts: In Section 7.1, we introduce the basic properties of AES used by these attacks to achieve low data complexity. Section 7.2 covers a selection of LDC attacks ranging from one to three rounds of AES-128. One of these attacks was used to construct a known-plaintext attack on six rounds of AES which we cover in Section 7.3.

7.1. Theoretical Basics

The following five basic properties of AES are used to achieve a low data complexity:

Property 1[1]: The amount of possible actual values for a given input and output difference to the AES S-Box is as follows:

- For 129/256 difference pairs, the transition is impossible.
- For 126/256 pairs, there are two possible actual values for each element of the pair.
- For 1/256 of the pairs, there exist four values.

Property 2: The `MixColumns` operation works on each column of the AES state. Thus, its input and output is four bytes each. The knowledge of four of these eight bytes enables the calculation of the other four values. Since `MixColumns` is a linear operation, the same holds if four *differences* are known.

Property 3: This property covers five important relations of the AES-128 key schedule. For each column of five consecutive round keys we are able to construct the relations given in Table 7.2. These equations were used by Bouillaguet et al. to derive the five relations between a few key bytes over three to five rounds shown in Figure 7.1. The corresponding equations are:

1. $rk_{i+2}[j] \oplus rk_{i+2}[j+8] = rk_i[j+8]$
2. $rk_{i+2}[j+4] \oplus rk_{i+2}[j+12] = rk_i[j+12]$
3. $rk_{i+2}[j+4] \oplus v_{i+1}[j] = rk_i[j+4]$
4. $rk_{i+4}[j+12] \oplus v_{i+3}[j] = rk_i[j+12]$
5. $rk_i[j+8] \oplus rk_i[j+12] \oplus v_{i+2}[j] = rk_{i+3}[j+12]$

The above relations are valid for $0 \leq j \leq 3$, which basically means that each equation holds for all bytes of the respective column.

Property 4: For rounds $i > 1$, Bouillaguet et al. found three relations between columns of consecutive states. The first relation can be constructed by starting with

$$S_{i-1,\texttt{MC}}[4 \dots 7] = \texttt{InvMC}(S_{i-1,\texttt{ARK}}[4 \dots 7]),$$

where `InvMC` denotes the inverse `MixColumns` operation. By knowing the four corresponding bytes of rk_{i-1}, we can replace the right side with bytes from S_i and the round key:

$$S_{i-1,\texttt{MC}}[4 \dots 7] = \texttt{InvMC}(rk_i[4 \dots 7] \oplus S_i[4 \dots 7]).$$

Further, by using the linearity of the key schedule, $rk_i[4 \dots 7]$ can be represented by eight bytes of rk_{i+1}:

[1] We covered this property before, but for completeness we mention it here again.

Rnd.	Column 0	Column 1	Column 2	Column 3
i	$rk_{i,0}$	$rk_{i,1}$	$rk_{i,2}$	$rk_{i,3}$
$i+1$	$rk_{i,0} \oplus v_i$	$rk_{i,0} \oplus rk_{i,1} \oplus v_i$	$rk_{i,0} \oplus rk_{i,1} \oplus rk_{i,2}$ $\oplus v_i$	$rk_{i,0} \oplus rk_{i,1} \oplus rk_{i,2}$ $\oplus rk_{i,3} \oplus v_i$
$i+2$	$rk_{i,0} \oplus v_i \oplus v_{i+1}$	$rk_{i,1} \oplus v_{i+1}$	$rk_{i,0} \oplus rk_{i,2} \oplus v_i$ $\oplus v_{i+1}$	$rk_{i,1} \oplus rk_{i,3} \oplus v_{i+1}$
$i+3$	$rk_{i,0} \oplus v_i \oplus v_{i+1}$ $\oplus v_{i+2}$	$rk_{i,0} \oplus rk_{i,1} \oplus v_i$ $\oplus v_{i+2}$	$rk_{i,1} \oplus rk_{i,2} \oplus v_{i+1}$ $\oplus v_{i+2}$	$rk_{i,2} \oplus rk_{i,3} \oplus v_{i+2}$
$i+4$	$rk_{i,0} \oplus v_i \oplus v_{i+1}$ $\oplus v_{i+2} \oplus v_{i+3}$	$rk_{i,1} \oplus v_{i+1} \oplus v_{i+3}$	$rk_{i,2} \oplus v_{i+2} \oplus v_{i+3}$	$rk_{i,3} \oplus v_{i+3}$

Table 7.2.: Relations between columns of five consecutive round keys.

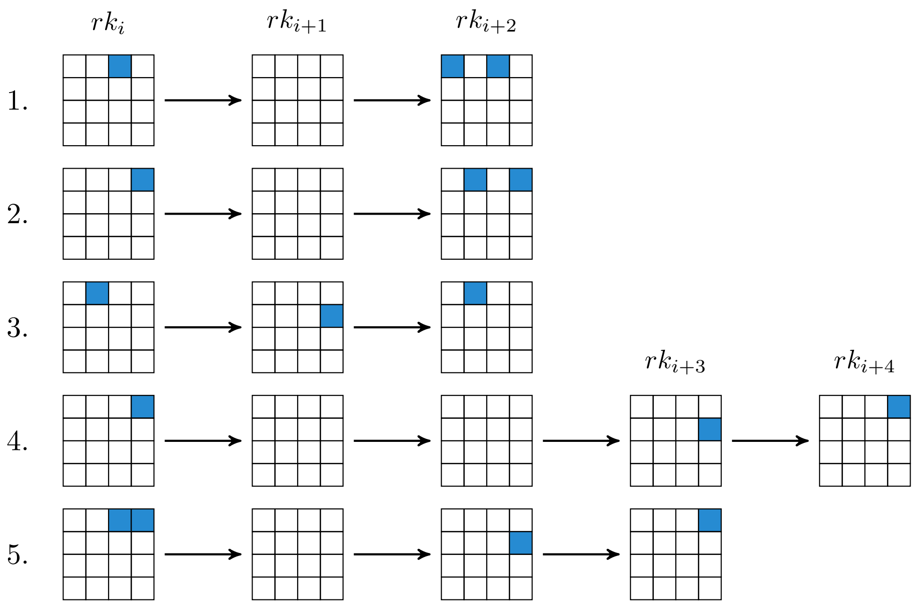


Figure 7.1.: Visualization of the five key schedule relations for the first byte of the corresponding column of rk_i.

$$S_{i-1,\texttt{MC}}[4\dots7] = \texttt{InvMC}(rk_{i+1}[0\dots3] \oplus rk_{i+1}[4\dots7] \oplus S_i[4\dots7]).$$

Since the round keys are added using `XOR`, it is possible to express them as $S_{i,\texttt{ARK}} \oplus S_{i+1}$. Thus, we get:

$$S_{i-1,\texttt{MC}}[4\dots7] = \texttt{InvMC}(S_{i,\texttt{ARK}}[0\dots3] \oplus S_{i+1}[0\dots3] \oplus S_{i,\texttt{ARK}}[4\dots7] \oplus S_{i+1}[4\dots7] \oplus S_i[4\dots7]).$$

The last step is to apply the inverse `MixColumns` operations where possible. Thus, we get:

$$S_{i-1,\texttt{MC}}[4\dots7] = S_{i,\texttt{MC}}[0\dots3] \oplus S_{i,\texttt{MC}}[4\dots7] \oplus \texttt{InvMC}(S_{i+1}[0\dots3] \oplus S_{i+1}[4\dots7] \oplus S_i[4\dots7]).$$

Similar equations can be constructed for columns 2 and 3. Hence, this yields three equations:

1. $S_{i-1,\texttt{MC}}[4\dots7] \oplus S_{i,\texttt{MC}}[0\dots3] \oplus S_{i,\texttt{MC}}[4\dots7] = \texttt{InvMC}(S_i[4\dots7] \oplus S_{i+1}[0\dots3] \oplus S_{i+1}[4\dots7])$
2. $S_{i-1,\texttt{MC}}[8\dots11] \oplus S_{i,\texttt{MC}}[4\dots7] \oplus S_{i,\texttt{MC}}[8\dots11] = \texttt{InvMC}(S_i[8\dots11] \oplus S_{i+1}[4\dots7] \oplus S_{i+1}[8\dots11])$
3. $S_{i-1,\texttt{MC}}[12\dots15] \oplus S_{i,\texttt{MC}}[8\dots11] \oplus S_{i,\texttt{MC}}[12\dots15] = \texttt{InvMC}(S_i[12\dots15] \oplus S_{i+1}[8\dots11] \oplus S_{i+1}[12\dots15])$

Property 5: Knowing the input and output state of one full round and one column of the round key ($rk_{1,0}$), we can retrieve $rk_0[7]$ and $rk_0[8]$ by a table lookup. Basically, there are at most twelve possible values for each value of $rk_{1,0}$ and on average actually only one. The time required to generate this table is 2^{32}, and the table itself has a size of 2^{24} 128-bit blocks. For details on how to construct this table as well as a thorough proof, see [BDD+10].

7.2. Low Data Complexity Attacks

All of the attacks covered in the this section have practical time and memory complexities. The first attack applies to two rounds *without* `MixColumns` in the second round, the other three attacks are all on full rounds (the last round *includes* `MixColumns`). Furthermore, the initial `AddRoundKey` is also considered in all four attacks. Thus, an attack on three rounds involves four round keys: the initial whitening key and three derived round keys.

The attacks on one, two and three rounds are known-plaintext attacks. The second attack on three rounds is a chosen-plaintext attack. As shown in Table 7.1, known-plaintext attacks require a higher amount of time. This is clear since such attacks have less degrees of freedom as chosen-plaintext attacks. Therefore, there are only chosen-plaintext attacks with practical time complexity on three and four rounds.

Note on figures: For all figures shown in this section, we use a similar scheme as in the original paper by Bouillaguet et al. to explain each step of the attack. The number within each cells (e.g. 4) indicates at which step the actual value for this byte is calculated during the attack. Furthermore, · indicates a known byte of a state or a round key. A cell +

marks guessed values for bytes, so, basically all possible values for such bytes are considered. Finally, ■ indicates that this byte's value is retrieved using differences. In case Property 5 is used to limit the amount of possible values for the two key bytes to twelve possibilities, then those two key bytes are depicted as ☒.

7.2.1. Attacking 2-Round AES-128 without Last `MixColumns`

We start with the attack on two rounds, without `MixColumns` in round 2. The attack is split into two phases, which are both depicted in Figure 7.2. The attack uses only two known plaintexts, and the bytes of the intermediate state for both of these plaintexts are retrieved in parallel.

We describe this attack in more detailed than the following LDC attacks since it is used in next section to construct the attack on six rounds AES-128.

The first phase starts with encrypting both plaintexts under the unknown key to retrieve their ciphertexts. With these two ciphertexts, we are able to trace their differences up to $S_{2,\texttt{SR}}$, which is the output state of `SubBytes` in round 2. From the initial whitening key rk_0 we then guess bytes 0, 5, 10 and 15, which allows us to calculate the values for column 0 in state $S_{1,\texttt{ARK}}$ and the differences of column 0 in S_2 (Steps 1-4 in Figure 7.2). Since we know the differences in column 0 before and after `SubBytes` in round 2, we are able to deduce the actual values of these bytes by using Property 1 (Step 5). We discard all key guesses which lead to impossible differences at the S-Box and get two or four pairs of actual values for valid differences from the DDT. So, on average there should be one pair of values per key guess. With these four bytes, we calculate the first column of rk_1 since we now know column 0 before and after `AddRoundKey` of round 1 (Step 6).

Additionally, since `MixColumns` is missing in the last round, bytes 0, 7, 10 and 13 of rk_2 can be calculated. We do this by applying `ShiftRows` the the known values of $S_{2,\texttt{SR}}$ (Steps 7-8). Furthermore, we calculate the key bytes $rk_0[2]$ and $rk_0[13]$ from the key schedule by using the known key bytes from rk_0 and rk_1 (Step 9). The basic property of key schedule used for this is:

$$rk_{i+1}[j] = rk_i[j] \oplus v_i.$$

For $j = 0$, we get v_0 from $rk_1[0]$ and $rk_0[0]$. Similarly, we compute $rk_0[2]$ from v_3 and $rk_1[2]$ for $j = 3$. Using similar (simple) properties of consecutive round keys, more key bytes can be retrieved in the following order: $rk_1[5]$, $rk_1[13]$, $rk_1[9]$, $rk_2[9]$, $rk_2[5]$, $rk_2[1]$, $rk_1[14]$, $rk_2[14]$ and $rk_0[9]$ (Steps 10-20, 25). Refer to Table 7.2 for the exact relations.

With these newly retrieved key bytes, we are able to calculate bytes 5, 6, 9 and 13 of S_2 upwards from the ciphertext (Steps 21-23). Further, the known key bytes of rk_1 allow to calculate bytes 5, 9 and 13 of $S_{1,\texttt{ARK}}$ (Step 24). The final step in Phase 1 is to use the plaintext and the retrieved key bytes to calculate byte 5 of $S_{1,\texttt{MC}}$ (Steps 26-28).

The second phase continues the process of switching between retrieving a few key bytes and calculating state bytes from them. First we use Property 3, Equation 2 to retrieve $rk_2[15]$ and Equation 1 to retrieve $rk_2[2]$ (Step 1). Next, we use simple key schedule properties to calculate $rk_1[15]$, $rk_2[11]$ and $rk_1[11]$ (Steps 2-4). Consequently, we can now calculate byte

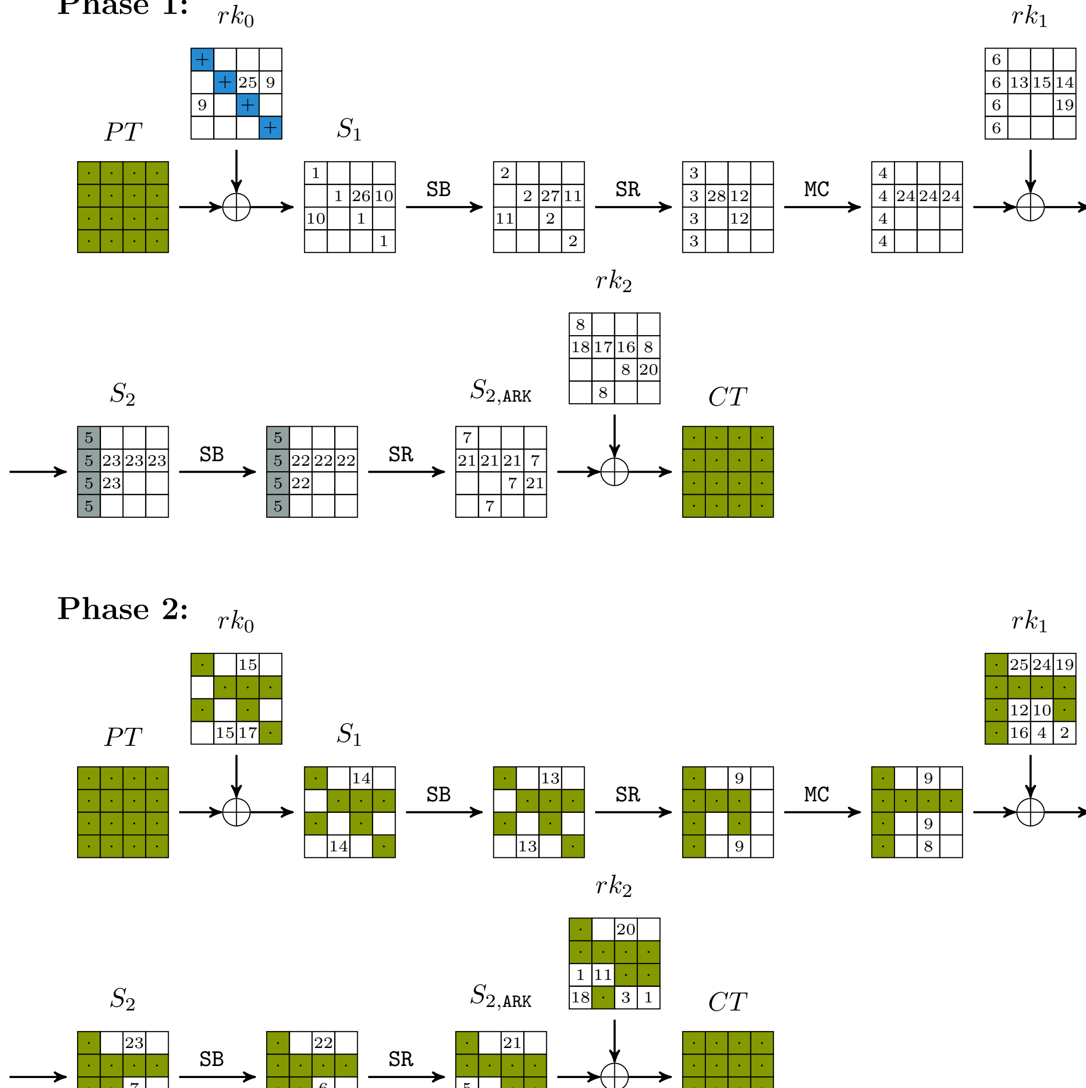


Figure 7.2.: The two phases of the LDC attack on two rounds of AES, where the second round is without `MixColumns`.

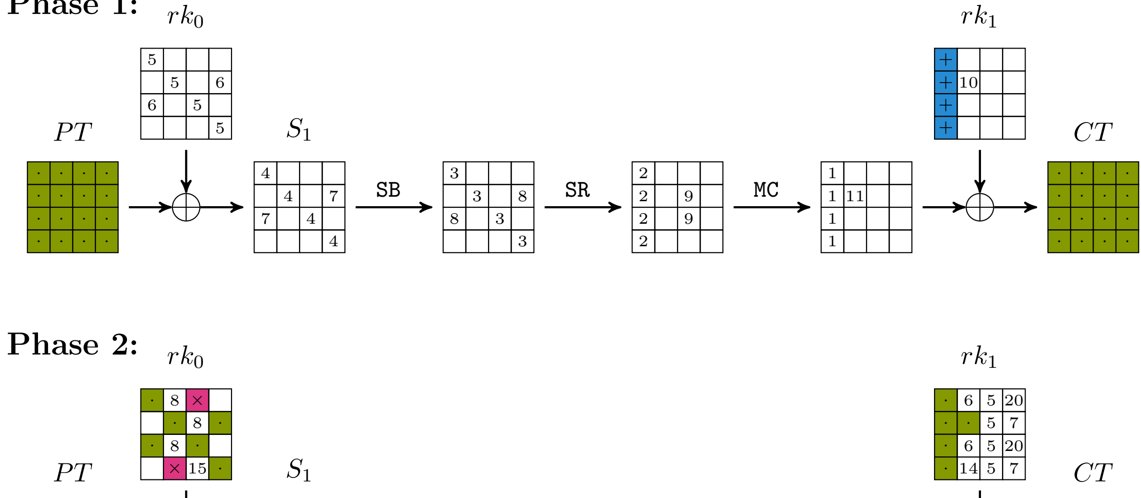


Figure 7.3.: The LDC attack on one full round using only one known plaintext.

11 of $S_{1,\texttt{ARK}}$ (Steps 5-8) and use Property 2 to retrieve bytes 10, 8 of this state and bytes 8, 11 of $S_{1,\texttt{MC}}$ (Step 9).

This enables us to retrieve three more key bytes: $rk_1[10]$, $rk_2[6]$ and $rk_1[6]$ (Steps 10-12). Further, we can calculate bytes 7 and 8 of S_1 (Steps 13-14), which allows to calculate the corresponding bytes of rk_0 (Step 15). Next, we retrieve key bytes $rk_1[7]$, $rk_0[11]$, $rk_2[3]$ and $rk_1[12]$ (Steps 16-19) and, using Equation 1 of Property 3, we can also retrieve $rk_2[8]$ (Step 20). Then, we calculate byte 8 of $S_{2,\texttt{ARK}}$ with the just retrieved key byte (Step 21). Finally, we calculate the last two missing bytes of rk_1 (Steps 22-25).

Overall, this attack has a memory complexity of 1. Since the time complexity is determined by guessing four key bytes, it is 2^{32} 2-round encryptions.

7.2.2. An Attack on 1 Round with 1 Known Plaintext

Next, we explain the even simpler attack on one round using only one known plaintext. This round-reduced version uses one full round plus the initial `AddRoundKey` with rk_0. Equally to the previous attack, it has a time complexity of 2^{32} 1-round AES encryptions, but has a higher memory complexity of 2^{24} since we use Property 5. The full attack with all its steps is shown in Figure 7.3.

First, we start by guessing four key bytes of column 0 for rk_1. This enables us to calculate certain state bytes which yield a few key bytes of rk_0 and one additional byte of rk_1 (Phase 1 in Figure 7.3). These few key bytes suffice to apply Property 5 in order to retrieve at most twelve possible values for $rk_0[7]$ and $rk_0[8]$. For each of these possible values for these bytes, we perform the following steps:

We use just retrieved possible values for key bytes $rk_0[7]$ and $rk_0[8]$ to calculate column 2 of $S_{1,\texttt{ARK}}$ (Steps 1-4). Together with the ciphertext we can now retrieve column 2 of rk_1 (Step 5). This enables us to use the key schedule to retrieve a few more bytes of both round keys and a few more state bytes. In Steps 13 and 19, the `MixColumns` property (Property 2) is used to retrieve the missing state bytes in the respective columns. Finally, we retrieve the remaining bytes of rk_1 and invert the key schedule to calculate the unknown encryption key rk_0.

In [BDF12], Bouillaguet et al. analyzed the attack model with a single known plaintext for more rounds and found attacks for up to four rounds. These attacks are all faster than exhaustive search. For example, their attack on four rounds has a time complexity of 2^{120} and a memory complexity of 2^{80}.

7.2.3. Attack on 2 and 3 Rounds Using 2 Chosen Plaintexts

This next attack on two full rounds with two chosen plaintexts is especially interesting since it has a time and memory complexity of only 2^8. We first describe this attack in full detail, and then cover how it can be extended to three full rounds with a time complexity of 2^{16}. Both attacks were found using tools developed by Bouillaguet et al. that automatically find low data complexity attacks. These tools are also described in [BDF12].

The chosen plaintexts are required to have a non-zero difference only on the first column. All other plaintext bytes have to be equal. After we have retrieved the ciphertexts by encrypting the chosen plaintexts under the unknown key, we start with a bottom up approach and calculate the difference for the full state $S_{2,\texttt{SR}}$ from the ciphertext difference. Now we assume that we know the value of $S_1[1]$. As shown in Figure 7.4, the knowledge of $S_1[1]$ also reveals $S_{1,\texttt{MC}}[13]$ (Step 1). Further, also the differences for all bytes of column 3 in S_2 are known because of Property 2. Since we now know the differences before and after `SubBytes` in round 2 for the whole column 3, we can deduce the actual values using the DDT for each byte (Step 2). Consequently, it is possible to deduce bytes 3, 6, 9 and 12 of rk_2 (Steps 3-4). Hence, by only knowing the value of one byte ($S_1[1]$) we are able to retrieve four bytes from the round key rk_2. Furthermore, the additional knowledge of bytes $S_1[0]$, $S_1[2]$ and $S_1[3]$ allows to retrieve the full rk_2.

The straightforward approach to retrieve rk_2 would be to try all values for these four bytes (bytes 0, 1, 2 and 3 of S_1) and retrieve the possible keys. This would take 2^{32} time and would result in about 2^{32} possible key candidates. However, Bouillaguet et al. found a different attack that is a lot faster by using the equations of Property 4. The basic idea is to try all 2^8 possible values for $S_1[1]$ and calculate the values $S_1[0]$, $S_1[2]$ and $S_1[3]$ from $S_1[1]$ (see also Figure 7.4):

Assume that $S_1[1]$ is known, $S_1[0]$ can then be calculated using Equation 3 of Property 4:

$$\begin{aligned} &S_{1,\texttt{MC}}[12\ldots15] \oplus S_{2,\texttt{MC}}[8\ldots11] \oplus S_{2,\texttt{MC}}[12\ldots15] \\ &\quad = \texttt{InvMC}(S_2[12\ldots15] \oplus S_3[8\ldots11] \oplus S_3[12\ldots15]). \end{aligned}$$

Since we assumed that $S_1[1]$ is known, all bytes marked with 1, 2, 3 and 4 in Figure 7.4 are known. This includes column 3 of S_2. Hence, the full right side of the equation is known

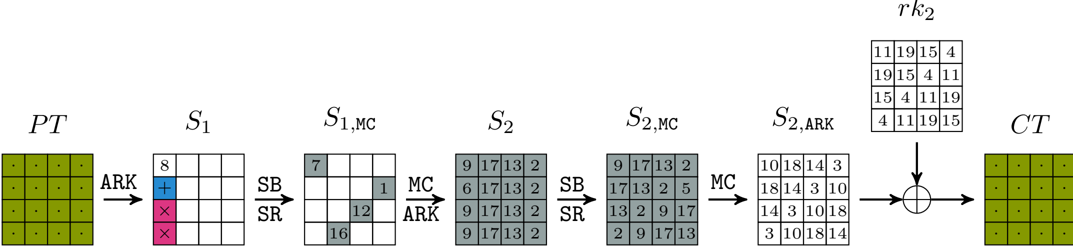


Figure 7.4.: The LDC attack on two full rounds using two chosen plaintexts. The time and memory complexity of this attack is as low as 2^8.

because $S_3[8\dots11]$ and $S_3[12\dots15]$ are just ciphertext bytes. From the left side, we also know $S_{1,\mathtt{MC}}[13]$ and $S_{2,\mathtt{MC}}[9]$, which allows us to use the above equation to retrieve $S_{2,\mathtt{MC}}[13]$ by:

$$S_{2,\mathtt{MC}}[13] = S_{2,\mathtt{MC}}[9] \oplus S_{1,\mathtt{MC}}[13] \oplus \mathtt{InvMC}(S_2[12\dots15] \oplus S_3[8\dots11] \oplus S_3[12\dots15])[1].$$

Note that $\mathtt{InvMC}(\dots)[1]$ selects the second byte of the resulting 4-byte vector. Having calculated $S_{2,\mathtt{MC}}[13]$ (Step 5), we can now also calculate $S_2[1]$ (Step 6).

Further, we know that bytes 1, 2 and 3 of column 0 in $S_{1,\mathtt{MC}}$ have zero difference. We use this knowledge and the known difference in $S_2[1]$ to deduce the difference of $S_{1,\mathtt{MC}}[0]$ using Property 2. Finally, we use the DDT to retrieve the actual values for $S_{1,\mathtt{SR}}[0]$ and hence also $S_1[0]$ (Steps 7-8). We can now compute four more bytes for rk_2 (Steps 9-11).

As for $S_1[2]$, Equation 2 of Property 4 can be used to retrieve its value from $S_1[1]$. First, we rearrange the equation to

$$S_{2,\mathtt{MC}}[4\dots7] \oplus S_{2,\mathtt{MC}}[8\dots11] = S_{1,\mathtt{MC}}[8\dots11] \oplus \mathtt{InvMC}(S_2[8\dots11] \oplus S_3[4\dots7] \oplus S_3[8\dots11]).$$

We can see that the third byte of the right-hand side can be calculated from $S_1[2]$. To use this, we construct a table for all possible values of $S_1[2]$ which is indexed by the third byte of the right-hand side of the above equation

$$r = S_{1,\mathtt{MC}}[10] \oplus \mathtt{InvMC}(S_2[8\dots11] \oplus S_3[4\dots7] \oplus S_3[8\dots11])[2].$$

For a known $S_1[1]$ we are able to calculate bytes $S_{2,\mathtt{MC}}[6]$ (Step 2) and $S_{2,\mathtt{MC}}[10]$ (Step 9). Since we know the values of these two bytes, we can calculate $r = S_{2,\mathtt{MC}}[6] \oplus S_{2,\mathtt{MC}}[10]$. Thus, we simply look up r in the table and retrieve the value for $S_1[2]$ (indicated by ☒ in Figure 7.4). Consequently, we use this value to compute four more bytes of rk_2 (Steps 12-15).

For $S_1[3]$ a similar table can be constructed using Equation 1 of Property 4. This time, the fourth byte of the vector is used and the rearranged equation to construct the table is:

$$S_{2,\mathtt{MC}}[3] \oplus S_{2,\mathtt{MC}}[7] = S_{1,\mathtt{MC}}[7] \oplus \mathtt{InvMC}(S_2[4\dots7] \oplus S_3[0\dots3] \oplus S_3[4\dots7])[3].$$

As before, the complete right-hand side can be calculated from $S_1[3]$. Thus, we enumerate all its values and index the table by the right-hand side of the equation. To retrieve the

corresponding value of $S_1[3]$ for a given $S_1[1]$, we just calculate the left-hand side of the equation and perform a lookup in the hash table. Thus, we can calculate the remaining four bytes of rk_2 (Steps 16-19).

For the full attack, we enumerate all possible values of $S_1[1]$ and calculate the other three bytes of this column as described above. This yields 2^8 values for the full round key rk_2 which can be checked using exhaustive search.

Performing the above calculations for one value of $S_1[1]$ equals about one AES encryption. Hence, the time complexity of the whole attack is 2^8 encryptions. As for the memory complexity, Bouillaguet et al. give an approximate of 2^8 AES blocks. However, we believe it is actually less than that since the attack only requires storage for two tables with 2^8 entries per table, and each entry has a size of one byte. This gives only $2 \cdot 2^8 \cdot 2^{-4} = 2^5$ AES blocks.

Extension to 3 Rounds

This attack can easily be extended to three rounds by adding one round at beginning. Here, only one non-zero difference in byte 0 of the plaintexts is allowed. If we guess byte 0 of whitening key rk_0, then we are able to calculate the differences in column 0 of S_2. Since the bytes in this column are the only active bytes in S_2, it suffices to run the above 2-round attack from here with the known differences in S_2 and the values of ciphertexts as input.

7.2.4. Attack on 3 Rounds with 9 Known Plaintexts

For an attack on three rounds of AES-128 with nine plaintexts, we assume three full rounds and the initial `AddRoundKey` with the whitening key. Additionally, we exchange `MixColumns` with `AddRoundKey` in the last round. As explained before, this is possible due to the linearity of both operations. However, we now have to use the equivalent round key (rk'_3) instead of rk_3. We also switch `MixColumns` and `AddRoundKey` in round 2. Further, we denote byte 0 of state $S_{2,\texttt{MC}}$ for the i-th plaintext as b_i[2]. Thus, $b_i = S^i_{2,\texttt{MC}}[0]$.

The first phase of this attack is to guess bytes 0, 7, 10 and 13 of rk'_3 and partially decrypt the corresponding ciphertext bytes up to S_3 (Steps 1-2 in Figure 7.5). Since we now know the actual values for column 0 of S_3, we can calculate the differences in b_i for every key guess and every ciphertext (Step 3). With these differences we construct the vector

$$b_1 \oplus b_2, b_1 \oplus b_3, \ldots, b_1 \oplus b_9$$

and store all encountered values in a hash table.

In the second phase, we perform similar computations using the plaintext. First, we guess bytes 0, 5, 10, 15 of rk_0 and byte 0 of rk_1. This allows us to partially encrypt all plaintexts up to byte 0 of $S_{2,\texttt{MC}}$ (Steps 4-6). Thus, we also get values and differences for all b_i. From these values we again construct the difference vector from above, and check for each vector if it appears in the hash table. If the vector occurs in the hash table, we keep the corresponding key guess. Since this is a 64-bit filter, only $2^{72} \cdot 2^{-64} = 2^8$ key guesses should remain.

[2]Normally, this is the state right before `MixColumns` in round 2. However, since we switched `AddRoundKey` and `MixColumns` in this round, this state is the input to `AddRoundKey`.

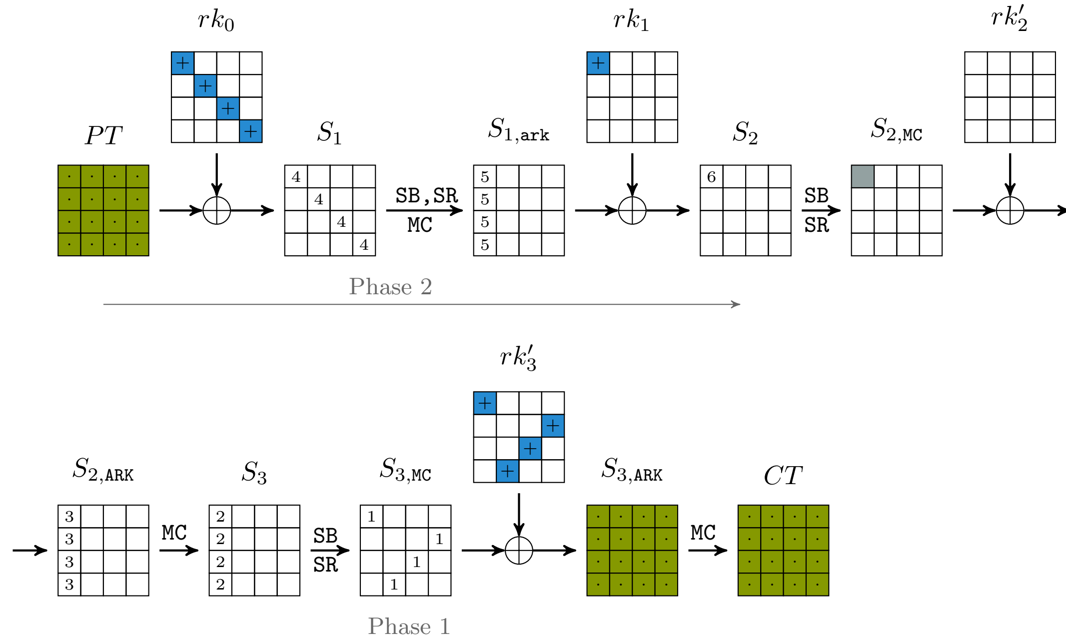


Figure 7.5.: The LDC attack on three full rounds using nine known plaintexts. Both phases calculate a 64-bit difference vector for byte 0 of $S_{2,\mathtt{MC}}$.

We can repeat this process for the remaining three columns to get 2^{32} suggestions for the full round key rk_0. Since this is a very small amount of possible keys, we can check them using a brute force attack to eliminate false-positives. Overall, the memory complexity is about $2^{32} \cdot 2^3 \cdot 2^{-4} = 2^{31}$ AES states, and the time complexity is about 2^{40} encryptions.

7.3. Known-Plaintext Attack on 6-Round AES-128

In this section, we present an example on how an LDC attack can be used as a building block for a more complex attack. In this case, a differential trail similar to that used for differential enumeration is used to limit the amount of possible values for an intermediate state. This differential is in fact equal to that from Section 6.4 except the last round is removed. Thus, it covers only three rounds (see also Figure 6.3). Nevertheless, the amount of possible actual values is the same with 2^{64}.

The basic concept of the attack is to place the differential from S_2 (input state to differential with only one non-zero difference in byte 0) to S_5 (output state of differential), which results in 2^{64} possible values for state S_4. Then, the LDC attack from Section 7.2.1 is applied to states $S_{4,\mathtt{ARK}}$ to S_7, which is the ciphertext.

The full attack is thus as follows:

1. Take 2^{109} plaintexts and encrypt them.

2. Insert the plaintext-ciphertext pairs into a hash table.

3. Use the hash table to create differential pairs where bytes 1-4, 6-9, 11-14 of the plaintext and bytes 1-6, 8, 9, 11, 12, 14 and 15 of the ciphertext are equal. Since this is a 192-bit filter, there will be about $2^{216} \cdot 2^{-192} = 2^{24}$ pairs left[3].

4. Assume that those are right pairs. Hence, in state S_4 there are only 2^{64} possible actual values for each potential right pair.

5. For each of those possible values, run the LDC attack from Section 7.2.1 on states $S_{4,\mathtt{ARK}}$ to S_7.

The data complexity of this attack is 2^{109} plaintexts. As for the time complexity, the LDC attack has a time complexity of 2^{32} and for each of the 2^{24} pairs there are 2^{64} possible states. Hence, the overall time complexity of this attack is $2^{24} \cdot 2^{64} \cdot 2^{32} = 2^{120}$. The advantage of this attack is that it has no requirements on the plaintexts. The memory complexity of the attack is $2^{24} \cdot 2^{64} = 2^{88}$ times the memory complexity of the LDC attack. According to the authors, this attack is currently the best *known-plaintext* attack on 6-round AES.

[3]Since we take 2^{109} plaintexts, there are $\binom{2^{109}}{2} \approx 2^{216}$ possible pairs we can create from them.

8. Biclique Attacks on AES

In general, meet-in-the-middle attacks on block ciphers received less attention because they require major parts of the cipher to be independent from the encryption key. Cipher properties such as high diffusion counteract this. Consequently, as we have seen with LDC attacks on AES above, standard meet-in-the-middle attacks only work on a limited number of rounds. However, Bogdanov et al. combined the meet-in-the-middle approach with another technique called *bicliques* [BKR11]. This enabled them to construct new attacks that are faster than a standard brute force attack and work on all three variants of AES, up to their respective full number of rounds. The concept of bicliques was first introduced by Khovratovich et al. for the cryptanalysis of hash functions, specifically the SHA-3 candidate Skein-512 and SHA-2 variants [KRS11]. Bogdanov et al. were the first to bring over this concept to block cipher cryptanalysis with their biclique attacks on AES, which we cover in this section.

The remainder of this chapter is mainly based on, and structured like [BKR11]. To familiarize the reader with this new technique, we first define bicliques and present the high-level concept of how to use them for attacks on block ciphers. Then, we present two techniques to construct bicliques as well as two concepts of how to recover the encryption key. Further, we present a biclique attack on the full AES-128.

8.1. The Basics of Biclique Cryptanalysis

A meet-in-the-middle attack works by splitting the key-space into groups of 2^{2d} keys, for some d. Within a group, the keys are represented as $2^d \times 2^d$ matrix $K[i,j]$ with $0 < i,j < 2^d$. For the attack, we choose an internal variable v and split the encryption function E of the cipher into two parts e_1, e_2, where $E = e_2 \circ e_1$. Furthermore, $K[i,j], e_1, e_2$ are chosen such that for a plaintext and key, v is equal for all keys in a row of $K[i,j]$:

$$P \xrightarrow[e_1]{K[i,\cdot]} v$$

Moreover, for a ciphertext and a key, v is identical for all keys in a column of $K[i,j]$:

$$v \xleftarrow[e_2]{K[\cdot,j]} C$$

The attacker then obtains a plaintext-ciphertext pair and uses $K[i,j]$ to compute 2^d possible values $\overrightarrow{v}$ from the plaintext, and 2^d values for $\overleftarrow{v}$ from the ciphertext. If $\overrightarrow{v_i} = \overleftarrow{v_j}$ for some i,j, then $K[i,j]$ is a candidate for the secret encryption key. To calculate the expected

number of key candidates, Bogdanov et al. use the formula $2^{2d-|v|}$, where $|v|$ is the bit size of v.

Given there exists such a meet-in-the-middle attack for a given number n of rounds, its performance advantage over a brute force attack is about 2^d cipher executions. This is due to the fact that the meet-in-the-middle attack tests 2^{2d} keys with only 2^d executions of n cipher rounds. If v is not a full internal state but only a fraction of a state (for example one byte), the advantage can be even higher than 2^d. This is because only the bytes necessary for v have to be computed and all others can be skipped.

8.1.1. Bicliques

Bogdanov et al. proposed to increase the number of rounds that can be attacked with meet-in-the-middle attacks by the use of bicliques.

Definition 8.1 (Biclique)**.** Let S, C denote two states of a block cipher, where C is l rounds after S. Further, let f be a sub-cipher of the full encryption algorithm $E = g \circ f$ such that $f_K(S) = C$ for a key K. The triple $(\{C_i\}, \{S_j\}, K[i,j])$ is called *biclique* if $f_{K[i,j]}(S_j) = C_i$ for $0 < i, j < 2^d$.

The parameter d is called *dimension*, and l is the *length* of the biclique. $K[i,j]$ is again represented as $2^d \times 2^d$ matrix. Basically, a biclique connects 2^d states S_j to 2^d states C_i using 2^{2d} keys as shown in Figure 8.1.

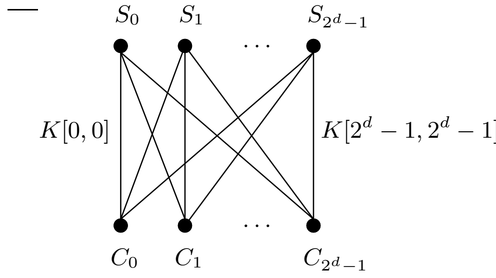


Figure 8.1.: A d-dimensional biclique as shown in [BKR11].

8.1.2. Concept of a Biclique Attack

The idea of a biclique attack is to combine the biclique with a meet-in-the-middle attack, and hence, increase the number of rounds that can be attacked. Throughout the rest of Chapter 8, we assume C_i to be ciphertexts. Consequently, the biclique is placed at the end of the block cipher and S_j is some intermediate state. We do this because the attacks on AES we present also have the biclique placed at the end. However, the biclique could also be placed at the beginning, where S_j would then be a plaintext and C_i an internal state of the cipher. When placing the biclique at the end, the concept for the full attack on a n-round block cipher is to cover the first m rounds using a meet-in-the-middle attack and the remaining $l = n - m$ using a biclique.

With the basic idea explained, we continue with the rough outline of a biclique attack, which is as follows:

1. Partition the key space into groups of 2^{2d} keys and find sets $\{C_i\}, \{S_j\}$ which satisfy:

$$\forall i, j : S_j \xrightarrow[f]{K[i,j]} C_i$$

2. Decrypt the ciphertexts C_i, and retrieve their corresponding plaintexts P_i from the decryption oracle. Then, $E_k(P_i) = C_i$ for the secret encryption key k.

3. Finally, check for each key $K[i,j]$ if it maps P_i to S_j. More formally, check if:

$$\exists i, j : P_i \xrightarrow[g]{K[i,j]} S_j$$

   Each $K[i,j]$ which satisfies this mapping is a key candidate. If there exists a meet-in-the-middle attack for m rounds, it can be used to check the keys.

Consequently, the advantage of a biclique attack over a brute force attack lies in Steps 1 and 3. In Step 1, the biclique yields a way to enumerate values for S_j for each ciphertext. The advantage of the biclique is that this enumeration process is faster than computing l cipher rounds. Similarly in Step 3, the process to verify a key $K[i,j]$ (by a meet-in-the-middle attack or similar method) is usually faster than computing m cipher rounds.

In the following section we explain two methods on how to construct bicliques (Step 1), and in Section 8.3 we cover two paradigms on how to recover the secret key (Step 3).

8.2. Biclique Construction for Block Ciphers

In [BKR11], Bogdanov et al. present two techniques to construct bicliques for a block cipher. Both methods are mostly independent of the actual cipher, and have in common that the key groups are chosen according to some specified differential trails. This approach is more advantageous than just fixing the sets $\{S_j\}$, $\{C_i\}$, and deriving keys from them because this would require at least 2^{2d} executions of the sub-cipher f. Furthermore, both techniques use related-key differential trails over f for the construction of the biclique. Note that although related-key differentials are used, the attack itself is still a single-key attack.

8.2.1. Independent Related-Key Differentials

For the first technique to construct bicliques, we use *independent* related-key differentials. Independent in this context means that the differential trails do not share any active components (for AES this would be active bytes of a state). Overall, we choose $2 \cdot 2^d$ differentials which we split into two sets Δ_i, ∇_j of equal size. We construct all differentials with respect to $S_0 \xrightarrow[f]{K[0,0]} C_0$ which is called *base computation*. Moreover, all differentials cover the full sub-cipher f and the two sets of differentials are constructed as follows:

Δ_i-differentials: Differentials in this set start with input difference 0, end in some output difference Δ_i, and use a key difference Δ_i^K. More formally:

$$0 \xmapsto[f]{\Delta_i^K} \Delta_i$$

with $\Delta_0^K = 0$ and $\Delta_0 = 0$ to respect the base computation.

∇_j-differentials: Differentials in this set start with input difference ∇_j, end in output difference 0, and use a key difference ∇_j^K. More formally:

$$\nabla_j \xmapsto[f]{\nabla_j^K} 0$$

with $\nabla_0^K = 0$ and $\nabla_0 = 0$.

Since the differential trails of both sets do not share active components, 2^d combined (Δ_i, ∇_j)-differentials can be constructed by `XOR` of two trails from different sets. Hence,

$$\forall i, j : \nabla_j \xmapsto[f]{\Delta_i^K \oplus \nabla_j^K} \Delta_i$$

It is easy to see that this is a valid construction because the trails are independent. Consequently, an active component belongs either to a Δ_i-differential or to a ∇_j-differential, but never to both. This is very similar to boomerang attacks [Wag99], first introduced by Wagner. Hence, the above construction can be seen as a boomerang on f with probability 1.

Since $(S_0, C_0, K[0,0])$ conforms to all differential trails of both sets, it also conforms to the combined differentials, and we can add it to the combined (Δ_i, ∇_j)-differentials using `XOR`:

$$S_0 \oplus \nabla_j \xmapsto[f]{K[0,0] \oplus \Delta_i^K \oplus \nabla_j^K} C_0 \oplus \Delta_i$$

This already looks very similar to the definition of a biclique. Thus, we simply obtain the d-dimensional biclique by setting:

- $S_j = S_0 \oplus \nabla_j$,
- $C_i = C_0 \oplus \Delta_i$, and
- $K[i,j] = K[0,0] \oplus \Delta_i^K \oplus \nabla_j^K$.

This construction is more efficient than a straightforward construction since it takes no more than $2 \cdot 2^d$ executions of f. Basically, this method allows us to create differentials of higher dimension. However, the length is still limited because the diffusion properties of a block cipher limit the length of the differential trails.

8.2.2. Interleaving Related-Key Differentials

The bicliques constructed by the above technique are clearly limited in length. The second approach presented here tries to construct longer bicliques by considering related-key differentials that overlap, and hence, share active components. However, if a block cipher is secure against differential cryptanalysis, constructing long bicliques with high degree is hard. Consequently, a trade-off has to be made. In this case, it is a trade-off between the length and degree of the biclique. By aiming to construct bicliques with very low degree, constructing long bicliques becomes easier because less degrees of freedom are consumed by the biclique itself.

For this method, we aim to create bicliques of minimal degree ($d = 1$) for a fixed key group of four keys. The construction is similar to that of the rebound attack [MRST09]:

1. Choose an **intermediate state** T in f, and split $f = f_2 \circ f_1$ such that $S_j \xmapsto{f_1} T$ and $T \xmapsto{f_2} C_i$.
2. Select truncated differentials Δ, $\bigtriangledown$, with the Δ-trails over f_1 and the $\bigtriangledown$-trails over f_2.
3. Guess the differences of the active components in all trails towards T. Then, retrieve the actual pairs for T that satisfy these guessed differences over the full sub-cipher f. This resembles the **inbound phase** of the rebound attack.
4. Propagate each possible pair of T outwards until S_j or C_i is reached. This step is similar to the **outbound phase** of the rebound attack. After this step, only a few pairs for S_j and C_i should be left. Those are the input and output states of the biclique.

This generic approach allows multiple improvements:

- Not all differences of the differential trails have to be guessed. Instead, guessing only parts of them is less restrictive. Choosing the components which are guessed carefully, ensures that wrong pairs are still filtered during the outbound phase.
- Instead of fixing all keys within a key group, only the differences of the key can be fixed. The actual values of the keys can then be retrieved during the construction of the biclique.
- A major advantage of this construction is its reusability: The construction of one biclique allows to produce multiple other bicliques with only some very small modifications.

8.3. Methods for Key Recovery

This chapter focuses on how to recover the encryption key. Basically, we describe two methods on how to perform the check

$$\exists i, j : P_i \xrightarrow[g]{K[i,j]} S_j$$

on the remaining rounds of the cipher which are not covered by the biclique.

Assume a n-round block cipher for which we can construct a biclique that covers at most l rounds. Let the encryption function of the cipher be $E = g \circ f$, and f cover the last l rounds (i.e. the biclique). There are two possibilities for the above check: If there exists a meet-in-the-middle attack for the remaining $n - l$ rounds, we can apply it to verify the 2^{2d} keys with less than 2^d executions of the sub-cipher g. This approach is called *long-biclique*, since the biclique is long enough to admit a meet-in-the-middle attack on the remaining rounds.

On the other hand, if there exists no meet-in-the-middle attack of the required length, we have to find an alternative method to check the above formula. At first, this approach seems disadvantageous because the complexity for recovering the key increases to 2^{2d}. However, Bogdanov et al. presented another technique called *matching with precomputations*, which allows to decrease the cost of recovering the key. Moreover, since the bicliques used for this technique are shorter and have an easier description, the attack becomes more compact and has lower data complexity. For this second approach, bicliques constructed from independent related-key differentials are used. Hence, this method is called *independent-biclique*.

In the remainder of this section, we analyze the complexities of both attack paradigms and describe the technique of matching with precomputations.

8.3.1. Long-Biclique

The idea of this approach is to construct a biclique of required length such that we are able to use a meet-in-the-middle attack on the remaining rounds of the block cipher. One advantage of this method is that we are able to use the meet-in-the-middle attack without major modifications. So, we basically extend the meet-in-the-middle attack by l rounds. A disadvantage is the construction of the biclique. Since we have to use interleaving related-key differential trails for the construction to get longer bicliques, we loose most degrees of freedom. As a result, we get higher data complexity because we might not have enough degrees left to restrict the plaintexts and ciphertexts. Furthermore, the total number of rounds we can attack is limited because both, meet-in-the-middle attack and biclique, are limited in their length.

As shown in Figure 8.2, we choose an internal variable v for the meet-in-the-middle attack, and recover the keys by calculating towards v from P_i and S_j. We also note that the biclique needs an appropriate dimension for the meet-in-the-middle attack. For example, if the meet-in-the-middle attack requires four plaintexts, the biclique has to have dimension $d = 2$.

To recover the full encryption key, we have to test all 2^k possible key values, where k is the bit size of the key. Consequently, we execute the attack 2^{k-2d} times. The time complexity of the full attack can then be calculated by the formula:

$$C_{full} = 2^{k-2d}(C_{biclique} + C_{match} + C_{falsepos})$$

$C_{biclique}$ denotes the complexity for the construction of the biclique. C_{match} is the complexity of the meet-in-the-middle attack, which is at most 2^d. Since there is a certain chance

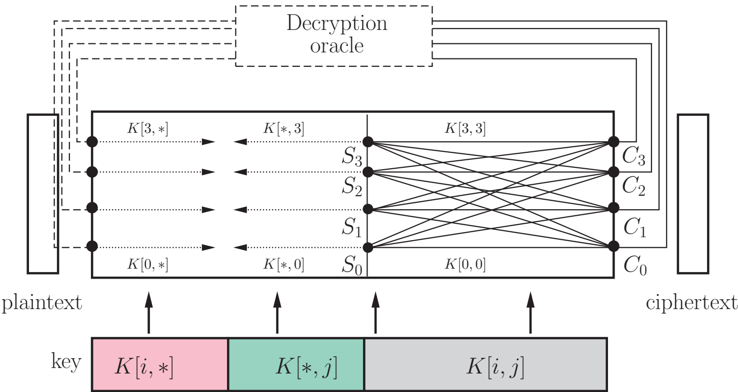


Figure 8.2.: A long-biclique attack with a 2-dimensional biclique at the end [BKR11].

for false-positives, $C_{falsepos}$ denotes the time complexity to eliminate these wrong key candidates. Normally, $C_{falsepos}$ is negligible since the overall time complexity is dominated by C_{match}. $C_{biclique}$ can also be high especially since the construction using differentials is rather time-consuming. However, since we can use one construction to obtain multiple bicliques, it is lower than C_{match}. The memory complexity is dominated by the construction of the biclique since the meet-in-the-middle attack itself has rather low memory requirements.

8.3.2. Matching with Precomputations

Matching with precomputations is basically a replacement for the meet-in-the-middle attack. Again, we aim to check the equation:

$$\exists i, j : P_i \xrightarrow[g]{K[i,j]} S_j$$

This alternative is required because we might not be able to construct long enough bicliques for a meet-in-the-middle attack to cover the whole cipher.

To recover the encryption key using matching with computations, we first choose a small internal component v (for example one byte of a state) between P_i and S_j. Then, we compute 2^d values for:

$$\forall i : P_i \xrightarrow[g]{K[i,0]} \overrightarrow{v}$$

and also 2^d values for:

$$\forall j : \overleftarrow{v} \xleftarrow[g]{K[0,j]} S_j$$

All $2 \cdot 2^d$ values are stored in memory. For each $K[i, j]$, $i > 0$, $j > 0$, we recompute only those parts that differ from the precomputed ones and are required to calculate v. The advantage

of this method depends on the diffusion properties of the block cipher. For AES, the key schedule has low diffusion, hence, a lot of computation can be skipped in that area.

8.3.3. Independent-Biclique

The independent-biclique approach uses shorter bicliques with higher dimension. To construct such bicliques, we use independent related-key differentials. Further, the key recovery is done by matching with precomputations since meet-in-the-middle attacks are not possible due to their limitations in length. The computational advantage of this approach is the high dimension of the biclique because the biclique has to be constructed only once for every 2^{2d} keys, and d is large. Moreover, the precomputations for the matching part are also only done once per 2^{2d} keys, and thus, they are negligible.

For the independent-biclique approach, the time complexity is calculated according to the formula:

$$C_{full} = 2^{k-2d}(C_{biclique} + C_{precomp} + C_{recomp} + C_{falsepos})$$

The complexity of the precomputations for the matching is denoted by $C_{precomp}$, and C_{recomp} is the complexity for the recomputations of v which depends on the diffusion properties of the cipher.

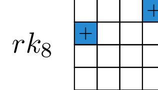
Since the biclique construction using independent related-key differentials is rather cheap, the overall time complexity for this approach is dominated by the recomputation complexity C_{recomp}. An additional influence on the time is the size of v since it has a direct impact on C_{recomp} and $C_{precomp}$. The memory requirements for this attack are at most 2^d computations of the full cipher which are required for matching with precomputations.

8.4. Biclique Attack on the Full AES-128

In this section, we describe the attack on the full ten rounds of AES-128. We place a biclique on rounds 8-10 and use matching with precomputations on rounds 1-7 to retrieve the encryption key.

The first step of the attack is to partition the key space of 2^{128} possible keys into 2^{112} groups of 2^{16} keys. Consequently, the biclique we construct has to be of dimension $d = 8$. We partition the key space with respect to rk_8, which is the key of round 8 (the beginning of biclique). The base keys ($K[0,0]$) for all groups are obtained by fixing $rk_8[1]$ and $rk_8[12]$ to zero and enumerating the remaining 14 bytes over all possible values. This yields 2^{112} base keys, each one defines a different group of keys. To get the keys within a group, we enumerate $i, j \in \{0, 1, \ldots, 255\}$ and add i, j to specific bytes of the group's base key using `XOR` (see Figure 8.3). Specifically, we set:

$$\begin{aligned} rk_8[1] &= rk_8[1] \oplus j \\ rk_8[9] &= rk_8[9] \oplus j \\ rk_8[8] &= rk_8[8] \oplus i \\ rk_8[12] &= rk_8[12] \oplus i. \end{aligned}$$



Base key for each key group.
Bytes 1 and 12 are fixed to zero.

rk_8

Enumeration of keys
within a group by i, j.

Figure 8.3.: Construction of key groups for the biclique attack on AES-128. The groups are defined on round key rk_8.

Note that although we fixed only two bytes for the base keys, we add i, j to four bytes of each base key. This is done to get the necessary key differences required for the Δ_i and ∇_j differential trails.

Now, we construct a 3-round biclique from an independent related-key differential: Initially, we perform the base computation by setting $C_0 = 0$ and calculating $S_0 = f^{-1}_{K[0,0]}(C_0)$. Further, we construct the two sets of differential trails Δ_i, ∇_j by using their corresponding key difference. More specifically, the Δ_i-differentials are based on the key difference Δ_i^K of rk_8, and the ∇_j-differentials are based on ∇_j^K of rk_8. The key difference Δ_i^K covers all keys from the group with $j = 0$ and $i \in \{0, \ldots, 255\}$. The key difference ∇_j^K covers all keys constructed from the base key with $i = 0$ and $j \in \{0, \ldots, 255\}$. These two key differences define the two sets of differential trails depicted in Figure 8.4. The combination of both trails yields the 8-dimensional biclique.

To calculate the values of S_j, we choose ciphertexts C_i according to the ciphertext differences of Δ_i. As shown in Figure 8.4, twelve bytes are active. This would result in 2^{96} possible ciphertexts when we enumerate the values of all active bytes. However, the values of bytes 10 and 14 of the ciphertext are equal because the key difference of $rk_{10}[10]$ is equal to the key difference of $rk_{10}[14]$. Hence, there only are 2^{88} possible ciphertexts. For each of these 2^{88} ciphertexts, we calculate the corresponding value for S_j by using the base computation for S_0 and both differential trails.

8.4.1. Key Recovery

After having constructed the biclique, we retrieve the corresponding plaintexts P_i for all the ciphertexts C_i and move on to the last step: the key recovery. This is done using *matching with precomputations*, where we choose v to be byte 12 of state S_3. Then, we perform the $2 \cdot 2^d$ precomputations towards v from P_i ($\overrightarrow{v}$) and S_j ($\overleftarrow{v}$) and store them in memory. Finally, we simply have to perform the necessary recomputations for the remaining keys $K[i, j]$, $i, j > 0$.

The computations from P_i towards $\overrightarrow{v}$ are shown in Figure 8.5. These recomputations are defined by the difference between $K[i, j]$ and $K[i, 0]$ of the base computation. To retrieve the key difference in rk_0, rk_1 and rk_2, we trace the key differences, used to define the differential trails on rk_8, backwards through the reverse key schedule. Hence, we get a difference in nine bytes of rk_0. Since rk_0 is added to S_1 by the first `AddRoundKey` operation, differences are introduced in state S_1 which we can trace towards $\overrightarrow{v}$. In this trace, we consider only bytes required to calculate $\overrightarrow{v}$. This results in 13 S-Boxes which have to be recomputed.

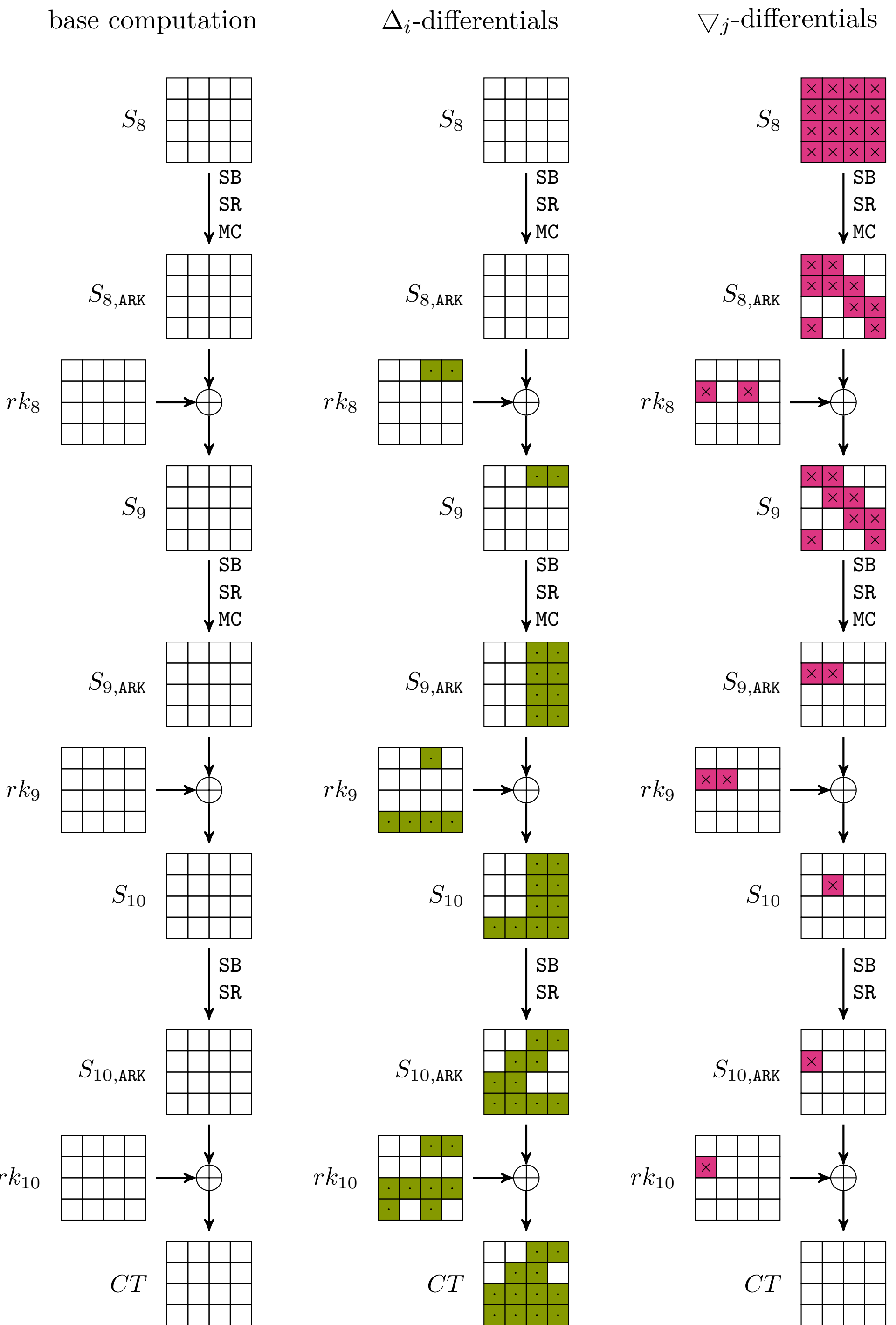


Figure 8.4.: The differential trails for the biclique attack on AES-128.

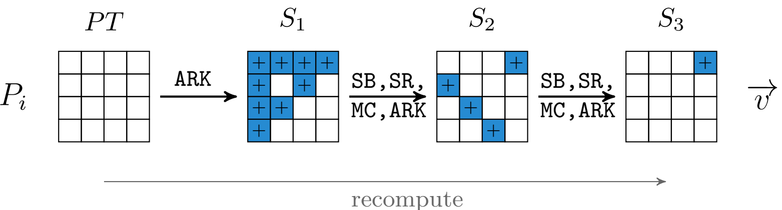


Figure 8.5.: Bytes that need to be recomputed between P_i and $\overrightarrow{v}$.

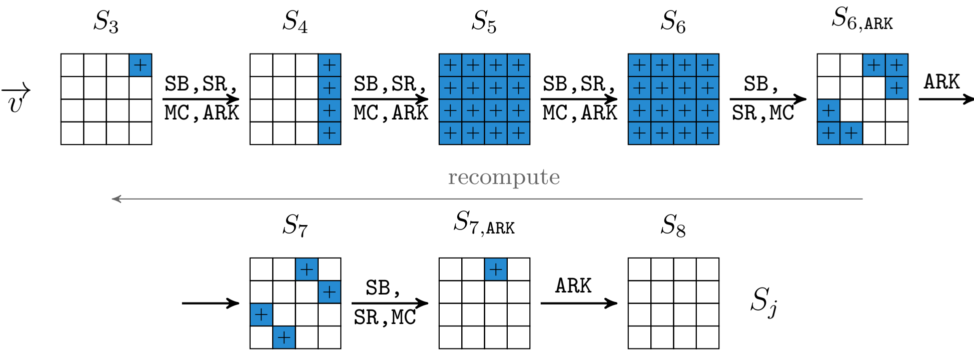


Figure 8.6.: Bytes that need to be recomputed between S_j and $\overleftarrow{v}$.

For the backward direction from S_j towards $\overleftarrow{v}$ we perform a similar analysis. This time, the differences between $K[i, j]$ and $K[0, j]$ determine the recomputations. The full trace of differences is shown in Figure 8.6. It can be seen from this figure that between S_6 and $S_{4,\mathtt{ARK}}$ all bytes have to be recomputed. However, overall we only have to evaluate 41 S-Boxes for each key in backward direction.

8.4.2. Complexity Analysis

As we explained in Section 8.3.3, the time complexity of the independent-biclique approach is determined by C_{recomp}. Hence, we perform an accurate analysis of this complexity. We do this by counting the number of S-Box evaluations in the key schedule and in `SubBytes`. From Figure 8.5 and Figure 8.6 we can directly see the active bytes and derive the required amount of operations. Overall, one recomputation involves 54 S-Box evaluations, which is equivalent to 3.375 `SubBytes` operations. The full AES-128 contains 10 `SubBytes` and its key schedule another 2.5, thus, overall $10 + 2.5 = 12.5$ evaluations of `SubBytes`. Consequently, one recomputation is equivalent to $2^{16} \cdot 3.375/12.5 = 2^{14.11}$ full AES-128 executions. Since $C_{biclique}$ and $C_{precomp}$ are both below 2^8, we assume them to be at most 2^7; $C_{falsepos}$ is $2^{16-8} = 2^8$. The full complexity of this attack is thus:

$$2^{112} \cdot (2^7 + 2^7 + 2^{14.11} + 2^8) = 2^{126.15}$$

The memory complexity is dominated by matching with precomputations, thus, this is at most 2^8 computations of rounds 1-7. As mentioned before, the data complexity of the

attack is 2^{88} chosen plaintexts. The success probability of this attack is 100% because every possible key from the key space is tested.

8.5. Overview of other Biclique Attacks on AES

In [BKR11], Bogdanov et al. present multiple attacks on all three variants of AES using bicliques. In this section, we give a short overview of these remaining attacks.

First, bicliques can also be used for attacks on full AES-192 and AES-256. Both attacks are very similar to the attack on full AES-128 presented in the previous section. The main differences are in the construction of the biclique because the key schedule of these variants differs from AES-128. Consequently, also the bytes to be recomputed differ because the active bytes in the round keys change. However, the overall concept and outline of these attacks remains the same as both attacks use bicliques constructed from independent related-key differentials and obtain the key by matching with precomputations. Concerning the performance, these attacks are also just marginally faster than brute force. The attack on the full AES-192 has a time complexity of $2^{189.68}$, and the attack on AES-256 has a time complexity of $2^{254.64}$.

The second group of attacks follow the long-biclique paradigm. Consequently, these attacks differ greatly from the above independent-biclique attacks since they use interleaving related-key differentials to construct the bicliques. The key is recovered using meet-in-the-middle attacks on up to three rounds. Although the long-biclique approach looks promising, it can not be used to attack the full versions of AES. Thus, the long-biclique attacks presented in [BKR11] work only on up to eight rounds of AES-128 and up to nine rounds of AES-256. Moreover, their performance is still only marginally faster than a brute force attack. Additionally, the memory and data requirements are much higher compared to the independent-biclique attacks. The long-biclique attack on AES-128 has a time complexity of $2^{124.97}$, memory complexity of 2^{102} and data complexity of $2^{126.33}$, which is almost the entire codebook. The attack on AES-256 has a time complexity of $2^{253.1}$, data complexity of 2^{120} and very low memory requirements of 2^{8}.

9. Summary

In this part, we have presented three papers describing recent attacks on AES in the single-key attack scenario. For each attack, we introduced the theoretical background knowledge and explained the underlying techniques it uses. Furthermore, we gave a detailed description of each attack and covered the most important extensions or trade-offs that are possible.

The multiset attack continues the series of improvements made to the initial saturation attack which was published together with the specification of AES. The general concept of the attack is to take a set of plaintext-ciphertext pairs with specific requirements, guess some key bytes at the beginning and end of the cipher, and use a distinguisher to filter out wrong key guesses. In case of the multiset attack, two basic techniques are used: The first is *multiset tabulation*, which yields a distinguisher for four full rounds of AES. This distinguisher allows to enumerate all possible multisets after four rounds, by using only 24 parameter bytes as input. The second technique is called *differential enumeration* and aims to fix as many of these 24 parameters as possible to known values. Thus, this enhancement further reduces the amount of possible multisets of the distinguisher. Both techniques combined were used in an attack on 7-round AES-128 with a time and memory complexity of 2^{113} each.

In the chapter on *low data complexity attacks*, we covered attacks that aim for minimal data requirements. We presented multiple techniques and properties of AES that can be used to construct attacks with low time complexity and low memory requirements. Although the resulting attacks only allow to attack up to four rounds of AES-128, they can still be used as building blocks for more sophisticated attacks. One example for this is the *known-plaintext* attack on six rounds of AES-128 which we presented in Section 7.3.

With the biclique attack, we presented a technique that was originally created for hash functions. Bogdanov et al. were the first to modify this tool for the cryptanalysis of block ciphers and used it to attack AES. We gave a thorough explanation of the techniques used to construct bicliques and recover the encryption key. We also presented the first attacks on all variants of AES with the full number of rounds which are faster than brute force and use bicliques. However, the resulting attacks are only marginally faster than a standard brute force attack. Thus, they are theoretical attacks only.

Part III.

Analysis and Implementation

10. Analysis of Single-Key Attacks

One goal of this thesis was to find improvements on the attacks presented above. Since some of them, for example the low data complexity attacks, work only on AES-128, we concentrated our improvement efforts on this variant of AES. Moreover, since practical applicability of related-key attacks is disputed, and the existing attacks in this scenario are already quite good, we focused on the single-key attack model.

From the attack techniques presented in Part II, we derived our general hypothesis that combining techniques from LDC attacks with multiset and similar attacks should allow to improve existing attacks on AES. More specifically, we formulated the following two concrete hypotheses:

1. Combining an LDC attack with distinguishers like the balanced property, or multiset tabulation allows to create improved attack.
2. The known-plaintext attack on 6-round AES presented in Section 7.3 can be enhanced using differential enumeration.

In the course of this chapter we analyze each specific hypothesis and verify if it in fact delivers an improved attack on AES-128. In Section 10.1 we check Hypothesis 1 by taking a closer look at the balanced property. To check Hypothesis 2 in Section 10.2, we modify the 6-round attack from Section 7.3 to a chosen-plaintext attack and try to extend it to seven rounds by using a different LDC attack.

10.1. Combining LDC Attacks with Square-like Distinguishers

Low data complexity attacks are ideal to recover the full encryption key from just a few rounds of AES by knowing only a minimal amount of plaintexts and their corresponding ciphertexts. It is possible to construct LDC attacks for up to three rounds of AES such that the time complexity is still practical. To use these techniques for more than just a few rounds, other techniques are required to combine them with LDC attacks. For every such attack, an attacker needs to know the full input and output state for the rounds on which he applies the LDC attack.

One proposed method to achieve this are differentials. They allow to narrow down the possible states at the end or beginning of the differential to a fraction of the 2^{128} overall possible states. Of course, this is only valid for plaintext-ciphertext pairs that correspond to that differential. For the known-plaintext attack on 6-round AES-128 from Section 7.3, such a differential is used on the first four rounds to decrease the number of possible intermediate

states at the end of the differential to just 2^{64}. For each of these possible states, an LDC attack is applied on the last two rounds to recover the full encryption key.

Our idea is to use other distinguishers like the balanced property from the saturation attack (see Section 4.2) or the multiset tabulation technique (see Section 6). They too allow to retrieve certain information about intermediate states. If the distinguisher is placed at the beginning, then the distinguishing property can be checked two to three rounds before the end of the cipher. In this section, we analyze if such distinguishers provide enough information on the intermediate states to combine them with LDC attacks for a new attack on AES.

10.1.1. Balanced Property

The balanced property covers three rounds. At the beginning there are 256 plaintexts that differ only in a single byte which takes all possible values. After encrypting these plaintexts through three rounds, the `XOR`-sum over all values is zero. We can observe this property for every single byte separately and it holds with high probability for the correct encryption key only. Taking the `XOR`-sum over 256 values and having it result in zero basically means that each single bit is set to `1` an even number of times. This knowledge allows us to throw away a few impossible internal states. More specifically, for every bit of a single byte we have a sequence of 256 bits. There are $2^{256} \cdot 2^{-1} = 2^{255}$ sequences for which the `XOR`-sum results in zero. Thus, for a full byte we get $2^3 \cdot 2^{255} = 2^{258}$ possible values for a single byte which satisfies the balanced property. This is only a minor reduction in possible values and hence still leaves an unacceptable amount of possible states at the end of the balanced property. As a result, just relying on the balanced property is not feasible. Moreover, it does not admit to combine this technique with an LDC attack for a feasible attack on AES.

10.1.2. Multiset Tabulation

The multiset tabulation technique is basically a generalization of the balanced property. It uses 24 bytes to describe all possible sequences a single byte takes after encrypting it through four full rounds (compared to the actual 256 bytes for a single sequence). This yields 2^{192} possible sequences for a single byte. Since a state consists of 16 bytes and each byte has 2^{192} possible sequences, this yields $2^4 \cdot 2^{192} = 2^{196}$ states. Thus, there are still too many possible states left, and it is not feasible to perform an LDC attack for each of them. Moreover, a multiset does not describe the actual values of the sequence, but rather an optimized representation which makes it hard or even impossible to deduce the actual values of the intermediate state.

10.1.3. Summary

Since the main reason for these high numbers of possible states are the large amount of plaintexts used (2^8), it might be possible to reduce the number of possible intermediate states by reducing the amount of plaintexts. This does not work for the balanced property,

but might be possible for the multiset technique and similar methods. However, this would require to completely redefine the multiset technique.

Overall, none of these distinguishers reduces the amount of possible intermediate states enough to be usable in combination with an LDC attack. Thus, this disproofs Hypothesis 1 from above.

10.2. Attacking 7-Round AES-128 Using Differential Enumeration and LDC Attacks

Here, we describe an attack on 7-round AES-128 that works by combining the differential enumeration technique, developed for the multiset attack, with an LDC attack on three rounds. Since this attack is very similar to the 6-round known-plaintext attack presented in Section 7.3, the attack we describe here can also be seen as an extension of the 6-round attack to seven rounds. However, the difference is that the 7-round attack we present here is a chosen-plaintext attack.

The basic idea of our 7-round attack is to use an LDC attack on the last three rounds of the cipher. Thus, state $S_{4,\texttt{ARK}}$ of the 7-round AES-128 algorithm is the input to the LDC attack. Since this input is in the middle of the cipher, there are 2^{128} possible values for state $S_{4,\texttt{SR}}$ we would have to consider. To reduce the amount of possible states and calculate the values in advance, we use a truncated differential trail on rounds 2-5. The full attack is thus split into two phases: differential enumeration and the LDC attack. The purpose of the first phase is to reduce the possible values of the input to second phase (state $S_{4,\texttt{SR}}$), the LDC attack.

10.2.1. Differential Enumeration

The first phase of our attack borrows heavily from the differential enumeration technique we covered in Section 6.4 to get a smaller set of possible intermediate states in $S_{4,\texttt{SR}}$.

The differential we use to accomplish this is the same as the one used for the multiset attack, and it is shown in Figure 6.3. The input difference has only one active byte (i.e. byte 0) and all other bytes have zero difference. At the end of the differential, we have exactly the same requirement: only one active byte and all the others zero. Considering all possible 2^{128} input states of the differential, there are about $2^{120} \cdot 2^{15} = 2^{135}$ possible pairs that match the input difference[1]. Assuming that four rounds of AES behave like a random permutation, this differential has a probability of 2^{-120}. This means within 2^{120} randomly chosen pairs, about one pair is a right pair w.r.t. the differential. Hence, for the whole codebook there exist about $2^{135} \cdot 2^{-120} = 2^{15}$ right pairs.

Consequently, for a single right pair we know that the amount of possible differences in state S_4 (of the 7-round AES-128) is reduced to $2^{32} \cdot 2^{32} = 2^{64}$ possible combinations. Thus, we can use a difference distribution table to get about 2^{64} pairs whose actual values for state

[1] Consider a set of 2^8 plaintexts that differ only in a single byte which takes each value exactly once. These can be used to construct $\binom{2^8}{2} = 2^{15} - 2^7$ possible pairs with a difference only in this active byte.

S_4. Further, we can trace these values through round 4 and compute 2^{64} values for $S_{4,\texttt{ARK}}$. Since this state is the input to the LDC attack, we have only 2^{64} possible inputs (instead of 2^{128}) for which we have to run the LDC attack.

10.2.2. Adapting the LDC Attack

The original LDC attack on three rounds which we use here, requires nine known plaintexts and was covered in Section 7.2.4. For the original LDC attack, we had to initially guess bytes 0, 7, 10 and 13 from the last round key. Since we switched `AddRoundKey` and `MixColumns` in round 2 and 3, we were able to calculate the actual values for column 0 of state $S_{2,\texttt{ARK}}$. We calculated a 64-bit difference vector for $b_i = S^i_{2,\texttt{MC}}[0]$ and stored all encountered values in a hash table. The full difference vector was:

$$b_1 \oplus b_2, b_1 \oplus b_3, \ldots, b_1 \oplus b_9.$$

By performing similar calculations from the plaintext towards $S_{2,\texttt{MC}}[0]$, we got additional key candidates and filtered them with those stored in the hash table. We kept only key guesses we encountered in both computations. By repeating this three more times for the remaining columns, we obtained about 2^{32} key guesses for rk_0. For these remaining key guesses, we used exhaustive search to find the full encryption key. The time complexity of this attack is about 2^{40} encryptions, the data complexity is 9 known plaintexts, and the memory complexity is 2^{31} (see also Section 7.2.4 and Figure 7.5).

Now, for the 7-round attack, we use the same LDC attack on the last three rounds. However, we do not have to switch `MixColumns` with `AddRoundKey` on the last round. Since this is anyways the last round of the full 7-round cipher, there is no `MixColumns` operation in this round. Further, we decrease the required inputs states from 9 to 8. To compensate for this reduction, we build every possible difference from the bytes b_i:

$$b_1 \oplus b_2, b_1 \oplus b_3, \ldots, b_1 \oplus b_9, b_2 \oplus b_3, b_2 \oplus b_4, \ldots, b_8 \oplus b_9.$$

These $\binom{8}{2} = 28$ differences provide a 224-bit filter for the key guesses. In the original attack we had only a 64-bit filter and were left with overall 2^{32} key guesses to verify. Now, with the 224-bit filter only a single key guess should remain. The time complexity of the attack stays the same at about 2^{40} encryptions. However, the memory complexity is slightly increased to $28 \cdot 2^{32} \cdot 2^{-4} \approx 2^{33}$ 128-bit blocks.

10.2.3. The Full Attack

For the full attack, we place the differential from Section 6.4 at rounds 2-5 as shown in Figure 10.1. This results in at most 2^{64} actual values for the state $S_{4,\texttt{SR}}$ or equally for state $S_{4,\texttt{ARK}}$. The latter is the input state of our 3-round LDC attack. In the first phase, we filter the plaintexts for right pairs w.r.t. the differential in the following way:

1. Encrypt 2^{83} groups of 2^{32} plaintexts (2^{115} in total) using the oracle. Within each group, bytes 0, 5, 10 and 15 take all values and all other bytes are constant.

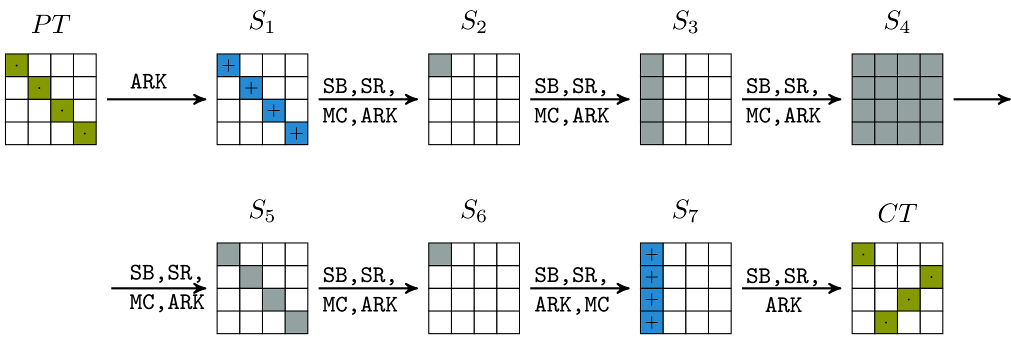


Figure 10.1.: Our attack on 7-round AES. ▪ indicate known bytes, ▪ shows calculated values using guessed key material and ▪ indicates active bytes of the differential.

2. Use the ciphertexts obtained in the previous step to construct pairs where only bytes 0, 7, 10 and 13 have non-zero differences (bytes marked with ▪ in Figure 10.1). To achieve this, we use a hash table for each group from Step 1 indexed by bytes 1-6, 8, 9, 11, 12, 14 and 15 of the ciphertext. Since we are able to construct $\binom{2^{32}}{2} = 2^{63}$ pairs with the correct plaintext difference for each group, and this is a 96-bit filter, we should have about 2^{50} of the $2^{83} \cdot 2^{63} = 2^{146}$ possible pairs left.

3. Guess four key bytes of the whitening key (rk_0) and partially encrypt the corresponding plaintext bytes.

4. Keep pairs with non-zero difference in $S_2[0]$, only. Since three bytes have to be equal, this leaves $2^{50} \cdot 2^{-24} = 2^{26}$ right pairs per key guess.

5. Guess four additional bytes of the last round key (rk_7), partially decrypt the corresponding ciphertext bytes and keep only pairs with non-zero difference in $S_6[0]$.

6. Thus, only $2^{26} \cdot 2^{-24} = 4$ right pairs per key guess are left.

At this point we know the following:

- There are only 2^{64} pairs of possible internal states in $S_{4,\texttt{SR}}$, and these possible states are independent from the key guesses we made.
- For each guess of the $2^{32} \cdot 2^{32} = 2^{64}$ key guesses we made during the filtering, there are four pairs (eight ciphertexts) we can use for an LDC attack together with the 2^{64} possible states for $S_{4,\texttt{SR}}$ as input.
- If we build all possible pairs with those eight ciphertexts in state S_6 during the LDC attack, we do not only get a difference in byte 0 but also get a difference in other bytes. This is because the difference trail we used for filtering the right pair is only valid for four pairs out of these $\binom{8}{2} = 28$ pairs. This allows us to use our LDC attack to recover the full key as this attack requires differences in at least one byte per column of state S_6.

- We can skip guessing four bytes from the last round key once per LDC attack since these coincide with the bytes we guessed when filtering the right pairs in the first phase of our attack.

There is one problem that makes the attack infeasible: For each LDC attack we take four pairs out of the 2^{64} possible input pairs and we have no knowledge on which of these states is the matching state for which of the eight ciphertexts under the unknown encryption key. This is a problem since the LDC attack considers the order of differences when creating the sequence of differences stored into a hash table. Thus, we have to consider every possible permutation of those 2^{64} states! This increases the time complexity of our attack drastically and thus makes it slower than exhaustive search.

10.2.4. Improving the Attack

To solve this problem, we tried to change the LDC attack such that it requires just one pair. For this purpose, we consider an LDC attack on three rounds that uses just one plaintext-ciphertext pair. Such an attack was found by Bouillaguet et al. in [BDF12]. The attack has time and memory complexities of 2^{80} and requires two known plaintexts. In [BDF12] the number of rounds for this attack is denoted by 2.5 since they assume the third and last round has no `MixColumns` operation. This fits our requirements since we place the LDC attack on the last three rounds.

For our attack we would need to slightly modify the differential enumeration phase that it only delivers one right pair w.r.t the differential per key guess. To achieve this, we reduce the number of groups of 2^{32} plaintexts from 2^{83} to 2^{81}. For each ciphertext pair, we execute the LDC attack together with one of the 2^{64} possible internal states. Without considering any specifics of the LDC attack, like which bytes need a non-zero difference for the attack to work, we first calculate the approximate overall time complexity of the full attack. The complexity results in $2^{64} \cdot 2^{64} \cdot 2^{80} = 2^{208}$ and is thus still far too high[2].

Unless, there exists a not yet discovered LDC attack on three rounds (two full and the last round) that requires only two plaintexts and has low time complexity and, this attempt to improve the time complexity of our 7-round attack does not work. Consequently, this also shows that Hypothesis 2 from above is wrong.

[2] We have one pair for each one of the $2^{32} \cdot 2^{32} = 2^{64}$ key guesses (2^{32} for the four bytes of rk_0, and 2^{32} for the last round key) from the filtering phase. Next, we have 2^{64} possible internal states and a time complexity of 2^{80} encryptions per LDC attack.

11. Implementation of the Biclique Attack on AES-128

The full biclique attack on AES-128, which we presented in Chapter 8, has a time complexity of $2^{126.15}$ which is only marginally faster than a standard brute force attack. Normally, when implementing a theoretical attack, the implementation introduces a certain overhead. This is mostly due to memory access and other operations that are not directly concerned with the attack. Since the difference in time complexity between the biclique attack and a brute force attack is very small, and the biclique attack is generally more complex, it is questionable if this difference remains.

Only very recently, Bogdanov et al. showed in [BKP+12] that a hardware implementation of a specifically modified biclique attack is in fact faster than a brute force attack. Compared to tailored hardware implementations, software implementations have less possibilities for optimizations since the software runs on a general purpose hardware. The upside of implementing an attack in software is that it does not require special hardware which makes it cheaper and more accessible. To verify the claim that the biclique attack is faster than brute force, we believe that in addition to a hardware implementation, also a comparison of software implementations should be performed. So, the goal for this part of this thesis was to verify if biclique attack is still faster than brute force when implemented in software.

Here, we present our software implementation of the biclique attack and show that the biclique attack is indeed also faster than a brute force attack in this scenario. For our analysis, we also created a highly optimized brute force attack in software. Similarly to the hardware implementation, we concentrated on targeting AES-128, however, we assume that for the other variants of AES the results will be similar. In Section 11.1, we take a look at the modified biclique attack by Bogdanov et al. and their hardware implementation. For our software implementations we use Intel's SSE extensions and the AES instruction set (*AES-NI*) which we introduce in Section 11.2. Furthermore, this section also covers a detailed description of how to achieve optimal performance with these instructions. Section 11.3 covers our implementations of the brute force attack and the biclique attack and their respective performances. Finally, we provide a discussion of the performance results of our implementations in Section 11.4.

11.1. Hardware Implementation by Bogdanov et al.

In [BKP+12] Bogdanov et al. describe their hardware implementation of the biclique attack and verify that it is also practically faster than a standard brute force attack. For this purpose, they modified the biclique attack to reduce its data complexity, and simplified the

matching phase where the key is recovered. The main difference to the original biclique attack is that they use only a 2-dimensional biclique that is applied on the first two rounds instead of the last three rounds. Furthermore, instead of choosing v for the key recovery as some part of an internal state on the last eight rounds, they choose v to be four bytes of the ciphertext. This simplifies the matching phase since it is basically just a forward computation from state S_3 to those four bytes of the ciphertext.

First, we describe the theory of the modified attack from [BKP+12]. In addition to the attack itself, we also give a comparison of each step to a brute force attack on AES. This provides the basis for the analysis of our software implementations of the biclique and brute force attack in Section 11.4. For this comparison, we evaluate the complexity of each step in S-Box computations since this is the most time-consuming operation of an AES round. Following the theoretical attack, we give an overview of the hardware implementation by Bogdanov et al. and its performance, especially the performance difference between the biclique attack and a brute force attack.

Note on figures: All figures in this section show the i-difference as ▪ and the j-difference as ☒.

11.1.1. Theoretical Attack

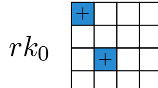
The modified biclique attack targets AES-128. The biclique is placed on the initial two rounds and the meet-in-the-middle matching to recover the encryption key is done on the remaining eight rounds. The key space is divided into 2^{124} groups of 2^4 keys each. These key groups are constructed from the initial cipher key rk_0 (whitening key) and do not overlap. The partitioning of the key space defines the dimension d of the biclique which is $d = 2$.

Biclique Construction

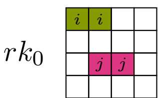
Each key group is constructed from a base key. We retrieve the base keys by setting the two least significant bits of $rk_0[0]$ and $rk_0[6]$ to zero (see Figure 11.1) and iterating the remaining bits of rk_0 over all possible values. Hence, bytes 0 and 6 of the base key have the binary value $b = b_0b_1b_2b_3b_4b_500$. To get the 16 keys within a key group, we enumerate differences $i, j \in \{0, \ldots, 3\}$ and add them to bytes 0, 4, 6, and 10 of the base key as follows (see also Figure 11.1):

$$\begin{aligned} rk_0[0] &= rk_0[0] \oplus i \\ rk_0[4] &= rk_0[4] \oplus i \\ rk_0[6] &= rk_0[6] \oplus j \\ rk_0[10] &= rk_0[10] \oplus j \\ &. \end{aligned}$$

Similar to the original attack, we use these key differences to construct two differential trails Δ_i and ∇_j. First, we define two key differences for rk_0. The key difference Δ_i^K covers all keys with $i \in \{1, \ldots, 3\}$ and $j = 0$. For the key difference ∇_j^K, we fix $i = 0$ and enumerate $j \in \{1, \ldots, 3\}$. Each of these key differences defines a differential trail, however,



Base key for each key group. Active bytes have the binary value $b = b_0b_1b_2b_3b_4b_500$.



Enumeration of keys within a group by differences i, j.

Figure 11.1.: Partitioning of the key space into groups. Partially zeroed bytes of the base key are indicated by ⊞.

to get more than one plaintext, we also add differences to the plaintext. The plaintext difference for each trail corresponds to the key difference of this trail (see Figure 11.2). So, for the Δ_i differential, we add the difference i to bytes 0 and 4 of the plaintext, and for the ∇_j differential, we add the j-difference to bytes 6 and 10. As shown in Figure 11.2, this also cancels the key differences introduced by the whitening key, thus, round 1 is the same for every differential. For each differential trail, we get three values for S_3, which we denote by S_3^i for the Δ_i trails, and S_3^j for the ∇_j differential.

By combining both differential trails (rightmost trail in Figure 11.2), we get the definition of the biclique which maps a set of 16 plaintexts to their corresponding values for the state S_3 with the keys of each key group. Note that contrary to the original biclique attack, both differential trails start with some non-zero difference and end in a non-zero difference. Moreover, the differential trails are not fully independent from each other because byte 12 of state S_3 is active in both trails. However, we are still able to combine both trails with the independent biclique construction technique (see Section 8.2.1) by considering byte 12 during the combination of both differential trails as described below.

Until here, we have constructed the biclique from two (almost) independent differential trails. This yields 16 values for S_3 by encrypting each one of the 16 plaintexts under the corresponding key from a key group. Thus, for each key group, we retrieve a different set of 16 values for S_3. This is called *precomputation phase*, and we have to calculate the full values for S_3 in an effective way. The algorithm is similar to the original biclique attack and has the following steps for each key group:

1. Perform the base computation by taking the all-zero plaintext and encrypting it with the base key of the group. Store the resulting value for S_3 (S_3^0). Moreover, store the value for rk_2 (rk_2^0) since we need to calculate the remaining round keys from it for the meet-in-the-middle matching.

2. Compute the values of S_3^i for every i by adding the key difference Δ_i^K to rk_0 and recomputing only the active bytes from the Δ_i trail. The inactive bytes are equal to the base computation. This yields three values for S_3^i.

3. Perform the same for every j and the ∇_j differential to get three values for S_3^j.

4. Combine the values for S_3^0 from the base computation, S_3^i from the Δ_i differential, and S_3^j from the ∇_j differential to get the 16 values for S_3 where each corresponds to the one

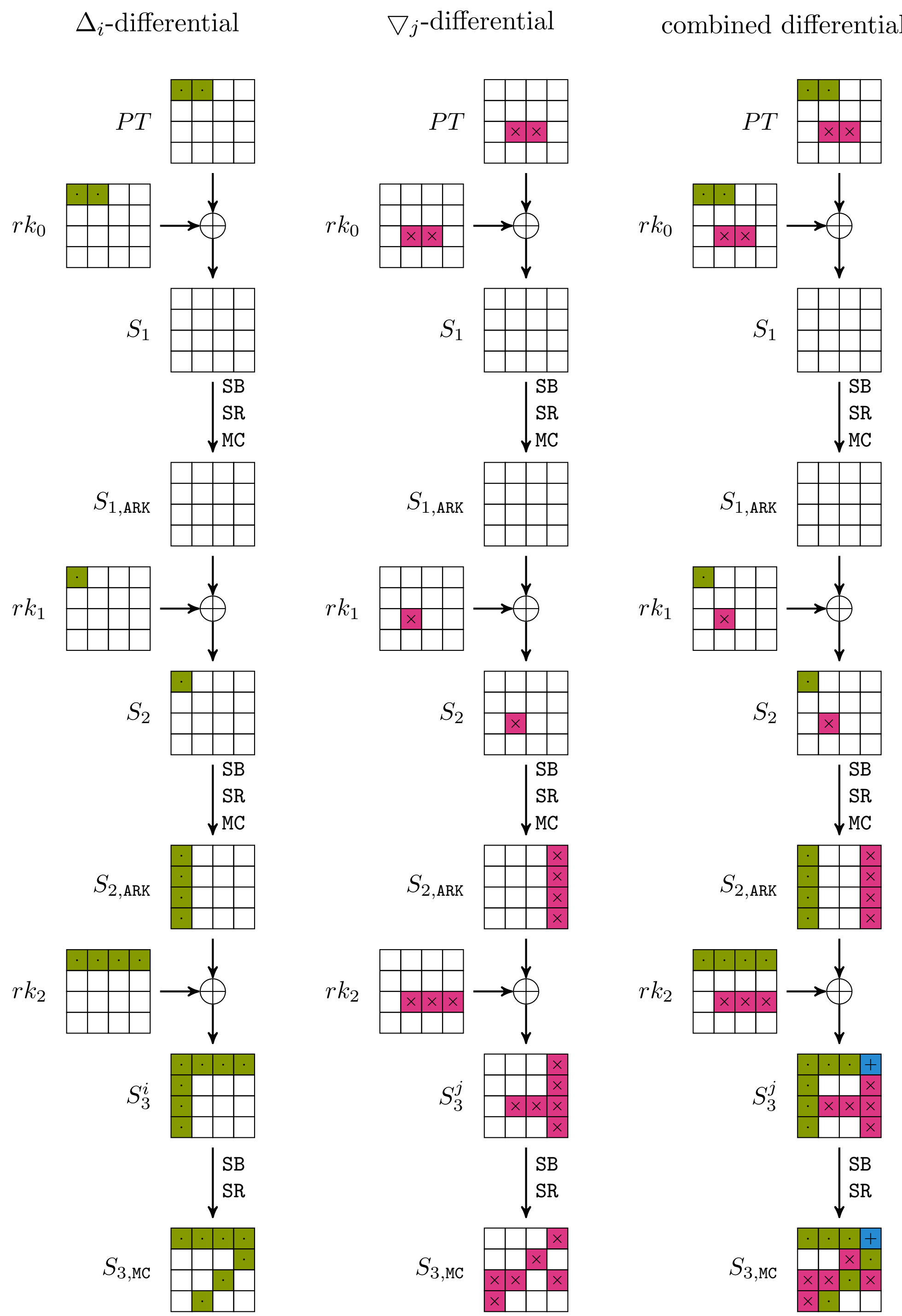


Figure 11.2.: The Δ_i, ∇_j differential trails, and the combined trail. ⊡, ⊠ indicate bytes to be recomputed in the i, j differential trail, respectively. Unmarked bytes are equal to the base computation, and bytes marked by ⊞ are active in both trails.

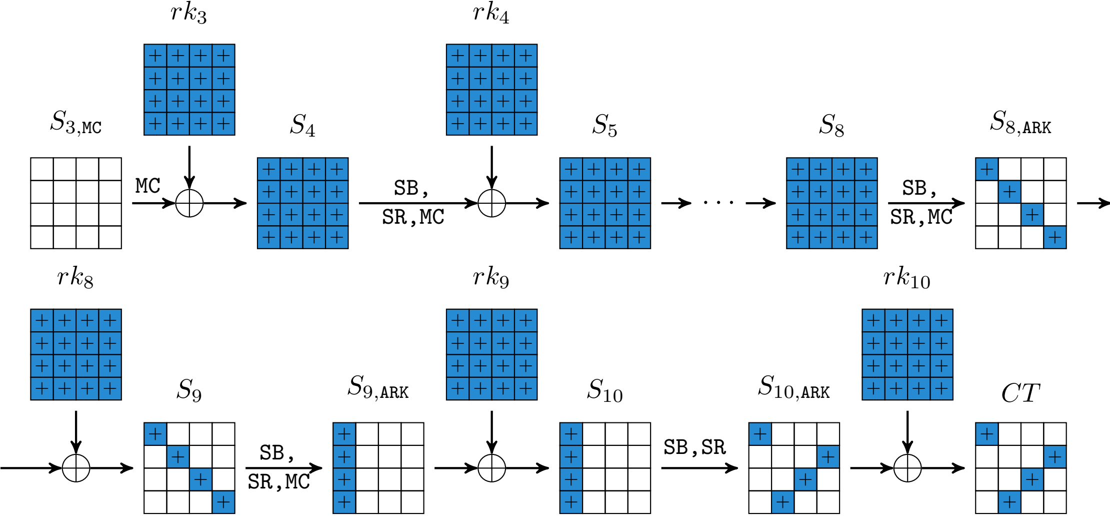


Figure 11.3.: Matching phase on rounds 4-10. Bytes that have to be recomputed for each key are marked with +.

of the plaintexts encrypted through the first two rounds under the corresponding round key. To get the values for rk_2, we just add the i, j differences to the corresponding bytes. This is possible because no S-Box has to be computed for the active key bytes in the key schedule. Since $S_3[12]$ is active in both differential trails, we consider this byte separately and retrieve its value by calculating $S_3^j[12] \oplus i$ or alternatively $S_3^i \oplus j$.

Thus, the advantage of this phase over a standard brute force attack is that we save S-Box computations by simply combining precomputed values for S_3. For a standard brute force attack, we perform 16 3-round AES computations to get 16 values for S_3. Here, we compute the full three rounds only for the base computation and then recompute only the required bytes for each differential trail. Moreover, we have to recompute the active bytes of each trail only three times for $i, j \in \{1, 2, 3\}$.

There is one possible improvement that enables us to save some S-Box evaluations in the matching phase: Instead of computing values for S_3, we can include the following `SubBytes` and `ShiftRows` operations and compute $S_{3,\texttt{MC}}$ instead. In the base computation, we compute the S-Box for bytes 5, 7, 9, and 11 (see Figure 11.2), and for each of the differential trails we compute the S-Box for their corresponding active bytes. This leaves only byte 12, which we have to compute for all 16 possible values of $S_{3,\texttt{MC}}$.

Key Recovery

As already described, the matching phase is simplified to match on four bytes of the ciphertexts. Thus, we first take the 16 plaintexts and retrieve the corresponding ciphertexts from the encryption oracle. From the output of the biclique (16 values for $S_{3,\texttt{MC}}$), we then compute only the required bytes to get the four ciphertext bytes. As shown in Figure 11.3, we compute the full states from S_4 up to $S_{7,\texttt{MC}}$. From this state on until the ciphertext, we only compute

the four active bytes. For the resulting four ciphertext bytes, we check if they match the corresponding ciphertext bytes retrieved from the encryption oracle. If they do, we have a possible key candidate which we have to verify with one additional plaintext-ciphertext pair. However, we should get only about one false key candidate per 2^{32} keys.

The advantage of this phase over brute force is that we avoid some S-Box computations by matching on only four bytes of the ciphertext instead of all 16 bytes.

Complexity of the Attack

Concerning the time complexity, Bogdanov et al. estimated the computation of the 16 values for S_3 using the biclique (precomputation phase) to be at most 0.3 full AES-128 encryptions. For the matching phase they estimated an effort similar to 7.12 AES-128 encryptions. Thus, the time complexity is

$$2^{124} \cdot (0.3 + 7.12) = 2^{126.89}$$

AES-128 encryptions to test for all possible keys. Thus, the biclique attack is about 55% faster than a brute force attack.

However, our calculation yields a slightly higher complexity as that given by Bogdanov et al.: To get an accurate estimate for the time complexity, we count the amount of S-Box evaluations that are required for the precomputation and matching phases. In this count we also include the S-Box evaluations required for the key schedule. Hence, one standard AES-128 round including the key schedule requires $16 + 4 = 20$ S-Box evaluations. Consequently, one full AES-128 encryption and key expansion requires 200 S-Box evaluations or 12.5 `SubBytes` operations.

In the precomputation phase we compute 16 values for state $S_{3,\texttt{MC}}$. These values are constructed from the base computation which is the encryption of the all-zero plaintext through the first two rounds under the base key of a key group. Additionally, we recompute the active bytes of the differential trails Δ_i, ∇_j where $i, j \in \{1, \dots, 3\}$. The total amount of S-Box is composed of:

- $2 \times 4 = 8$ S-Box evaluations for the key expansion up to rk_2.
- $2 \times 16 = 32$ S-Boxes for computing S_3^0 in the base computation.
- $2 \times 3 = 6$ S-Boxes for recomputing the active bytes in round 2 of both differential trails (for all i, j).
- For computing $S_{3,\texttt{MC}}$, we have to compute 4 S-Boxes for the inactive bytes, $4 \times 11 = 44$ for the active bytes of both differential trails in round 3, and 16 S-Box evaluations for computing byte 12.

This results in a total of 110 S-Boxes. Computing three AES rounds (incl. the key schedule) for 16 keys as it would be done in a brute force attack takes $16 \cdot 3 \cdot 20 = 960$. Thus, the precomputation phase is about the same as 0.11 3-round AES-128 encryptions or equally 0.55 full AES-128 executions.

The matching phase has to be performed for every one of the 16 keys in a key group. The total amount of S-Box evaluations per key is made up of:

- $5 \times 16 = 80$ S-Boxes for the full rounds 4-8. Note that in round 8 we only compute four output bytes of `MixColumns` but we still have to compute `SubBytes` for the full state before.
- $4 + 4 = 8$ S-Boxes for the last two rounds.
- Since we cannot skip any S-Box evaluations of the key schedule, we add another $8 \times 4 = 32$ S-Boxes.

This results in a total of $16 \times 120 = 1920$ S-Box evaluations per key group. Since a brute force attack for 16 keys on seven rounds requires $16 \cdot 7 \cdot 20 = 2240$ S-Box computations. Hence, the matching phase is about the same as 0.86 7-round AES-128 encryptions or equally 9.6 full AES-128 executions. The resulting time complexity of the full modified biclique attack is thus:

$$2^{124} \cdot (0.55 + 9.6) = 2^{127.34}.$$

Consequently, this biclique attack is about 37% faster than a standard brute force attack on AES-128.

The data complexity is defined by the biclique and thus, equals 2^4.

11.1.2. Hardware Implementation

For their evaluation of the biclique attack's performance, Bogdanov et al. implemented a standard brute force attack and the modified biclique attack from above on FPGAs (more specifically on the RIVYERA platform with 128 Xilinx Spartan-3 XC3S500 FPGAs). Furthermore, they realized both attack in application-specific integrated circuits (ASICs). Since the greatest bottleneck in an AES round is the S-Box, they implemented the S-Box with composite finite field inverters in $GF(((2^2)^2)^2)$ [HV06, SM03] which is currently one of the fastest ways to implement the AES S-Box.

Additionally, to achieve full utilization of every hardware component, they used *pipelining* as much as possible. The idea of a pipeline resembles that of a real-world assembly line by splitting an operation, for example an AES round or S-Box computation, into multiple stages. These stages are executed in parallel in a time-sliced manner. So, after the first stage of operation 1 is finished, its output is directly forwarded to stage 2, and while stage 2 is in progress, stage 1 for the next operation is done in parallel. This concept is also used in CPUs where a single instruction is split into micro-operations (μops).

For their brute force implementation, they used an 11-stage pipeline for one AES round from which they put ten in series to get the full AES-128 encryption algorithm. The small size of the full AES logic allowed them to put two encryption cores on each FPGA. Their reported performance for the brute force attack is 526×10^6 keys/s per FPGA.

For the biclique attack, they created two implementations: a conceptual attack which resembles the theoretical attack, and an inline implementation where the biclique output

states are computed on-the-fly. The reason for computing the biclique output on-the-fly was to reduce the memory access which was very high in the conceptual attack. This achieved a huge increase in performance, and thus, the biclique implementation with on-the-fly computation is faster than the brute force attack.

To perform the precomputations on-the-fly, they compute the active bytes of differential trails in parallel to the base computation. The output of the base computation is directly forwarded to the matching phase, and the bytes for the differential trails are temporarily stored in memory. The matching phase performs four serialized AES rounds and then computes the last two rounds for the four necessary bytes only. The verification of key candidates is performed offline, outside of the FPGAs on a standard CPU. Overall, they managed to fit four biclique engines on each FPGA and reported a performance of 945×10^6 keys/s per FPGA.

Overall, this hardware implementation of the biclique attack is about 44% faster than the brute force attack. However, the main reason for this is that the biclique attack is about a factor 2 smaller in size of its implementation than the brute force attack. Thus, Bogdanov et al. were able to fit four biclique cores onto a single FPGA. For the brute force attack, they could only put two cores onto an FPGA. Consequently, this comparison includes also the sizes of both implementations. To compare only the performance, we calculate the performance of a single core: The biclique core is able to test 236.25×10^6 keys/s and a brute force core tests 263×10^6 keys/s. Which shows that a single biclique core is actually slower than a brute force core. Hence, this implementation of the biclique attack is mainly faster because of its smaller size, which allows to fit more cores onto a single FPGA.

11.2. Background on AES-NI and Optimization

For our software implementation we chose to use the Intel AES instruction set *AES-NI* because it provides the fastest way to implement AES on a standard CPU and makes the implementation easier to compare. Moreover, AES-NI gave us an equal basis for the brute force and biclique implementations since the AES instructions take exactly the same amount of time for both implementations. In this section we provide an overview of the AES-NI technology and give an insight into which techniques we used to achieve optimal performance.

11.2.1. Intel AES Instruction Set

The AES-NI extensions were proposed by Intel and are described in [Int10]. The extensions are available on Intel CPUs starting with the x86 micro architecture called *Nehalem* (Intel Core family). Additionally, AMD included this extensions into their *Bulldozer* CPU architecture. Today, almost all current *Sandy Bridge* and *Ivy Bridge* CPUs support AES-NI which makes it a de facto standard instruction set for desktop computers.

Basically, AES-NI builds atop of Intel's "single instruction, multiple data" (SIMD) instruction extensions called *Streaming SIMD Extensions* (*SSE*) which, in its current version *SSE4.2*, provides 16 128-bit registers with dedicated integer and floating-point instructions [Int12, Chapters 9-13]. The idea of SIMD is to process a vector of data with a single

instruction. A simple example is the addition of four independent 32-bit values (a_1, a_2, a_3, a_4) with four other 32-bit values (b_1, b_2, b_3, b_4). In SSE it is possible to load both sets of four values into two 128-bit registers and use the SIMD addition instruction to perform four 32-bit additions $(a_1 + b_1, a_2 + b_2, a_3 + b_3, a_4 + b_4)$ at once.

For the AES-NI instruction set, Intel integrated certain operations for AES directly into the hardware, thus, making them faster than any pure software implementation and providing more security against timing attacks [Int10]. Overall, AES-NI adds the following operations for key schedule, encryption and decryption:

`AESENC`, `AESDEC` performs one full encryption or decryption round, respectively. Both operations take two operands: a round key and an input state. The round key operand can also be a memory location.

`AESENCLAST`, `AESDECLAST` performs the last encryption or decryption round. The operands are the same as for `AESENC`.

`AESKEYGENASSIST` computes the `SubWord` and `RotWord` operations required for the key schedule. This is a 3-operand instruction where two operands contain the input values and the result is stored into the third operand. The full algorithm for this instruction is shown in Listing 11.1.

`AESIMC` performs `InvMixColumns` on the given 128-bit register and stores the result to another 128-bit register.

A full description of all instructions can be found in [Int12, Volume 2].

```
1  AESKEYGENASSIST(uint32 SRC[4], uint32 DEST[4], uint8 RCON)
2  {
3      uint32 t0, t1;
4
5      t0 = SRC[1];
6      t1 = SRC[3];
7      DEST[0] = SubWord(t0);
8      DEST[1] = RotWord(SubWord(t0)) ^ (uint32)RCON;
9      DEST[2] = SubWord(t1);
10     DEST[3] = RotWord(SubWord(t1)) ^ (uint32)RCON;
11 }
```

Listing 11.1: The algorithm of the `AESKEYGENASSIST` instruction. `SRC` is a 128-bit input register, `RCON` a 8-bit constant also given as input, and `DEST` is the 128-bit output register.

11.2.2. Optimizing for Performance

Optimizing software implementations for high performance requires some background knowledge on how CPUs operate. Here, we give a short overview of the basic concepts used in

todays CPUs, and how to gain better performance by using them properly. Most of this chapter is based on the software optimizations resources provided by Agner Fog in [Fog12b] and [Fog12c] which provide a good introduction into software optimization for C++, C, and Assembly.

Measuring performance in seconds or μseconds is not useful, because to reproduce a result, a CPU with the same clock frequency and architecture would be required. Thus, it is better to measure performance in CPU cycles which are independent of clock cycle and only depend in actual CPU architecture. Moreover, for most of the basic instructions the amount of cycles required for a instruction is the same throughout multiple architectures.

Memory Access

The most important and first step in optimization for speed is to reduce memory access, because fetching a value from memory requires the CPU to access external resources (RAM) outside the core and wait for the value to be ready. To reduce the time required for memory access, CPUs have multiple levels of cache which keep recently fetched values from the RAM inside the CPU core. Thus, code that requires frequent access to only a small segment of memory is still very fast because that specific area of the RAM is fetched from the CPU cache. This idea is also used for table lookup implementations of the AES S-Box. Furthermore, cache management is done fully automatically by the CPU itself, so there is no need for the developer to do anything.

Out-of-order Execution

Another technique CPUs perform automatically is *out-of-order execution*. A good example where this technique is useful is loading a value from memory into a register. Instead of waiting for the value to be loaded, out-of-oder execution enables the CPU to perform other operations that do not require this value. So, the CPU basically has the ability to reorder the instructions of the program such that the outcome is the same, but a minimal amount of CPU cycles is wasted with doing nothing.

In this context it is also worth to note that the CPU is able to rename registers as required. For example, if the same register is used for two independent pieces of code, the CPU can basically perform one of the pieces with a different (unused) register. This increases the usage of out-of-order execution. However, out-of-order execution only works to a certain extend, thus, compilers still take an important role in the optimization process.

Multiple Execution Units

Even before multi-core CPUs, a single CPU core already contained multiple *execution units*. Thus, a single CPU could do multiple things at the same time as long as they where independent from each other and required different execution units. Today, a standard CPU has at least one floating point addition unit, one floating point multiplication unit, and two Arithmetic Logic Units (ALUs) for integer operations. As a result, in a single clock cycle a CPU could perform a floating point addition, a multiplication, and two integer additions

Instruction	Nehalem		Sandy Bridge	
	Duration	Throughput	Duration	Throughput
AESENC, AESDEC, AESENCLAST, AESDECLAST	5	2	8	4
AESIMC	5	2	2	2
AESKEYGENASSIST	5	2	8	8

Table 11.1.: The performance of all AES-NI instructions on Nehalem and Sandy Bridge architectures as measured in [Fog12a]. The "Throughput" columns list the reciprocal throughput.

at the same time. Furthermore, there is also at least one dedicated memory read, and one memory write unit. This is especially important, because memory read and write operations can thus be done in parallel to other operations. So, in a program with a low amount of memory operations the memory operations are negligible because they can be done in parallel to other operations on ALUs or floating point units. Naturally, having multiple execution units also increases the impact of out-of-order execution.

Pipeline Utilization

One CPU feature which we already introduced in the context of the biclique hardware implementation is *pipelining*. This is especially important for AES-NI and some SSE instructions since these instruction normally take more than one CPU cycle. Due to the CPUs ability to split a single instruction into multiple μops, pipelining allows the CPU to start processing an SSE instruction before the previous has finished. Obviously, this only works if both instructions are independent from each other. To create highly optimized implementations, it is thus important to know the exact amount of CPU cycles an instruction takes (*duration*), and how many cycles have to pass until the next instruction can be scheduled (*reciprocal throughput*).

11.2.3. AES-NI Specific Improvements

Since our implementations are mostly concerned with AES, we take a closer look on the performance of the AES instructions. Table 11.1 lists the duration and reciprocal throughput for the AES instructions on Nehalem and Sandy Bridge architectures. Since Westmere is just a 32nm die shrink of Nehalem and Ivy Bridge is a 22nm die shrink of Sandy Bridge, the performance on these architectures is equal to Nehalem and Sandy Bridge, respectively.

From the table, we can clearly see that in order to use the pipeline effectively, we have to perform multiple independent encryptions in parallel. This allows us reduce the amount of cycles for the overall implementation because the instructions overlap each other. For instance, if we perform four independent `AESENC` instructions in parallel on a Nehalem CPU, it would require 5 cycles until the first operation is finished, and after that, every second cycle another operation finishes. So, in total it requires only eleven cycles for those four operations

to finish instead of 20 cycles if we do not parallelize them. Interestingly, the AES instructions are slower in the newer Sandy Bridge architecture. To minimize the impact of this slower instructions on this architecture, we can performing more independent instructions in parallel (e.g. 8). Naturally, this is limited by the amount of available 128-bit registers.

For parallel execution, it is further important to consider the performance of SSE instructions since they also of require more than one CPU cycle. Moreover, the reciprocal throughput plays an important role. On the Nehalem architecture it is for example possible to execute three SSE `XOR` instruction (`PXOR`) in a single cycle. However, this works only if both operands are 128-bit registers. If one of the operands is a memory address, then only one `PXOR` instruction can be executed per cycle. Thus, when we have to `XOR` the same value (stored in memory) to multiple other values the intuitive way to implement this to use a memory operand as shown in Listing 11.2. However, this can be improved by first loading that value from memory to a register and then computing `XOR` with that register as shown in Listing 11.3. Since there is one dedicated execution unit for loading values from memory, this memory operation is negligible.

```
1 pxor (%rsp), %xmm0
2 pxor (%rsp), %xmm1
3 pxor (%rsp), %xmm2
4 pxor (%rsp), %xmm3
```

Listing 11.2: Intuitive approach.

```
1 movdqa (%rsp), %xmm8
2 pxor %xmm8, %xmm0
3 pxor %xmm8, %xmm1
4 pxor %xmm8, %xmm2
5 pxor %xmm8, %xmm3
```

Listing 11.3: Improved code.

A detailed listing with the performance of all instructions on various CPU architectures is given in [Fog12a].

On-the-fly Key Expansion

As mentioned before, memory operations take a lot of time until the value is ready, especially, if the cache is not large enough to hold the whole data that is required. For our brute force attack and also the biclique attack, we have to perform the key schedule for every key we test. Compared with standard implementations of AES where the key practically never changes or at least changes only rarely, this introduces additional computations. Thus, we analyzed the impact of two approaches for computing the key schedule: on-the-fly computation or precomputation. The precomputation variant is the standard approach to compute the key schedule. At first, we execute the full key schedule, then, store the round keys to memory, and finally run the full encryption algorithm. This way, we have additional memory operations for writing the round keys to memory and loading them again afterwards.

The on-the-fly approach tries to minimize memory access, by keeping the current round key in a register and computing the next round key shortly before it is required. We basically alternate between one key schedule round and one encryption round where the just computed round key is used. Listing 11.4 shows this for four keys in parallel. Since we test multiple keys in parallel, the amount of required memory operations depends on the number of keys.

```
int i;
uint128 keys[4], states[4], tmp;

/*
 * Full 9 rounds with on-the-fly key expansion
 * and 4 keys in parallel.
 */
for (i = 0; i < 9; i++) {
    key_schedule_round(keys[0], tmp);
    key_schedule_round(keys[1], tmp);
    key_schedule_round(keys[2], tmp);
    key_schedule_round(keys[3], tmp);
    encrypt_round(states[0], keys[0]);
    encrypt_round(states[1], keys[1]);
    encrypt_round(states[2], keys[2]);
    encrypt_round(states[3], keys[3]);
}
```

Listing 11.4: On-the-fly key expansion for four keys in parallel.

For testing four keys in parallel it is possible to create a memory-less implementation since there are enough register available. When testing more than that, for example eight keys in parallel, we require memory operations regardless of using the on-the-fly or precomputation approach. The reason for this is that for computing one key schedule round, we require at least one additional temporary 128-bit register. Our evaluations showed that in most cases the on-the-fly computation is faster than the precomputation approach.

Reducing `AESKEYGENASSIST` Instructions

To increase the performance for the key schedule even further, we optimized the usage of the `AESKEYGENASSIST` instruction. As shown in Listing 11.1 this operation computes `SubWord` and `RotWord` for two 32-bit words. Normally, we would just use one of those results when running the key schedule for a single key. However, since we execute the key schedule for four or more independent keys in parallel, we can use this property to half the amount of `AESKEYGENASSIST` instructions. As this is the most time expensive operation of the key schedule (the other operations are just byte-shuffle and `XOR` instructions), this yields a great performance boost.

Nevertheless, this improvement adds a few additional byte-shuffle and `XOR` instructions to assemble the input for `AESKEYGENASSIST` and extract the resulting output values. Despite of this additional instructions, this modification still improves the key schedule. Especially on the Sandy Bridge architecture the gain is higher, because the AES instruction takes 8 cycles. Overall this improvement saves about 9 cycles on Nehalem and about 17 cycles on Sandy Bridge CPUs. For the full code of the key schedule see Appendix A.

AVX Extensions

Starting with the Sandy Bridge CPU generation, the *AVX extension* provides even larger registers that are 256 bits wide. Furthermore, the data path has been extended from 128 bits to 256 bits, thus, AVX allows most of the SSE instruction to become non-destructive, because before AVX most SSE instructions wrote their result to one of the input registers. AVX introduced new versions for almost every SSE instruction which accept an additional operand where the result of the instruction is stored without altering the input registers.

For our implementations we did not use the 256-bit registers because the AES instructions only operate on the lower 128 bits and ignore the upper 128 bits of those registers. However, we used the non-destructive instructions where possible to safe most of the `mov` instructions.

11.3. Software Implementation

For comparison and to find the optimal implementation, we created two sets of implementations for the brute force attack and the biclique attack:

- C implementations with the use of *intrinsic functions*.
- Assembly (ASM) implementations in the *x86 assembly* language to avoid any suboptimal influences by a compiler.

Intrinsic functions are a set of functions which are directly translated to their corresponding CPU instruction by the compiler. This allows us to use the AES-NI instructions within a C program and, moreover, gives us access to other SSE instructions. The idea for our implementations in C is to use the compiler's optimization routines and at the same time avoid memory access by using SSE instructions as much as possible. This way, we do not have to care about the amount of available 128-bit registers because the compiler takes care of that. For the assembly implementations, we had full control over the optimization process which allowed us to reduce the memory access as much as possible. Our goal for both approaches was to achieve very high utilization of the CPU pipeline.

11.3.1. Test Environment

Since AES-NI performs differently on Nehalem and Sandy Bridge architectures, we tested our implementations on two test systems:

- **Westmere:** MacBook Pro with Intel Core i7 620M, running Ubuntu 11.10 and gcc 4.6.1
- **Sandy Bridge:** Google Chromebox with Intel Core i5 2450M, running Ubuntu 12.04 and gcc 4.6.3

On both systems we tested our implementations without running X11, and for the C implementations we used the following compiler flags: `-march=native -O3 -finline-functions -fomit-frame-pointer -funroll-loops`.

For measuring the CPU cycles, we used Intel's *time stamp counter* which can be read with a specific instruction and returns the number of elapsed cycles since its last reset. To measure the performance of our implementations, we read the time stamp counter on each iteration of our main loop and stored the values into a buffer. At the end, we took the median of all samples and calculated the CPU cycles per encrypted byte (cycles/byte). This is the unit for all measurements we give in the remainder of this chapter. For more details on the time stamp counter and sample code, see Appendix B.

11.3.2. Brute Force Reference Implementation

The implementation of our reference brute force attack is quite straight forward. Although, to find the best implementation, we evaluated various approaches. The basic variant tests eight keys in parallel which means that in the main loop we perform key schedule and encryption for eight keys at once. For the intrinsics variant, we compute the key schedule on-the-fly. Since we require at least one temporary register per key for one round of the key schedule, and there are only 16 128-bit registers, we do not have enough registers to avoid all memory read and write operations.

The alternative to this is to perform the full key schedule first, store the round keys in memory and then perform the full encryption. We used this approach for the assembly implementation to verify our tests that the on-the-fly approach is faster in most cases.

Since the memory operations have a large influence on a programs performance, we also created a fully memory-less implementation of the brute force attack. To do this, we reduced the amount of keys we test in parallel from eight to four. Thus, we have enough registers available to hold the current states and expand the round keys on-the-fly.

The full performance measurements for all implementations are shown in Table 11.2. The 8x variants test eight keys in parallel but require memory access, and the 4x variants test four keys in parallel without any memory access. The table shows nicely that the memory-less implementation can in fact be faster under certain circumstances. Overall, the performance of a implementation depends highly on the duration of the AES instructions. On the Sandy Bridge architecture, the instructions take longer, thus, testing only four keys in parallel does not utilize the CPU pipeline optimally.

Interestingly, the 8x C implementation is the overall fastest on the Sandy Bridge architecture. This can be explained by compiler optimizations since the compiler rearranges the instructions to use possibly few CPU cycles. Note that the 8x C and ASM implementations are not directly comparable since in the assembly implementations we run the full key expansion before the encryption and in the C implementation we execute the key schedule rounds on-the-fly.

If AVX is available, it additionally allows to safe some `mov` instructions because the non-destructive SSE instructions can be used. However, this has no drastic impact on the results. Note that our C implementations automatically us the AVX instructions if they are available on the target architecture.

Approach	**Westmere**	**Sandy Bridge**
C, 4x	3.09	3.80
ASM, 4x	**3.00**	3.80
ASM-AVX, 4x	-	3.80
C, 8x	3.86	**3.45**
ASM, 8x	3.40	3.95
ASM-AVX, 8x	-	3.93

Table 11.2.: Performance measurements for the various software implementations of the *brute force attack*. All values are given in cycles/byte.

11.3.3. Biclique Implementation

Instead of implementing the original biclique attack in software, we chose to implement the modified biclique attack by Bogdanov et al. as covered in Section 11.1. The main reason for this is the low data complexity of the attack with only 16 plaintext-ciphertext pairs. Additionally, the matching phase of the attack is simpler than in the original attack since it is performed on four ciphertext bytes. This modifications allowed us to implement the attack with almost no memory operations. The only difference from the theoretical attack to our implementation is that we compute the biclique only to state S_3 and not $S_{3,\texttt{MC}}$ due to limitations of AES-NI.

Since the biclique attack considers groups of 16 keys, our main loop consists of three steps:

1. Precompute values for S_3^0, rk_2 and active bytes of S_3^i, S_3^j.
2. Combine the precomputed bytes to 16 values for the full state S_3.
3. Encrypt the remaining rounds and match with the ciphertexts from the encryption oracle.

Since we have to compute five full encryption rounds and two reduced rounds for Step 3, this is the most time expensive step. For this reason, we concentrated on improving the performance of this step (matching phase) by creating two variants, similar to the brute force attack. For the first variant, we match four keys in parallel (4x) and use a minimal amount of memory operations. For the second variant we test eight keys in parallel (8x) to achieve higher utilization of the CPU pipeline, but risk more memory accesses. For both variants we compute the key schedule on-the-fly because it yields faster code.

In general, our implementations of the biclique attack are very close the theoretical attack. However, since we use Intel's AES instructions, it is not possible to recompute only the required bytes in every case. This is due to the fact that the AES instructions only operate on full 128-bit states and there are no instructions for single bytes. Thus, we often compute one round for the full state although we would only need to compute it for a few bytes. This is mainly the case in the matching phase where we compute the full round 8 instead of computing `MixColumns` the four required bytes only.

Approach	Westmere	Sandy Bridge
C, 4x	2.71	**3.18**
ASM, 4x	**2.61**	3.24
ASM-AVX, 4x	-	3.21
C, 8x	3.41	3.39

Table 11.3.: Performance measurements for the various software implementations of the *biclique attack* given in cycles/byte.

The limitations of AES-NI are also the reason why we did not extend the precomputation phase to $S_{3,\texttt{MC}}$ as proposed for the modified biclique attack. Let us consider this in more detail: The fastest way to compute `SubBytes` for a full state using AES-NI is to use `AESENCLAST` with an all-zero round key and apply `InvShiftRows`[1] afterwards. Since `AESENCLAST` alone has the same performance as `AESENC`, computing only `SubBytes` would take more time than to just compute a full round. Hence, we start the matching phase with state S_3 and compute the full round 3 for every key of the group.

Overall, for the full matching phase (4x and 8x variants) we compute six full rounds with `AESENC` and then compute the remaining two rounds as reduced rounds. For a reduced round we collect the required bytes of four states into one 128-bit register and use again the AES instructions to compute the round. Therefore, the sole advantage of the matching phase over brute force is that the last two rounds are computed for the required four bytes only.

Similar to the brute force implementations, we created the 4x and 8x variants in C with the use of intrinsics. Further, we implemented the 4x variant directly in x86 assembly. For the 4x assembly implementation we were able to create an almost memory-less variant which requires only four memory read and four memory write operations per key group. Moreover, pipelining and out-of-order execution take care that these operations are negligible. Therefore, the 4x assembly implementation yields the best performance on the Westmere architecture.

The full performance results for our implementations of the biclique attack are shown in Table 11.3. The 4x assembly implementation is slightly slower on the Sandy Bridge architecture due to longer execution times for the AES instructions. Generally, for the biclique attack the 4x variants yield better results on both architectures, although, the 8x variant is only slightly slower. When we compare the 4x variants, we nicely see the limitations of out-of-order execution. The C variant is the fastest because the compiler rearranges the instructions, although the compiler generates about the same amount of memory access and AES instructions as our assembly implementations use.

[1] This can be done by a simple SSE byte-shuffle instruction.

Attack	Westmere	Sandy Bridge
Brute Force	3.00	3.45
Biclique	2.61	3.18
Biclique faster by	13%	8%

Table 11.4.: Performance results for the biclique attack and the brute force attack.

11.4. Performance Results

Table 11.4 lists the best performance results for the biclique attack and the brute force reference attack on both of our test platforms. We can clearly see that the biclique attack is faster than the brute force attack in all scenarios. The average difference is 0.33 cycles/byte or 5.28 cycles per key. Thus, the biclique attack is on average about 10.5% faster than the brute force attack. Further, if we take the performance of the brute force attack as baseline for the time complexity of 2^{128}, our software implementation of the biclique attack has an average time complexity of about $2^{127.84}$. This strengthens the claim that the biclique attack is faster than a standard brute force attack, although, only very slightly.

To compute the performance in keys/s, we assume a recent CPU with four cores and a clock frequency of 2.80 GHz. Since our implementation uses only a single CPU core, we can run four biclique attacks in parallel. Hence, the implementation of the biclique attack achieves a performance of $268 \cdot 10^6$ keys/s, and for the brute force implementation we get $233 \cdot 10^6$ keys/s.

It is evident for these results that the difference in performance between brute force and biclique attack for the software implementations is far less distinct than for the hardware implementation. As mentioned before, the advantage of the biclique attack in hardware is mainly due to its smaller size. In the software setting, we completely ignore the code size of the implementations as they do not have an impact on the performance. For the software implementation of the brute force attack it is also worth to note that we here too used the improvements on AES-NI from Section 11.2.3. Thus, our brute force attack is not a completely standard brute force attack, but also a highly optimized implementation of this attack.

Moreover, there is one factor that actually reduces the performance of the biclique attack in software: AES-NI. As we have shown for the theoretical modified biclique attack, the precomputation phase requires only an equivalent of 0.55 AES-128 encryptions, and the matching phase requires about 9.6 AES-128 encryptions. Together, this results in an equivalent of 10.15 AES-128 encryptions for 16 keys. However, since the AES instruction set provides only instructions on full states, we loose some of this advantage. For example, we did not run the precomputations up to state $S_{3,\mathtt{MC}}$. Thus, our implementation is actually closer to 14 AES-128 encryptions. Nevertheless, AES-NI provided an equal basis for a fair comparison of both implementations.

To use all advantages of the biclique attack, it would be interesting to create a software implementation of the biclique attack without the use of AES-NI. This way, it might be possible to get closer the the theoretical 10.55 equivalent AES-128 encryptions.

12. Conclusions

This thesis was devoted to attacks on the widely used block cipher AES. As there have been multiple publications concerning new attacks on AES in recent years, we analyzed three of these attacks in full detail. For our analysis we concentrated on the single-key attack model. Single-key attacks are one of the most common attacks since they have a wide range of practical applications.

In Part I, we provided the basic preliminary information on the block cipher AES and its round transformations. Moreover, we introduced the two most common forms of cryptanalysis, namely linear cryptanalysis and differential cryptanalysis. Since this thesis focuses on attacking AES, we also provided a detailed description of the security mechanisms AES uses against common forms of cryptanalysis.

Part II concentrated on three prominent recent attacks. The first attack was the multiset attack. It is basically an improvement of the saturation attack that was published together with the initial specification for AES. A previous improved version of the saturation attack applied to 6-round AES with a time complexity of 2^{44}. For seven rounds, the saturation attack is only marginally faster than a brute force attack. The described multiset attack improved this and makes it possible to attack all AES variants with seven rounds with a time complexity of 2^{103} encryptions.

Next, we presented a completely different approach for attacks on AES, namely low data complexity attacks. This kind of attacks are less concerned with the amount of rounds that can be attack, but focus more on minimal data requirements. A great example for a low data complexity attack is the attack on two full AES rounds which we presented in Section 7.2.3. This attack requires only two chosen plaintexts. Moreover, it enables an attacker to retrieve the *full encryption key* with a time complexity of only 2^8 2-round AES encryptions.

The third recent attack on AES was the biclique attack by Bogdanov et al. This is the first attack to apply to the full versions of all three AES variants. More importantly, this attack is a single-key attack, which is less restrictive and more closely related to real-world applications than related-key attacks. Since bicliques were originally used for cryptanalysis of hash functions, we also presented the techniques developed by Bogdanov et al. to use bicliques for cryptanalysis of block ciphers. Although the resulting attacks apply to the full number of rounds, their time complexity is only marginally faster than a standard brute force attack. Thus, this attack is far from any feasible application.

Our contributions are collected in Part III, which covers our efforts for new, improved attacks in Chapter 10. For our analysis we focused on attacking AES-128 by combining techniques from multiple recent attacks, especially low data complexity attacks. Although our analysis of recent attacks did not yield new attacks or improvements, this provides a proper documentation of our efforts for future reference.

Moreover, Part III covered the second goal of this thesis, which was to verify the claimed time complexity of the biclique attack. Only very recently, Bogdanov et al. presented their hardware implementation of this attack to verify its claimed performance. We covered their results and theoretical attack in Section 11.1. Moreover, we showed that their implementation of the biclique attack gains its advantage over the brute force attack mainly through smaller size on hardware.

Since dedicated hardware is expensive and also allows more fine-grained improvements than software implementations, we aimed to verify the claimed performance in software. In Chapter 11, we provided a detailed description of how we implemented the biclique attack on AES-128 in software. In that context, we first gave an overview of software optimization techniques, as well as a thorough introduction into Intel's AES-NI extensions which we used for our software implementation. Further, we documented our extensive comparison between the implementations of the biclique attack and the brute force attack. Our highly optimized implementations of these attacks also reflect the claimed advantage of the biclique attack over exhaustive key search. On average, our biclique attack achieves a throughput of about 268×10^6 keys/s on a *standard desktop computer* which is about 10.5% of the performance our brute force attack on the same platform achieves.

Our intensive analysis of the biclique attack has shown that it gains its advantage over exhaustive key search in two main areas: The fast enumeration of intermediate states by a biclique, and the reduced matching in only four bytes. As shown, this does not threaten the security of AES since the attack is only slightly faster than exhaustive key search. In fact, we claim that the biclique attack itself is more closely related to a highly optimized brute force attack than to the other attacks presented in this thesis. The matching phase is a good indication for this as it save computations by simply reducing the amount of bytes on which it matches the ciphertexts from 16 to 4. This does not use any specific properties of AES and can easily be done for the brute force attack on any other block cipher.

Overall, none of the presented attacks really threatens the practical security of AES as the cipher still has a large enough security margin to make these attacks infeasible on the full number of rounds. As finding new and better attacks becomes harder, we believe that AES will remain a secure cipher in the foreseeable future. However, we also believe with the continuous increase in computing power, the invention of new technology and the improvements of attack techniques, the current security margin of AES will still be reduced further.

A. AES Key Schedule

In this chapter, we take a closer look on the AES key schedule algorithm and its improved implementation for the biclique attack and brute force attack in C using AES-NI.

A.1. Pseudo Code for Key Schedule Algorithm

```
KeyExpansion(uint32 CipherKey[Nk],
             uint32 ExpandedKey[4*(Nr+1)])
{
    uint32 tmp; // 32 bit unsigned integer (4 bytes)

    // copy over initial cipher key
    for (i=0; i<Nk; i++) {
        ExpandedKey[i] = CipherKey[i];
    }
    // expand round keys
    for (i=Nk; i<4*(Nr+1)) {
        // previous column
        tmp = ExpandedKey[i-1];
        if (i % Nk == 0) {
            // column 0 (first) of each round key
            tmp = SubWord(RotWord(tmp)) ^ Rcon[i/Nk];
        } else if ((Nk > 6) && (i % Nk == 4)) {
            // special case for column 4 (AES-256 only)
            tmp = SubWord(tmp);
        }
        ExpandedKey[i] = ExpandedKey[i-Nk] ^ tmp;
    }
}
```

Listing A.1: The key schedule given in C-pseudo code. The `^` sign denotes bitwise `XOR` and `%` denotes modulo.

Listing A.1 gives a C-like pseudo code description of the key schedule algorithm. Initially cipher key copied to output array (`Nk*4` bytes). Then we calculate the first column of each round key by taking column `i-1` (last one of the previous round key) and applying the S-Box to the rotated column. Afterwards we `XOR` it with the round constant for this key schedule round `Rcon[i/Nk]`. For all other columns we just take the previous columns and do not

apply anything to it. Finally in line 22 we add the same column of the previous round key to get the new value for the current column.

A.2. AES-NI Implementation

```
1  /*
2   * 1 round of AES-128 key schedule for a single key.
3   */
4  #define aesni_expand_key_rnd(rk, rcon, tmp0, tmp1) do { \
5      tmp1 = _mm_set1_epi16(0); \
6      tmp0 = _mm_aeskeygenassist_si128(rk, rcon); \
7      tmp0 = _mm_shuffle_epi32(tmp0, 255); /* 11111111 */ \
8      tmp1 = (__m128i)_mm_shuffle_ps((__m128)tmp1, \
9          (__m128)rk, 16); /* 00010000 */ \
10     rk = _mm_xor_si128(tmp1, rk); \
11     tmp1 = (__m128i)_mm_shuffle_ps((__m128)tmp1, \
12         (__m128)rk, 140); /* 10001100 */ \
13     rk = _mm_xor_si128(tmp1, rk); \
14     rk = _mm_xor_si128(tmp0, rk); \
15 } while(0)
```

Listing A.2: C-macro for one round of the AES key schedule using AES-NI intrinsic functions.

```
1  /*
2   * 1 round of AES-128 key schedule for 4 keys in parallel.
3   * Enhanced to use only 2 AESKEYGENASSIST instructions.
4   */
5  #define expand_key_4(rcon,k0,k1,k2,k3,t0,t1,t2,t3,t4,t5) do { \
6      t0 = _mm_set1_epi16(0); \
7      t1 = _mm_set1_epi16(0); \
8      t2 = _mm_set1_epi16(0); \
9      t3 = _mm_set1_epi16(0); \
10     t4 = (__m128i)_mm_shuffle_ps((__m128)k0, (__m128)k2, 204); \
11     t5 = (__m128i)_mm_shuffle_ps((__m128)k1, (__m128)k3, 204); \
12     /* keygenassist for 2 keys at once: bytes
13        15-12 for k2,k3 and bytes 7-4 for k0,k1 */ \
14     t4 = _mm_aeskeygenassist_si128(t4, rcon); \
15     t5 = _mm_aeskeygenassist_si128(t5, rcon); \
16     /* xor key bytes */ \
17     t0 = (__m128i)_mm_shuffle_ps((__m128)t0, (__m128)k0, 16); \
18     t1 = (__m128i)_mm_shuffle_ps((__m128)t1, (__m128)k1, 16); \
19     t2 = (__m128i)_mm_shuffle_ps((__m128)t2, (__m128)k2, 16); \
20     t3 = (__m128i)_mm_shuffle_ps((__m128)t3, (__m128)k3, 16); \
21     k0 = _mm_xor_si128(t0, k0); \
22     k1 = _mm_xor_si128(t1, k1); \
23     k2 = _mm_xor_si128(t2, k2); \
24     k3 = _mm_xor_si128(t3, k3); \
25     t0 = (__m128i)_mm_shuffle_ps((__m128)t0, (__m128)k0, 140); \
26     t1 = (__m128i)_mm_shuffle_ps((__m128)t1, (__m128)k1, 140); \
27     t2 = (__m128i)_mm_shuffle_ps((__m128)t2, (__m128)k2, 140); \
28     t3 = (__m128i)_mm_shuffle_ps((__m128)t3, (__m128)k3, 140); \
29     k0 = _mm_xor_si128(t0, k0); \
30     k1 = _mm_xor_si128(t1, k1); \
31     k2 = _mm_xor_si128(t2, k2); \
32     k3 = _mm_xor_si128(t3, k3); \
33     /* extract result of keygenassist and xor with k0-k3 */ \
34     t0 = _mm_shuffle_epi32(t4, 85); \
35     t1 = _mm_shuffle_epi32(t5, 85); \
36     t2 = _mm_shuffle_epi32(t4, 255); \
37     t3 = _mm_shuffle_epi32(t5, 255); \
38     k0 = _mm_xor_si128(t0, k0); \
39     k1 = _mm_xor_si128(t1, k1); \
40     k2 = _mm_xor_si128(t2, k2); \
41     k3 = _mm_xor_si128(t3, k3); \
42 } while(0)
```

Listing A.3: C-macro for one the key schedule round with four keys in parallel.

B. Measuring CPU Cycles on x86 CPUs

The simplest method the get performance measurements for software running on the x86 micro-architecture is to use the CPU's *time stamp counter*. This counter delivers the number of CPU cycles which have elapsed since it has last been reset. Measuring in CPU cycles provides an advantage over measuring performance in time because the measurements are independent from the CPU's clock frequency.

B.1. The Time Stamp Counter

The time stamp counter was introduces with Intel's first Pentium processors. The current counter value is a 64-bit unsigned integer value which can be retrieved using the CPU instruction `rdtsc`. Since the Pentium processors had only 32-bit registers, the values was stores in two registers. `EDX` holds the upper 32 bits and `EAX` the lower 32 bits. With the introduction of 64-bit CPUs Intel choose to keep compatibility with older CPUs and did not modify this behavior. However, on 64-bit processors the upper 32 bits of `RAX` are cleared.

Since todays processors have multiple CPU cores this instruction has to be used more carefully because the operating system has control over which process runs on which core. Moreover, the operating system can move the execution of a single process (also single-thread processes) from one core to another while the program is running. This presents a problem for programs that use the time stamp counter since each core has its own counter which is not synchronized across the other CPU cores. To overcome this problem, a program has to be locked to a single core. This was easily achieved for our programs because they had only a single execution thread. Thus, we could simply assign each program a single CPU core using the `taskset` provided by Linux.

Another set of features that influence the correctness of the `rdtsc` instruction are power-saving features. A CPU can for example dynamically reduce the clock frequency of each core if the CPU load is currently low. If any power-saving feature is enabled, the results of the time stamp counter are undefined. Hence, it is required to disable all power-saving features before the measurement process. On Linux, this can simply be done by executing the following command for each core: `echo performance > /sys/devices/system/cpu/cpu<id>/cpufreq/scaling_governor` (replace `<id>` with the zero-based number of the core).

B.2. Measurement Process

For our measurements, we collected a time stamp counter reading at the beginning of each iteration of the programs main loop. We used a buffer to store the last 1024 counter readings. At the end of our program, we took the median value of these 1024 samples and used it as the average performance. The advantage of taking the median value instead of computing the overall average value is that it yields a more stable measurement.

Since reading the current value from time stamp counter adds an additional instruction introduces a certain overhead. To eliminate the measurement overhead, we first measure the amount of cycles the `rdtsc` instruction requires and subtract this overhead in the end.

The full C-macro for taking performance measurements using CPU cycles is shown in Listing B.1.

```
#define CYCLE_SAMPLE_COUNT 1024

#define cpucycles(CYCLES) __asm__ \
    volatile("rdtsc; shlq $32,%%rdx; orq %%rdx,%%rax"
        : "=a" (CYCLES)
        :: "%rdx");

#define cpucycles_start(CYCLES) cpucycles(CYCLES)

#define cpucycles_stop(CYCLES, overhead) do { \
    uint64_t tmp; \
    cpucycles(tmp); \
    CYCLES = tmp - CYCLES - overhead; \
} while (0)

/*
    Measures overhead introduced by cpucycles and
    stores it into given variable "overhead".
*/
#define OVERHEAD_LOOPS 100
#define measure_overhead(overhead) do { \
    int i; \
    uint64_t cycles[OVERHEAD_LOOPS]; \
    for (i=0; i < OVERHEAD_LOOPS; i++) { \
        cpucycles_start(cycles[i]); \
        cpucycles_stop(cycles[i], 0); \
    } \
    median(cycles, OVERHEAD_LOOPS, overhead); \
} while (0)


/*
    Calculates the differences between consecutive
    elements of an array with cpucycle(...) samples.
    The resulting differences are stored inplace,
    thus replacing the sample values.
*/
#define cycle_diffs(cycles, len, overhead) do { \
    int i; \
    for (i=0; i < len-1; i++) { \
        cycles[i] = cycles[i+1] - cycles[i] - overhead; \
    } \
    cycles[len-1] = cycles[len-2]; \
} while(0)
```

Listing B.1: C-macro for measuring CPU cycles and calculating the measurement overhead.

List of Figures

2.1. The AES encryption function, which has a fixed length input (plaintext) and output of the same length (ciphertext). The encryption key is the same for encryption and decryption. 7
2.2. Representation of the AES state as 4×4 matrix. The numbers indicate the position of the corresponding byte in the cipher's input array. 8
2.3. Derivation of round keys from the key schedule's output words w_i (columns). 14
2.4. The round transformation of AES with all steps and notations used throughout this thesis. rk_i denotes the i-th round key. 16

3.1. A very simple round of an example cipher without any linear operations except the key addition. 23

4.1. Transformation of a Λ-set through three rounds of AES. ⊞ indicates active bytes of the Λ-set, ⊡ indicates bytes for which only the balanced property holds and □ represents constant bytes. 30
4.2. The last round of the basic 4-round saturation attack on AES. The numbering corresponds to the attack description in Section 4.2.2. 32
4.3. The partial decryption and filtering stage of the saturation attack on the last two rounds. 33

6.1. First step in the multiset construction. ⊞ indicates known bytes, ⊡ marks calculated values and ■ marks calculated differences. 43
6.2. Improved multiset construction by considering plaintexts. ⊠ marks bytes where we know the actual value for one plaintext only. 44
6.3. The maximum possible amount of differences per state of the 4-round expected-probability differential. 46

7.1. Visualization of the five key schedule relations for the first byte of the corresponding column of rk_i. 54
7.2. The two phases of the LDC attack on two rounds of AES, where the second round is without `MixColumns`. 57
7.3. The LDC attack on one full round using only one known plaintext. 58
7.4. The LDC attack on two full rounds using two chosen plaintexts. The time and memory complexity of this attack is as low as 2^8. 60
7.5. The LDC attack on three full rounds using nine known plaintexts. Both phases calculate a 64-bit difference vector for byte 0 of $S_{2,\texttt{MC}}$. 62

8.1. A d-dimensional biclique as shown in [BKR11]. 66
8.2. A long-biclique attack with a 2-dimensional biclique at the end [BKR11]. . . 71
8.3. Construction of key groups for the biclique attack on AES-128. The groups are defined on round key rk_8. 73
8.4. The differential trails for the biclique attack on AES-128. 74
8.5. Bytes that need to be recomputed between P_i and $\overrightarrow{v}$. 75
8.6. Bytes that need to be recomputed between S_j and $\overleftarrow{v}$. 75

10.1. Our attack on 7-round AES. ▣ indicate known bytes, ⊞ shows calculated values using guessed key material and ■ indicates active bytes of the differential. 85

11.1. Partitioning of the key space into groups. Partially zeroed bytes of the base key are indicated by ⊞. 89
11.2. The Δ_i, ∇_j differential trails, and the combined trail. ▣, ⊠ indicate bytes to be recomputed in the i, j differential trail, respectively. Unmarked bytes are equal to the base computation, and bytes marked by ⊞ are active in both trails. 90
11.3. Matching phase on rounds 4-10. Bytes that have to be recomputed for each key are marked with ⊞. 91

List of Tables

2.1. The AES S-Box table. The values are given in hexadecimal notation and a cell (x, y) holds S-Box(xy). 11

3.1. The partial difference distribution table for the AES S-Box. 22

5.1. A list of all recent attacks presented in this thesis. Additionally, three recent related-key attacks are also shown. CP, CC indicates *chosen* plaintext resp. ciphertext attacks and KP indicates *known* plaintext attacks. 40

7.1. A list of best low data complexity attacks from [BDD+10] and [BDF12]. . . . 52
7.2. Relations between columns of five consecutive round keys. 54

11.1. The performance of all AES-NI instructions on Nehalem and Sandy Bridge architectures as measured in [Fog12a]. The "Throughput" columns list the reciprocal throughput. 97
11.2. Performance measurements for the various software implementations of the *brute force attack*. All values are given in cycles/byte. 102
11.3. Performance measurements for the various software implementations of the *biclique attack* given in cycles/byte. 103
11.4. Performance results for the biclique attack and the brute force attack. 104

Listings

2.1. High level description of AES given in C-pseudo code as depicted in [DR02], where Nr is the number of rounds. 9

11.1. The algorithm of the AESKEYGENASSIST instruction. SRC is a 128-bit input register, RCON a 8-bit constant also given as input, and DEST is the 128-bit output register. 95
11.2. Intuitive approach. 98
11.3. Improved code. 98
11.4. On-the-fly key expansion for four keys in parallel. 99

A.1. The key schedule given in C-pseudo code. The ^ sign denotes bitwise XOR and % denotes modulo. 109
A.2. C-macro for one round of the AES key schedule using AES-NI intrinsic functions. 110
A.3. C-macro for one the key schedule round with four keys in parallel. 111

B.1. C-macro for measuring CPU cycles and calculating the measurement overhead. 115

Bibliography

[BBD+98] Eli Biham, Alex Biryukov, Orr Dunkelman, Eran Richardson, and Adi Shamir. Initial Observations on Skipjack: Cryptanalysis of Skipjack-3XOR. In *Selected Areas in Cryptography*, Lecture Notes in Computer Science, pages 362–376. Springer, 1998.

[BBS99a] Eli Biham, Alex Biryukov, and Adi Shamir. Cryptanalysis of Skipjack Reduced to 31 Rounds Using Impossible Differentials. In *EUROCRYPT*, Lecture Notes in Computer Science, pages 12–23. Springer, 1999.

[BBS99b] Eli Biham, Alex Biryukov, and Adi Shamir. Miss in the Middle Attacks on IDEA and Khufu. In *FSE*, Lecture Notes in Computer Science, pages 124–138. Springer, 1999.

[BDD+10] Charles Bouillaguet, Patrick Derbez, Orr Dunkelman, Nathan Keller, Vincent Rijmen, and Pierre-Alain Fouque. Low Data Complexity Attacks on AES. *IACR Cryptology ePrint Archive*, 2010:633, 2010.

[BDF12] Charles Bouillaguet, Patrick Derbez, and Pierre-Alain Fouque. Automatic search of attacks on round-reduced aes and applications. Cryptology ePrint Archive, Report 2012/069, 2012. `http://eprint.iacr.org/`.

[BDK+10] Alex Biryukov, Orr Dunkelman, Nathan Keller, Dmitry Khovratovich, and Adi Shamir. Key Recovery Attacks of Practical Complexity on AES-256 Variants with up to 10 Rounds. In *EUROCRYPT*, pages 299–319, 2010.

[Ber05] Daniel J. Bernstein. Cache-timing attacks on AES. Technical report, University of Illinois, Chicago, 2005.

[BG11] Alex Biryukov and Johann Großschädl. Cryptanalysis of the Full AES Using GPU-Like Special-Purpose Hardware. *IACR Cryptology ePrint Archive*, 2011:710, 2011.

[Bih94] Eli Biham. New Types of Cryptanalytic Attacks Using Related Keys. *J. Cryptology*, 7(4):229–246, 1994.

[BK09] Alex Biryukov and Dmitry Khovratovich. Related-Key Cryptanalysis of the Full AES-192 and AES-256. In *ASIACRYPT*, pages 1–18, 2009.

[BKP+12] Andrey Bogdanov, Elif Bilge Kavun, Christof Paar, Christian Rechberger, and Tolga Yalcin. Better than Brute-Force — Optimized Hardware Architecture for

Efficient Biclique Attacks on AES-128. In *Workshop records of Special-Purpose Hardware for Attacking Cryptographic Systems – SHARCS 2012*, pages 17–34, 2012. http://2012.sharcs.org/record.pdf.

[BKR11] Andrey Bogdanov, Dmitry Khovratovich, and Christian Rechberger. Biclique Cryptanalysis of the Full AES. In *ASIACRYPT*, pages 344–371, 2011.

[BP09] Joan Boyar and Rene Peralta. New logic minimization techniques with applications to cryptology. Cryptology ePrint Archive, Report 2009/191, 2009. http://eprint.iacr.org/.

[Bra09] Gregory V. Brad. *Algebraic Cryptanalysis*. Springer Verlag, 1 edition, 2009.

[BS91] Eli Biham and Adi Shamir. Differential Cryptanalysis of DES-like Cryptosystems. In *Proceedings of the 10th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '90, pages 2–21, London, UK, UK, 1991. Springer-Verlag.

[BW99] Alex Biryukov and David Wagner. Slide Attacks. In *FSE*, volume 1636 of *Lecture Notes in Computer Science*, pages 245–259. Springer, 1999.

[Can05] David Canright. A Very Compact S-Box for AES. In *Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, Lecture Notes in Computer Science, pages 441–455. Springer, 2005.

[CBW08] Nicolas Courtois, Gregory V. Bard, and David Wagner. Algebraic and Slide Attacks on KeeLoq. In *FSE*, volume 5086 of *Lecture Notes in Computer Science*, pages 97–115. Springer, 2008.

[DGV93] Joan Daemen, René Govaerts, and Joos Vandewalle. Weak Keys for IDEA. In *CRYPTO*, pages 224–231, 1993.

[DKR97] Joan Daemen, Lars R. Knudsen, and Vincent Rijmen. The Block Cipher Square. In *FSE*, Lecture Notes in Computer Science, pages 149–165. Springer, 1997.

[DKS10] Orr Dunkelman, Nathan Keller, and Adi Shamir. Improved Single-Key Attacks on 8-Round AES-192 and AES-256. In *ASIACRYPT*, pages 158–176, 2010.

[DR02] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES — the Advanced Encryption Standard*. Springer-Verlag, 2002.

[DR05] Joan Daemen and Vincent Rijmen. Probability distributions of Correlation and Differentials in Block Ciphers. Cryptology ePrint Archive, Report 2005/212, 2005. http://eprint.iacr.org/.

[DS08] Hüseyin Demirci and Ali Aydin Selçuk. A Meet-in-the-Middle Attack on 8-Round AES. In *FSE*, pages 116–126, 2008.

[FKL+00] Niels Ferguson, John Kelsey, Stefan Lucks, Bruce Schneier, Michael Stay, David Wagner, and Doug Whiting. Improved Cryptanalysis of Rijndael. In *FSE*, pages 213–230, 2000.

[Fog12a] Agner Fog. Instruction tables – Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. `http://www.agner.org/optimize/instruction_tables.pdf`, 2012. Accessed 2012-09-02.

[Fog12b] Agner Fog. Optimizing software in C++: An optimization guide for Windows, Linux and Mac platforms. `http://www.agner.org/optimize/instruction_tables.pdf`, 2012. Accessed 2012-09-02.

[Fog12c] Agner Fog. Optimizing subroutines in assembly language: An optimization guide for x86 platforms. `http://www.agner.org/optimize/instruction_tables.pdf`, 2012. Accessed 2012-09-02.

[Fou98] Electronic Frontier Foundation. *Cracking DES: Secrets of Encryption Research, Wiretap Politics and Chip Design*. O'Reilly Media, September 1998.

[GM00] Henri Gilbert and Marine Minier. A Collision Attack on 7 Rounds of Rijndael. In *AES Candidate Conference*, pages 230–241, 2000.

[HV06] Alireza Hodjat and Ingrid Verbauwhede. Area-Throughput Trade-Offs for Fully Pipelined 30 to 70 Gbits/s AES Processors. *IEEE Trans. Computers*, 55(4):366–372, 2006.

[Int10] Intel Corporation. Intel® Advanced Encryption Standard (AES) Instruction Set, White Paper. Technical report, Intel Mobility Group, Israel Development Center, Israel, January 2010.

[Int12] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation, March 2012.

[Kho10] Dmitry Khovratovich. *New Approaches to the Cryptanalysis of Symmetric Primitives*. PhD thesis, University of Luxembourg, 2010.

[KKS00] John Kelsey, Tadayoshi Kohno, and Bruce Schneier. Amplified Boomerang Attacks Against Reduced-Round MARS and Serpent. In *FSE*, pages 75–93, 2000.

[Knu92] Lars R. Knudsen. Cryptanalysis of LOKI91. In *AUSCRYPT*, Lecture Notes in Computer Science, pages 196–208. Springer, 1992.

[Knu94] Lars R. Knudsen. Truncated and Higher Order Differentials. In *Fast Software Encryption: Second International Workshop*, Lecture Notes in Computer Science, pages 196–211. Springer, 1994.

[Knu98] Donald E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms, 2nd Edition*. Addison-Wesley, 1998.

[KRS11] Dmitry Khovratovich, Christian Rechberger, and Alexandra Savelieva. Bicliques for Preimages: Attacks on Skein-512 and the SHA-2 family. *IACR Cryptology ePrint Archive*, 2011:286, 2011.

[KS99] John Kelsey and Bruce Schneier. Key-Schedule Cryptanalysis of DEAL. In *Selected Areas in Cryptography*, pages 118–134, 1999.

[KSW96] John Kelsey, Bruce Schneier, and David Wagner. Key-Schedule Cryptoanalysis of IDEA, G-DES, GOST, SAFER, and Triple-DES. In *CRYPTO*, Lecture Notes in Computer Science, pages 237–251. Springer, 1996.

[LDKK08] Jiqiang Lu, Orr Dunkelman, Nathan Keller, and Jongsung Kim. New Impossible Differential Attacks on AES. In *INDOCRYPT*, pages 279–293, 2008.

[Mat93] Mitsuru Matsui. Linear Cryptoanalysis Method for DES Cipher. In *EUROCRYPT*, Lecture Notes in Computer Science, pages 386–397. Springer, 1993.

[MRST09] Florian Mendel, Christian Rechberger, Martin Schläffer, and Søren S. Thomsen. The Rebound Attack: Cryptanalysis of Reduced Whirlpool and Grøstl. In *FSE*, Lecture Notes in Computer Science, pages 260–276. Springer, 2009.

[MVO01] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 5th edition, 2001.

[NIS81] NIST. *Guidelines for Implementing and Using the NBS Data Encryption Standard (Withdrawn FIPS PUB 74)*. National Institute of Standards and Technology, 1981.

[NIS01] NIST. *Specification for the Advanced Encryption Standard (AES)*. National Institute of Standards and Technology, 2001.

[Sch11] Martin Schläffer. *Cryptanalysis of AES-Based Hash Functions*. PhD thesis, Graz University of Technology, Austria, 2011.

[Sha49] Claude E. Shannon. Communication Theory of Secrecy Systems. *Bell Systems Technical Journal*, 28:656–715, 1949.

[SM03] Akashi Satoh and Sumio Morioka. Hardware-Focused Performance Comparison for the Standard Block Ciphers AES, Camellia, and Triple-DES. In *ISC*, Lecture Notes in Computer Science, pages 252–266. Springer, 2003.

[Wag99] David Wagner. The Boomerang Attack. In *FSE*, Lecture Notes in Computer Science, pages 156–170. Springer, 1999.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Ort, Datum/Place, Date

Unterschrift/Signature