FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Linear Cryptanalysis of Baby Rijndael and Implementation Side Channels of AES

by

Josef Kokeš

A dissertation thesis submitted to
the Faculty of Information Technology, Czech Technical University in Prague,
in partial fulfilment of the requirements for the degree of Doctor.

Dissertation degree study programme: Informatics

Department of Information Security

Prague, August 2021

Supervisor:
prof. Ing. Róbert Lórencz, CSc.
Department of Information Security
Faculty of Information Technology
Czech Technical University in Prague
Thákurova 9
160 00 Prague 6
Czech Republic

Abstract and contributions

In this dissertation thesis we study selected security aspects of AES, the Advanced Encryption Standard. Specifically, we approach the issue from three sides:

First, we are trying to verify whether AES is indeed resistant to linear cryptanalysis. It was designed as such, in accordance with the requirements as well as the modern cipher design, but the size and complexity of the cipher make it difficult to show definitively that the cipher is resistant in all possible situations. We solve this problem by performing the tests against Baby Rijndael, a reduced model of the cipher built along the same design principles. We show that the success of linear cryptanalysis depends on more variables than usually assumed and that some keys or some plaintexts are more susceptible to an attack than others, but overall the effort required far outweighs that of brute-force trying of all keys. That demonstrates that Baby Rijndael and in extension AES are indeed resistant to this form of cryptanalysis.

Then we consider side channels created by a particular implementation. While they are much less universal, only applying to those specific implementations rather than all of them, they are generally much more efficient in breaking the encryption and recovering either the key or the plaintext.

Our first side channel involves the execution environment where AES encryption is being performed. We propose an algorithm, suitable for the Intel Architecture, that can automatically detect that encryption is taking place by monitoring access to AES S-boxes, and due to the interactions between AES state, key and S-boxes, are able to recover both the key and the plaintext, in most implementations that use these S-boxes. We also discuss the options of achieving the same results with implementations not dependent on S-boxes, i.e. with AES-NI based implementations and with vector unit based bit-slicing implementations. While bit-slicing seems to be impossible to detect universally, AES-NI can definitely be monitored to recover both the data and the key.

Our second side channel deals with implementation errors on the side of developer. We use reverse engineering to analyze an existing application Drive Snapshot that makes use of AES, recovering its key generation process. We demonstrate that while the process makes use of strong cryptographic algorithms and adheres, for the most part, to current best

practices, the programming errors present in the code cause it to fail in a number of ways, not the least being the fact that 148 out of 256 bits of the AES key are revealed to the attacker, making a brute-force attack on the key much easier (still impossible, though). We also demonstrate other errors that simplify the attack on both the key and the password. Finally, we discuss the possible causes for these errors and ways the development process could be changed to prevent them in the future.

In particular, the main contributions of the dissertation thesis are as follows:

- Analysis of Baby Rijndael reveals that it is an AES model designed along the same lines as AES itself, which makes it suitable to various analyses which would be difficult or even impossible with full AES. It has already been used to perform tests of various cryptanalytic techniques, with varying results.
- Analysis of Baby Rijndael shows that there are heretofore unknown properties of linear cryptanalysis, such as the varying performance of differing linear approximations used. This leads us to believe that there is more to discover in this area, with possible implications on other ciphers.
- Despite these discoveries, the practical verification shows that linear cryptanalysis is not sufficiently strong to break Baby Rijndael with a computational complexity lower than brute force, even in the most advantageous contexts. That result tends to scale to AES itself and assure us of its resistance to linear cryptanalysis.
- We show that side channels created by implementations of AES can be powerful enough to make attacks feasible. We present an algorithm which enable the attacker who can control the execution flow of an application to automatically detect the use of AES and recover both the encryption key and the plaintext, as long as a software-based AES implementation is used. We also show that some hardware-assisted implementations can also be attacked under the right circumstances.
- We performed a security analysis of Drive Snapshot, particularly its key generation process. We discovered a number of vulnerabilities which were reported to the developer for fixing. We also analyzed the causes of these vulnerabilities and proposed steps to be taken by developers to avoid these vulnerabilities in future applications.

Keywords:
AES, Rijndael, Baby Rijndael, model, linear cryptanalysis, multiple approximations, side channel, dynamic analysis, key recovery, programming errors.

Acknowledgements

First and foremost, I would like to express my utmost gratitude to my dissertation thesis supervisor, prof. Róbert Lórencz. It was he who introduced me into the fascinating field of cryptanalysis, and it was he who kept encouraging me and supporting me every time I needed it throughout my studies. His experience and his willingness to share it guided me throughout obstacles that I would have though insurmountable otherwise. This thesis would never get finished without his support.

I am also deeply thankful to the staff of the whole Faculty of Information Technology, both the teachers who re-awakened in my the hunger for knowledge and the administrative personnel who enabled me to breeze through the normally annoying administrative aspects. In particular I would like to thank Mgr. Lenka Fryčová for her unending support of us PhD students throughout the study. Thank you!

My research has also been partially supported by the Ministry of Education, Youth, and Sport of the Czech Republic under the research program OP VVV MEYS funded project CZ.02.1.01/0.0/0.0/16_019/000 0765 "Research Center for Informatics".

Finally, my warmest thanks belong to my family members for their love and support in these difficult times.

Contents

List of Figures

List of Tables

List of Algorithms

Abbreviations

AES	Advanced Encryption Standard
API	Application Programming Interface
CBC	Cipher Block Chaining mode of encryption
CCM	Counter With CBC-MAC mode of encryption
CPU	Central Processing Unit
CT	ciphertext
CTR	Counter mode of encryption
DES	Data Encryption Standard
ECB	Electronic Codebook mode of encryption
GF	Galois Field
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
HW	Hardware
KDF	Key Derivation Function
MAC	Message Authentication Code
MMX	Multi Media Extensions
NIST	National Institute of Standards and Technology
OS	Operating System
PBKDF2	Password Based Key Derivation Function 2
PRF	Pseudo-Random Function
PRNG	Pseudo-Random Number Generator
PT	plaintext
RSA	Rivest-Shamir-Adleman Cryptosystem
SNFS	Special Number Field Sieve
SSE	Streaming SIMD Extensions
SSH	Secure Shell
SSL/TLS	Secure Socket Layer/Transport Layer Security
VM	Virtual Machine

Chapter 1

Introduction

With our ever increasing use of information technology and our growing dependence on electronic information, the need to protect this information from unauthorized use keeps gaining in importance. There are many methods for achieving this protection, each useful in its own specific area. We focus on one small part of this security, and that is the practical security of the cipher AES against an attacker who wants to recover the secret key used to encrypt some data packet and with this key, recover the original readable data.

1.1 Motivation and Background

At the core of our work stands cryptanalysis — the study of cryptographic system with the intention of discovering weaknesses that would allow the cryptanalyst to break the encryption system and get access to the original unencrypted data [22]. Perhaps even more important than that, cryptanalysis is also commonly used to evaluate the security of cryptosystems, by attempting to break them completely or in part in order to establish how could an attacker with a given set of knowledge and resources at their disposal endanger the security of the protected data.

There are many different ways of performing such an evaluation, depending on the goal of the analyst, on the assumptions about the analyst's information about the system, on their abilities to influence or even control the cryptosystem, and on the particular attack scenario. The analyst can choose to analyze the cryptosystem on its own or in interaction with other systems; they can analyze the theoretical concept, e.g. the algorithm(s) involved, of the cryptosystem as it performs separated from all other considerations, but they can also focus on some specific implementation of the cryptosystem, because practical aspects of such an implementation can introduce weaknesses that do not exist in the theoretical concept itself.

At the core of the "pure" cryptanalysis, dealing with the theoretical concepts of the cryptosystems irrespective of their implementations, we find three branches of cryptanalysis:

- *Algebraic cryptanalysis* tries to express the cryptosystem as a set of equations than can be solved using algebraic methods [3]. In the past, cryptosystems used to be linear and as a result, the algebraic methods were able to break them as easily as by solving a system of linear equations. In modern times, cryptosystems are created in such a way as to prevent this specific vulnerability: the modern cryptosystems contain non-linear elements and as a result their expressions consist largely on non-linear equations for which we currently lack an effective solving mechanism. A large part of algebraic cryptanalysis attempts to find such a solving mechanism, at least for the specific cipher being analyzed.

- *Linear cryptanalysis*, first published in [42], attempts to express the cryptosystem as a system of linear equations that could be solved easily using e.g. the Gauss elimination algorithm [34]. Of course, since modern cryptosystems are not linear, a fully linear expression is not possible; for that reason, linear cryptanalysis focuses on specific parts of the cryptosystem which can be expressed as linear equations, even if only in some cases. The fact that the created expressions are only probabilitic is offset by using multiple samples of plaintexts and their respective ciphertexts encrypted with a single key.

- *Differential cryptanalysis* also attempts to express the cryptosystem as a system of linear equation, but unlike linear cryptanalsysis it focuses more on the relations between plaintexts and ciphertexts: Instead of working with any available pair, it adds specific requirements on either the plaintexts or the ciphertexts, typically that for each plaintext-ciphertext pair the attacker is given another pair where the plaintext differs from the original in some known way — e.g. that some bits are inverted; the attacker does not know the original values of these bits, but she knows exactly how they were changed to make the second pair [10]. While the fact that the equations being formed are linear causes the same issues as with the linear cryptanalsys, this additional information about the relation between the plaintexts can eliminate certain aspects of the cryptosystem in ways which make for a more effective cryptanalysis technique.

In our thesis, we focused particularly on the linear cryptanalysis, because we found the concept itself fascinating and worth the study, but much easier to understand than the algebraic cryptanalysis. At the same time, while the technique is arguably less powerful than the differential cryptanalysis, it is closer to the real-world scenario of what an attacker would be working with and trying to achieve, e.g. solve the problem "here's a set of captured ciphertexts along with their known decryptions acquired through another means, try to decrypt the rest".

The second cornerstone of this thesis is the cipher AES [50] — it is a well-known and well-studied cipher which is being used in practical applications all over the IT world, so it's imperative that we are certain of its security. Of course, as a modern cipher, it has been designed with the three cryptanalytic techniques outlined above in mind [19], so it should be immune to them all — and to the linear cryptanalysis in particular. But it is

difficult to practically verify whether the design is actually resistant to linear cryptanalysis due to the sheer size of the cipher: Because of the size of its block as well as its key, it is not possible to explore all the possibilities or even a significant portion of them. Fortunately, the structure of AES makes it easy to create reduced models of it [17] and experiment on them; specifically, we can create smaller models designed along the same design principles, automatically even [61], and then exhaustively check these models for their behavior in all possible situations. Should we find a significant weakness, that may (or may not) impact the AES itself; on the other hand, if these reduced models proved resistant to linear cryptanalysis, we can expect the full AES to be at least as resistant as the model, and thus unbreakable by this means.

Since it is expected that AES was correctly designed and will resist linear cryptanalysis, we also decided to approach the problem of breaking the cipher from another direction — that we would attempt to find such weaknesses in specific implementations of AES that might help recover the key and/or the plaintext. Of course, since such weaknesses would be caused by the implementation rather than the cipher itself, they would not apply to the AES in general, only to that specific implementation; on the other hand, experience with other ciphers and other implementations gives us hope that any found vulnerabilities would likely be more serious than weaknesses in the cipher itself and might allow for practical attacks. Also, it might be possible to deduce some general recommendations for other implementations as to which mistakes should be avoided.

1.2 Goals of the Thesis

We set the following goals for our dissertation thesis:

- Analyze the design of AES: What were the requirements and design principles? From which components is the cipher composed, what was the motivation for choosing them? Can they be modified while maintaining the requirements and the design principles?

- Choose a suitable reduced model of AES that would conform to the design of AES both in respect to both principles and their realization, but would be sufficiently smaller in size to allow for an exhaustive verification of its security from linear cryptanalysis.

- Perform such a verification. Identify possible weak spots in the cipher design and if any are found, evaluate their impact on AES itself.

- Moving on to the implementation aspects: Assuming an attacker who is able to control an operating environment in which AES encryption and decryption is being performed, evaluate the options for an automatic detection of these operations and if possible, for automatic recovery of the encryption key, the plaintext, or both.

- Select a suitable real-world application that uses AES for encryption. Evaluate the cipher use in this application with respect to recovery of the key. If any vulnerabilities are found, identify mistakes which led to the creation of these vulnerabilities and propose guidelines on avoiding them.

1.3 Structure of the Thesis

The thesis is organized into six chapters:

1. *Introduction*: Describes the motivation behind our efforts along with our goals.

2. *The Rijndael Cipher*: Introduces the reader to the necessary theoretical background of the design of the Rijndael cipher, of which AES is a special case. It analyzes the components of the cipher along with the motivation for their selection by the cipher's designers, as shown primarily in [19].

3. *Linear Cryptanalysis of Baby Rijndael*: Describes Baby Rijndael [5], one of the possible models of AES. In the first part of the chapter we analyze the components of the cipher and verify that they match the components of AES in their design and selection principles. In the second part we perform an exhaustive linear cryptanalysis of the cipher, trying all combinations of plaintexts, keys and linear approximations to verify whether some specific combination exhibits worse behavior than the rest. We also attempt to combine the results of multiple approximations to improve the recovery of the overall key.

4. *Automatic Detection of AES Operations*: Discusses the different ways of implementing AES on Intel Architecture CPUs, both software-only and hardware-assisted bitslicing [37] or dedicated instructions [28], and for each such way proposes techniques that can be used to automatically detect that AES is being used. We show that the while the implementation methods differ significantly in this aspect, in many cases it is possible to not only automatically detect that AES is being used, but also to recover both the key and the plaintext of that use. Tests on real-world applications show that our technique is effective against different implementations of AES as long as they use the expected implementation concept. [A.2]

5. *Insecure Use of AES*: Demonstrates, on an example of a long-standing backup tool Drive Snapshot, a number of weaknesses of the AES implementation in that application, and discusses their possible causes. While these weaknesses are not vulnerabilities in the design of the cipher itself, we show that they significantly reduce the complexity of a possible attack — more than 57 % of the bits of key are revealed to the attacker and additional vulnerabilities in the key generation and key storage scheme allow for further exploitation. In extreme cases, the complexity of the attack falls under 26 bits or even to almost zero!

6. *Conclusions*: Summarizes the results of our research, suggests possible topics for further research, and concludes the thesis.

Chapter 2

The Rijndael Cipher

The Rijndael cipher is a symmetric block cipher designed in 1998 by Vincent Rijmen and Joan Daemen [18]. It was submitted to the public competition announced in 1997 by the National Institute of Standards and Technology with the purpose of finding a successor to the Data Encryption Standard, and after a three-round process emerged in 2001 as the chosen algorithm for Advanced Encryption Standard [50] not only because of its security, but also for its ability to be efficiently implemented on various hardware platforms [26][57]. As such, the cipher gained significant popularity worldwide and enormous support from developers. Today it is used in many applications in many different fields of information technology, both hardware and software. AES is supported by modern CPUs [28] as well as modern operating systems [46][27] and it is an integral part of many security-related standards and protocols, including such commonly used protocols as the SSL/TLS (where it is one of the very few remaining block encryption algorithms, as per the current TLS version 1.3) [54], SSH [4] or WPA.

2.1 The Design of Rijndael

Rijndael is a iterative symmetric block cipher in the form of a substitution-permutation network. Its input consists of a block of data (the so-called *plaintext*, PT) and the encryption *key*. The plaintext is converted into an internal *state* and masked by the key. The state then undergoes a number of *rounds*, where each round applies the same set of transformations to the state, including substitutions, permutations and an addition of the key. After a set number of rounds the internal state is considered sufficiently modified from the initial plaintext, or encrypted, and is converted back into a block of data — the *ciphertext* (CT). That concludes the encryption of the block and a new block can be encrypted, if necessary. The full process is described in detail in [19]; we will only focus on the key aspects of the description here.

2.1.1 Design Principles

In designing Rijndael, the creators applied a number of principles. Some of these are considered standard security practices, some of these were mandated by NIST in their requirements for the submitted ciphers or in their evaluation criteria, and some were chosen by the creators [19]. Generally, the selection was done in the following steps:

1. Define properties which are necessary for the security of the cipher, with reason given for these properties.
2. Propose ways for achieving these security objectives, e.g. list operations which would ensure that the requirements are satisfied.
3. From the options, select those that tend to be more secure than the others.
4. If multiple options remain, select those that are easier to implement on the target architectures (primarily 8-bit and 32-bit CPUs).
5. If multiple options remain, select the "simplest" one, for a given metric of simplicity.

The last rule attempts to ensure that the design of the cipher follows the "Nothing Up My Sleeve" principle, a phrase originally associated with magicians but widely adopted in cryptography [56], of avoiding choices which might raise questions about their security in the context of specific knowledge of the designer. For example, DES uses S-box contents which have never been entirely explained, although now we know that they were designed to resist differential cryptanalysis (not publicly known at the time of standardization of the cipher), GOST 28147-89 uses varying, and in some cases unpublished, S-box contents, the NIST-proposed Dual EC Random Number Generator was long suspected [59] and recently shown to contain a backdoor hidden in the constants used in its design [6], etc. Rijndael combats possible suspicions by essentially saying, "here's a list of choices which are optimal according to the security and efficiency criteria; we consider all of them equal but since we had to choose one, we chose X because it has some irrelevant but objective and easy-to-understand property; in case of doubts, another option can be selected".

2.2 Components of the Cipher

Rijndael consists of the following components, which are used in various stages of the encryption or decryption to transform the plaintext into ciphertext.

2.2.1 Cipher State

The cipher state of Rijndael is represented by a column-order major matrix of bytes. The matrix consists of four rows and a variable number of columns N_b, depending on the desired block size. The minimum specified block size is 128 bits with $N_b = 4$, and the maximum

is 256 bits with $N_b = 8$. Any value of N_b between these extremes is possible, allowing Rijndael to support block sizes of 128, 160, 192, 224 and 256 bits.

When transforming the input block, which is essentially a sequence of bytes, into the cipher state matrix, the values are copied one by one into columns from left to right, filling each column from the top to bottom. That is, for a 128-bit block, the 16 consecutive plaintext bytes $(p_0, p_1, ..., p_15)$ form a matrix A:

$$A = \begin{pmatrix} p_0 & p_4 & p_8 & p_{12} \\ p_1 & p_5 & p_9 & p_{13} \\ p_2 & p_6 & p_{10} & p_{14} \\ p_3 & p_7 & p_{11} & p_{15} \end{pmatrix}, p_i \in 0..255 \tag{2.1}$$

It should be noted that while the specification of the cipher adheres to the conventional interpretation of bytes as a sequence of 8 bits, i.e. it specifies the element p_i as an element of a Galois field of size $2^8 = 256$, the specification is flexible enough to allow, if the need for such a modification arose, for different-sized bytes, by the simple expedient of switching to a different Galois and adjusting the operations that depend on the size of an element. There is, indeed, a natural way for such an adjustment for all operations involved.

2.2.2 Key

Similarly to the state, the key of Rijndael is represented by a column-order major matrix of bytes. The key matrix also consists of 4 rows and a variable number of columns N_k, depending on the desired key size. The minimum specified key size is 128 bits with $N_k = 4$ and can be increased in 32-bit increments up to 256 bits with $N_k = 8$.

When transforming the original (or master) key into the key matrix, columns are filled from left to right, and within a column from top to bottom, just like it's the case for the state. Thus a 128-bit key represented by 16 bytes $(k_0, k_1, ..., k_15)$ will form a matrix K:

$$K = \begin{pmatrix} k_0 & k_4 & k_8 & k_{12} \\ k_1 & k_5 & k_9 & k_{13} \\ k_2 & k_6 & k_{10} & k_{14} \\ k_3 & k_7 & k_{11} & k_{15} \end{pmatrix}, k_i \in 0..255 \tag{2.2}$$

2.2.3 Rounds

Rijndael is an iterative cipher, meaning that its operation can be expressed as a series of repeated transformations which affect its state, with the initial state being the plaintext. Each repetition of the transformations is called a *round*; the output of a round becomes the input of the next round, until a predetermined number of rounds has been completed and the final cipher state becomes the ciphertext. Generally, it is assumed that by increasing the number of rounds we gain increased security at the cost of decreased performance; anyone designing a new iterative cipher is faced with the problem of finding the best trade-off between these two aspects.

In Rijndael, the number of rounds N_r depends on both the size of the key and the size of the block. As per chapters 3.5 and 5.5 of [19], the choice of N_r was predicated by the following criteria:

1. The creators proposed the concept of *K-security*:

   > A block cipher is K-secure if all possible attack strategies for it have the same expected work factor and storage requirements as for the majority of possible block ciphers with the same dimensions. This must be the case for all possible modes of access for the adversary (...) and for any a priori key distribution. [19]

   Assuming that past certain dimensions of a cipher (given by the size of the block and the size of the key) only a negligible minority of ciphers of that size will contain exploitable weaknesses, that means that a K-secure cipher can only be broken with approximately the same effort as ciphers without exploitable weaknesses, regardless of their knowledge about the plaintext, the ciphertext, or the relations between chosen keys, or even their ability to specify it.

2. At the time of the submission of Rijndael to the AES competition, the best known attacks that would violate the K-security property of the cipher were only able to operate on Rijndael variants with less than 6 rounds, making that a reasonable starting point.

3. In Rijndael with a 128 bit block, it takes two rounds for a change of a single bit in the state to spread uniformly to all bits of the state — in the first round, the change propagates to the whole column of the state, and in the second round these changes propagate to all other columns.

4. Known cryptanalytic attacks generally aim to propagate known features of a state after n rounds to attack the $(n+1)$-th or $(n+2)$ rounds. In order to defend against them, the initial number of rounds was increased by 2 rounds, plus another 2 to account for possible as-yet-unknown attacks that might appear in the future.

5. For keys longer than 128 bits, the number of rounds was increased by 1 for every extra 32 bits of the key, to counter the fact that 1) with a longer key the full keyspace increases exponentially but a violation of K-security does not require an exponential improvement over that, and 2) more bits in a key give the attacker more possibilities in related-key scenarios.

6. For blocks larger than 128 bits, an extra round was added for every extra 32 bits of the block, to counter 1) the increased number of rounds needed for full diffusion of one changed bit (three compared to two, because after only two rounds $(N_b - 4)$ columns remain unaffected by the change), and 2) the fact that more bits in the state may provide the attacker with more flexibility in designing their attack, despite that the creators themselves failed to find a way to exploit it.

7. For variants where both the block size and the key size were increased over 128 bits, only the largest increment in the number of rounds was used, in order to minimize the impact on performance of the cipher.

As a result of these decisions, N_r was set to 10 rounds for the Rijndael variant with 128 bits of both the key and the block sizes, growing up to 14 rounds for the variants with 256 bit key and/or block sizes. That value was seen as providing a sufficient security margin for foreseeable types of attacks while causing the least decrease in performance of the cipher.

2.2.4 Round Structure

Each of the N_r rounds, except for the first and the last, consists of four transformations of the cipher state run in sequence. These transformations are:

1. SubBytes
2. ShiftRows
3. MixColumns
4. AddRoundKey

Each of the transformations provides a certain function or property of the cipher, as explained in the successive sections.

The first and the last round introduce the following differences:

- The state that enters the first row is *not* simply a copy of the plaintext. Instead, an initial masking of the plaintext with the key using the AddRoundKey transformation is performed before the plaintext enters the first round. This modification is necessary to prevent known-plaintext attacks from circumventing the majority of the first round, effectively decreasing the number of rounds by one.
- The last round does not perform the MixColumns step in order to improve the cipher's performance, since MixColumns is easily the most computationally-intensive component of the cipher and it only performs a linear transformation which, on its own, would be easily circumvented by an attacker.

For decryption, the steps are reversed.

In essence, the cipher as a whole can be expressed as algorithms 2.1 and 2.2.

Alternatively, using the algebraic properties of the operations, especially the independence of (Inv)SubBytes and (Inv)ShiftRows and the fact that the places of InvMixColumns and AddRoundKey can be exchanged provided that InvMixColumns had also been applied to the key, we can reorder the decryption steps to match the steps of encryption, as shown in Alg. 2.3. This modification can be useful e.g. for hardware implementations of the cipher.

Algorithm 2.1 Rijndael Encryption

```
1: function RijndaelEncrypt(PlainText, ExpandedKey, Rounds)
2:     State ← AddRoundKey(PlainText, ExpandedKey[0])
3:     for Round ← 1 to Rounds − 1 do
4:         State ← SubBytes(State)
5:         State ← ShiftRows(State)
6:         State ← MixColumns(State)
7:         State ← AddRoundKey(State, ExpandedKey[Round])
8:     end for
9:     State ← SubBytes(State)
10:    State ← ShiftRows(State)
11:    State ← AddRoundKey(State, ExpandedKey[Rounds])
12:    return State
13: end function
```

Algorithm 2.2 Rijndael Decryption

```
1: function RijndaelDecrypt(CipherText, ExpandedKey, Rounds)
2:     State ← AddRoundKey(State, ExpandedKey[Rounds])
3:     State ← InvShiftRows(State)
4:     State ← InvSubBytes(State)
5:     for Round ← Rounds − 1 downto 1 do
6:         State ← AddRoundKey(State, ExpandedKey[Round]
7:         State ← InvMixColumns(State)
8:         State ← InvShiftRows(State)
9:         State ← InvSubBytes(State)
10:    end for
11:    State ← AddRoundKey(CipherText, ExpandedKey[0]
12:    return State
13: end function
```

It needs to be noted that unlike the Feistel scheme which ensures the invertibility of the cipher regardless of the specific round transformations used [58], a substitution and permutation network scheme requires that every round transformation is itself invertible — if this weren't the case, then the cipher itself could not be inverted and thus decryption would not be possible.

2.2.4.1 SubBytes

The SubBytes transformation is the cornerstone of the *confusion* property of Rijndael. Its purpose is to replace each element of the cipher state with another element in such a way, that:

1. The replacement is non-linear. Specifically, even in the worst case scenario the cor-

Algorithm 2.3 Equivalent Rijndael Decryption

```
1: function RijndaelEqDecrypt(CipherText, ModifiedExpandedKey, Rounds)
2:     State ← AddRoundKey(State, ModifiedExpandedKey[Rounds])
3:     for Round ← Rounds − 1 to 1 do
4:         State ← InvSubBytes(State)
5:         State ← InvShiftRows(State)
6:         State ← InvMixColumns(State)
7:         State ← AddRoundKey(State, ModifiedExpandedKey[Round])
8:     end for
9:     State ← InvSubBytes(State)
10:    State ← InvShiftRows(State)
11:    State ← AddRoundKey(State, ModifiedExpandedKey[0])
12:    return State
13: end function
```

relation between the input and the output value should be as low as possible and the probability that the propagation of differences between input values to output values should be as low as possible.

2. The algebraic expression of the transformation is complex.

Both of these requirements are necessary to make algebraic attacks on the cipher difficult.

For implementation and performance purposes, the designers chose to use the same replacement function for all rounds of the cipher, contrary to some competing designs including DES, which use multiple functions. One function allows for a very efficient implementation in both hardware and software.

In case of Rijndael, the SubBytes transformation was defined as a function which would transform one input byte of the state into one output byte of the state, as shown in Fig. 2.1, implemented using a substitution table.

The substitution table is formed in the following fashion:

A given byte $a = a_0 + 2a_1 + 2^2a_2 + ... + 2^7a_7, a_i \in \{0, 1\}$, represents a polynomial $a(x) = a_0 + a_1x + a_2x^2 + ... + a_7x^7$ in a Galois field GF(2^8). Its replacement value $b = b_0 + 2b_1 + 2^2b_2 + ... + 2^7b_7, b_i \in \{0, 1\}$ is found by calculating its inverse value modulo irreducible polynomial $m(x)$ and applying an affine transformation modulo $n(x)$:

$$b(x) = \left(a^{-1}(x) \pmod{m(x)}\right) \cdot c(x) + d(x) \pmod{n(x)}, \tag{2.3}$$

where $m(x) = 1 + x + x^3 + x^4 + x^8$, $n(x) = 1 + x^8$, $c(x) = 1 + x^4 + x^5 + x^6 + x^7$ and $d(x) = x + x^2 + x^6 + x^7$. In adherence to the "nothing up my sleeve" principle above, the choice of the operations and constants follows very simple reasoning:

- Modular inverse is a natural non-linear operation, and when performed in a Galois field modulo an irreducible polynomial, it is also invertible, except for a value of `0x00`

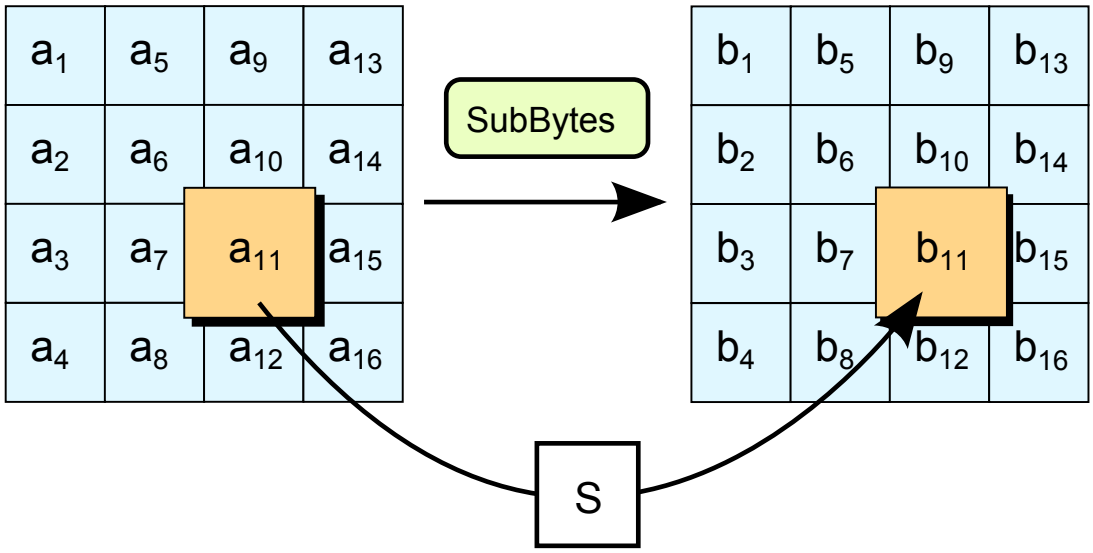

Figure 2.1: The SubBytes transformation of Rijndael. Each byte of the state is replaced with another byte by looking it up in the substitution table S. This replacement is constant throughout the cipher, regardless of the round or any other variable.

for which the inverse does not exist; for the purpose of the cipher it is defined that `0x00` is its own inverse.

- The affine transformation is itself linear and not affect the non-linearity of the inverse in any way. The purpose of the multiplication is to modify the simple algebraic expression of the inverse alone into a much more complicated form with high degrees of polynomials, the purpose of the addition is to ensure that no value (such as `0x00`) can pass through the overall transformation unmodified, and preferably that a half of the bits gets modified even in the worst-case scenario.
- Since many affine transformations exist that would have that property, the designers of Rijndael chose to apply an additional restriction that the transformation should have no fixed points and no opposite fixed points, that is:

$$\text{SubBytes}(x) \oplus (x) \neq \texttt{0x00}, \forall x, \text{SubBytes}(x) \oplus (x) \neq \texttt{0xFF}, \forall x, \tag{2.4}$$

  although there isn't any known attack that would exploit these points if they did exist. In this sense this requirement can be considered optional.
- The actual polynomial selected are not important as long as they maintain the properties listed above. For example, the choice of $m(x)$ was made because it's the smallest polynomial capable of generating GF(2^8). Similarly, $c(x)$ and $d(x)$ were chosen for their "nice appearance" while achieving the properties listed above.

In order to achieve maximum performance, the SubBytes transformation is implemented as a replacement table rather than through the defining operations — that is, instead of calculating modular inverses, multiplications and additions, the results of these operations are saved in a table and a lookup is performed any time SubBytes needs to be performed; an array of 256 values where the input represents the index to the array and the value at that index represents an output is an example of an efficient implementation. The inverse operation can be performed using a table with the same structure but of course a different content.

2.2.4.2 ShiftRows

The ShiftRows operation is one of the two steps which aim to achieve diffusion in the cipher by shifting the rows of the state matrix cyclically to the left (when encrypting) and right (when decrypting), as shown in Fig. 2.2. It wouldn't be effective on its own, it needs to work in tandem with MixColumns.

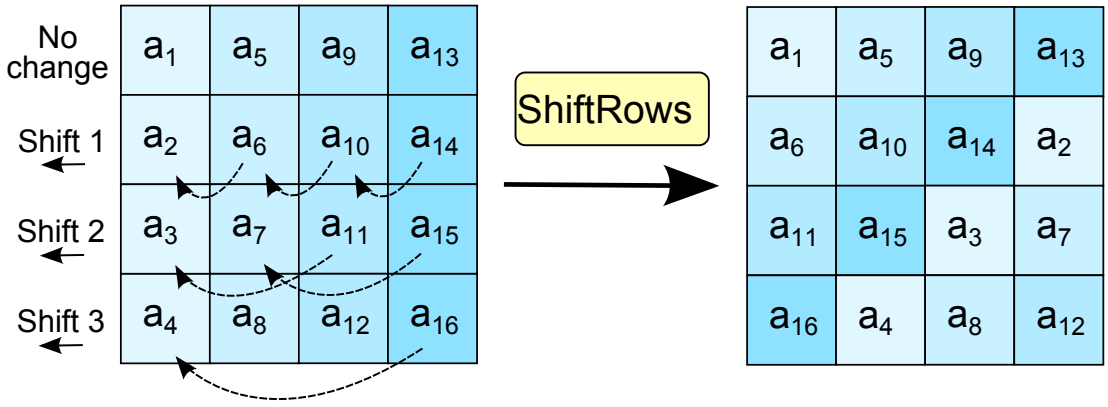

Figure 2.2: The ShiftRows transformation of Rijndael. Each row of the state is cyclically shifted left (resp. right for decryption) by a set number of columns, different for each row.

The choice of ShiftRows was mostly mandated by the simplicity of the operation and the ease of implementation in both hardware and software: It would be just as possible to use perform any other permutation withing a row of the state matrix with similar results, but shift by a whole byte are by far the easiest to implement, regardless of the size of the matrix.

With that decision fixed, the question remained as to the size of the shifts. The chief criterion was that each row should shift by a different number of places, because that allows for the fastest diffusion of a changed columns into all other columns. That effectively means that for 128 bit blocks, the shifts must be by 0, 1, 2 and 3 bytes respectively, because there are four rows and four different shift values; the decision of which shift will happen in

which row was made once again with the "nothing up my sleeve" approach — the simplest choice is to shift the n-th row by n places.

With larger block sizes, there are more options. For example, a 160 bit block still has four rows but five possible shift values to choose from, so variations are possible. It was shown [19] that some combinations of shifts perform better than others against truncated differential attacks and saturation attacks. The less effective variants were discarded and from the remaining options, the simplest were chosen.

2.2.4.3 MixColumns

The MixColumns is the second diffusion component of the cipher, complementary to the ShiftRows transformation. Its purpose is to mix the values within each column of the state, thus spreading the value of every bit among all other bits in the column. The main criteria for the selection of the operation were its linearity and the relevant diffusion power, both of which are a direct result of the wide trail strategy in cipher design used to combat the linear and differential cryptanalysis, as described in [19].

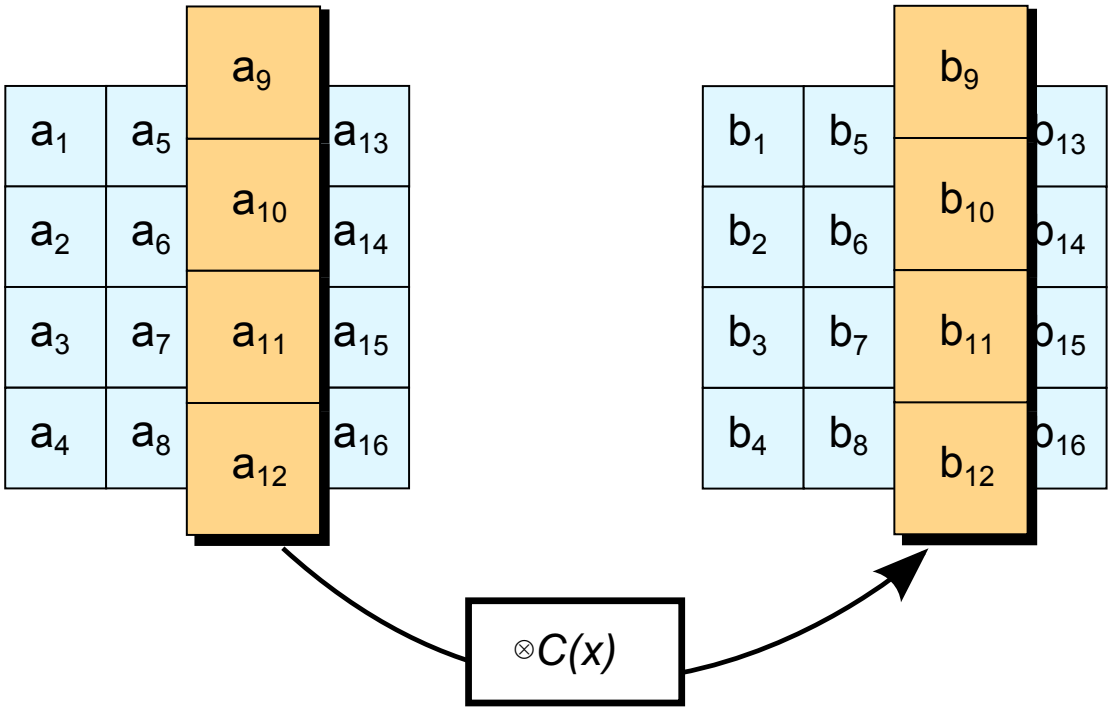

Figure 2.3: The MixColumns transformation of Rijndael. Each column is processed individually by performing a matrix multiplication of the column and a fixed matrix. The decryption only differs in its use of a different (inverse) matrix.

A supplementary criterion of a good performance of the cipher even on 8-bit CPUs led to the selection of a polynomial multiplication: Interpret the four input values (b_0, b_1, b_2, b_3) in the column as the coefficients of a polynomial $b(x) = b_0 + b_1x + b_2x^2 + b_3x^3$ over GF(2^8), and the four output (d_0, d_1, d_2, d_3) in the column as the coefficients of a polynomial $d(x) = d_0 + d_1x + d_2x^2 + d_3x^3$ over GF(2^8). Then $d(x)$ can be calculated as $d(x) = b(x) \cdot c(x) \pmod{x^4 + 1}$, where $c(x)$ is a fixed polynomial over GF(2^8) with coefficients compatible with the requirements on the diffusion power and the general requirements on the cipher components, e.g. invertibility.

The performance requirement strongly favors coefficients with a low value and a few non-zero bits, such as `0x00`, `0x01`, `0x02` or `0x03`: the multiplication by either `0x00` or `0x00` requires no calculation at all, multiplication by `0x02` can by implementing by shifting the value by 1 bit and the multiplication by `0x03` can be implementing using one shift and one addition. For this reason, these values were favored for the coefficient selection, and among all possible combinations, one was selected that 1) exhibited the relevant diffusion power and 2) had inverse coefficients that could also be easily calculated. That led to the selection of $c(x) = \texttt{0x03} \cdot x^3 + \texttt{0x01} \cdot x^2 + \texttt{0x01} \cdot x + \texttt{0x02}$ with an inverse polynomial $c'(x) = \texttt{0x0B} \cdot x^3 + \texttt{0x0D} \cdot x^2 + \texttt{0x09} \cdot x + \texttt{0x0E}$. Alternatively, the calculations can be expressed in matrix form as:

$$\begin{pmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{pmatrix} = \begin{pmatrix} \texttt{0x02} & \texttt{0x03} & \texttt{0x01} & \texttt{0x01} \\ \texttt{0x01} & \texttt{0x02} & \texttt{0x03} & \texttt{0x01} \\ \texttt{0x01} & \texttt{0x01} & \texttt{0x02} & \texttt{0x03} \\ \texttt{0x03} & \texttt{0x01} & \texttt{0x01} & \texttt{0x02} \end{pmatrix} \times \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} \pmod{x^4 + 1} \tag{2.5}$$

for encryption and:

$$\begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} \texttt{0x0E} & \texttt{0x0B} & \texttt{0x0D} & \texttt{0x09} \\ \texttt{0x09} & \texttt{0x0E} & \texttt{0x0B} & \texttt{0x0D} \\ \texttt{0x0D} & \texttt{0x09} & \texttt{0x0E} & \texttt{0x0B} \\ \texttt{0x0B} & \texttt{0x0D} & \texttt{0x09} & \texttt{0x0E} \end{pmatrix} \times \begin{pmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{pmatrix} \pmod{x^4 + 1} \tag{2.6}$$

for decryption.

2.2.4.4 AddRoundKey

The AddRoundKey step introduces the key to the state. In each round, and before the first round, one round key is XORen onto the state: Each round key takes a form of a matrix of the same dimensions as the matrix of the state, and state element in row i and column j gets XORed with the round key element in the same row i and column j, as shown in Fig. 2.4. Since XOR is its own inverse, the decryption is performed in exactly the same way.

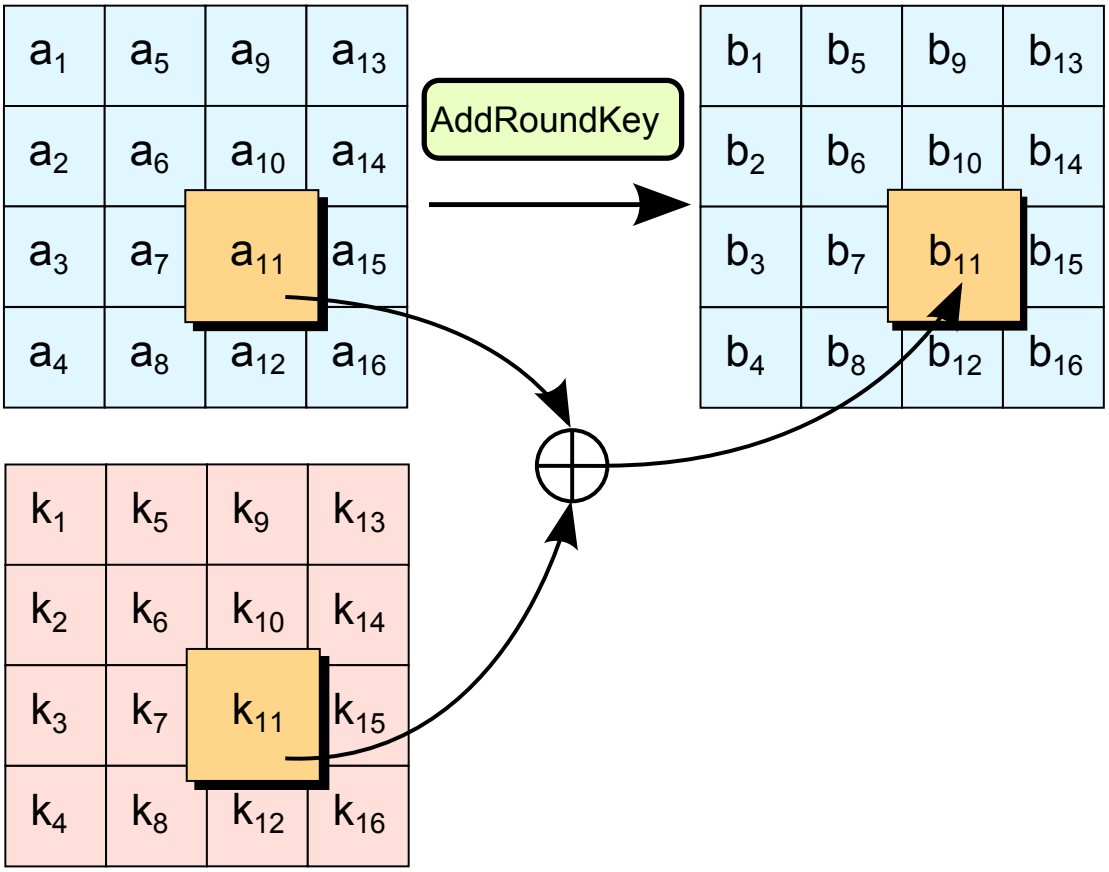

Figure 2.4: The AddRoundKey transformation of Rijndael. For each round, a specific round key is derived from the user's key, in such a way that the dimensions of the round key matrix are the same as those of the state. Then each element of the state is XORed with the respective element of the round key. Decryption is exactly the same.

2.2.5 Key Schedule

The previous step introduced an issue that needs to be addressed, and that is the discrepancy between the size of the cipher's key and the need to use multiple round keys. First, there's just one cipher key but 11 to 15 round keys. Second, the round keys dimensions must match the dimensions of the state but there is no such requirement for the cipher key; there could be a match, as in Rijndael with 128 bit blocks and 128 bit keys, but not necessarily — Rijndael with 128 bit blocks and 256 bit keys obviously has a larger cipher key that the state, and even the opposite is possible for Rijndael with 256 bit blocks and 128 bit keys. Clearly, the cipher needs a mechanism which will take the user-provided key of a fixed size of $32N_k$ bits and generate $N_r + 1$ round keys of possibly different size of $32N_b$ bits. This mechanism is called "key schedule" or "key expansion".

round	1	2	3	4	5	6	7
Rcon(round)	0x01	0x02	0x04	0x08	0x10	0x20	0x40
round	8	9	10	11	12	13	14
Rcon(round)	0x80	0x1B	0x36	0x6C	0xD8	0xAB	0x4D

Table 2.1: The values for the round-dependent constant `Rcon` for common round values of Rijndael.

The key expansion mechanism of Rijndael is predicated upon 4 basic requirements:

1. It should be possible to efficiently implement it in both hardware and software, and on different architectures, with respect to both the processing power (the number of steps to be taken for each new key) and the memory consumption (the amount of working memory required for executing the key expansion algorithm).

2. The algorithm should use constants with different value for each round, thus eliminating symmetry.

3. It should perform an efficient diffusion of the bits of the cipher's key to the round keys.

4. The mechanism should have such a non-linear component as to significantly complicate progression of differences between keys to the round keys.

In order to satisfy these requirements, and in particular the efficient implementation on 8-bit architectures, the following scheme was introduced: The expanded key takes a form of a matrix W with 4 rows and $N_b(N_r+1)$ columns. The first N_k columns are directly copied from the cipher key. The remaining $N_b(N_r+1) - N_k$ columns are iteratively calculated from the preceding columns as shown in Alg. 2.4:

The function uses SubBytes for its non-linear element and the round-dependent constant Rcon, which can be expressed as:

$$\text{Rcon}(round) = x^{round} \pmod{m(x)} \tag{2.7}$$

but is usually used pre-calculated for all common round values (see Tab. 2.1).

The graphical representation of the main part of the algorithm (after copying the cipher's key) for Rijndael variants with $N_k \leq 6$ is also shown in Fig. 2.5.

The key expansion function is itself invertible, although that is not by itself a requirement for a block cipher to be invertible. In Rijndael, this property was introduced in order to minimize the memory footprint of the cipher as it is not necessary to pre-calculate the whole expanded key during the initialization of the cipher: It is always possible to iteratively calculate further columns of an expanded key from the last N_k columns, both during encryption and during decryption. This property will prove useful in one of our attacks — see chapter 4.

Algorithm 2.4 Rijndael Key Expansion

```
1: function RijndaelEncrypt(byte K[4][N_k], byte W[4][N_b(N_r + 1)])
2:     In:  K = the cipher key
3:     Out: W = the expanded key
4:     for col ← 0 to N_k − 1 do
5:         for row ← 0 to 3 do
6:             W[row][col] ← K[row][col]
7:         end for
8:     end for
9:     for col ← N_k to N_b(N_r + 1) do
10:        if col mod N_k = 0 then
11:            W[0][col] = W[0][col − N_k] ⊕ SubBytes([W[1][col − 1]) ⊕ Rcon(col/N_k)
12:            for row ← 1 to 3 do
13:                W[row][col] ← W[row][col − N_k] ⊕ SubBytes(W[row + 1 mod 4][col − 1])
14:            end for
15:        else if N_k > 6 and col mod N_k = 4 then
16:            for row ← 0 to 3 do
17:                W[row][col] ← W[row][col − N_k] ⊕ SubBytes(W[row][col − 1])
18:            end for
19:        else
20:            for row ← 0 to 3 do
21:                W[row][col] ← W[row][col − N_k] ⊕ W[row][col − 1]
22:            end for
23:        end if
24:    end for
25: end function
```

2.3 Rijndael vs AES

Rijndael was selected by NIST to become the new Advanced Encryption Standard (AES). Unlike Rijndael, though, AES is far more restrictive in its choice of parameters; specifically, the block size is always 128 bits (Rijndael allows 128 to 256 bits with 32 bit increments), the key size is either 128, 192 or 256 bits (Rijndael also allows 160 and 224 bits) and any constants, polynoms and other values where a variation is possible in Rijndael were fixed at the values chosen by the cipher designers. In this sense, AES is an application of Rijndael where a certain flexibility has been sacrificed in favor of more security (as only a subset of all possible Rijndael variants needs to ne evaluated) and a simplified implementation. So far, the benefits seem to be borne out in practice.

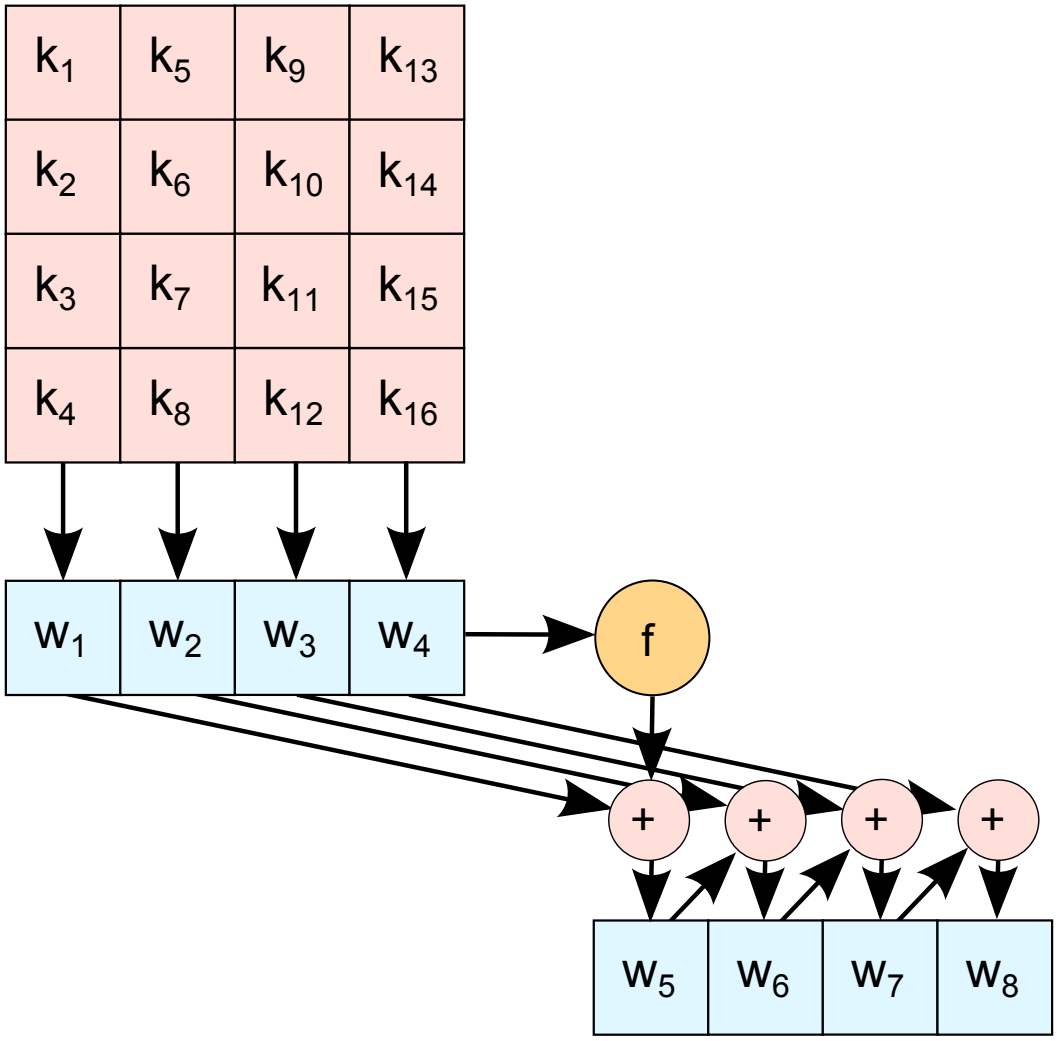

Figure 2.5: The main part of Key Expansion (after copying the cipher key) for key lengths up to 192 bits. f is the round-dependent non-linear function that XORs the first byte of the column with a round-dependent constant and then performs SubBytes on all bytes of the column.

Chapter 3

Linear Cryptanalysis of Baby Rijndael

Since their introduction to the public cryptanalytic research, linear and differential cryptanalysis have been accepted by the cryptological community and are now the standard techniques to be used to analyze the security of existing and proposed ciphers alike. However, it seems that differential cryptanalysis [9] is the more successful one — not only in itself, but also in its various modifications such as impossible differential cryptanalysis [7][8], boomerang attacks [65], related key attacks [12][11] or biclique attacks [13], which often represent the state-of-the-art in cryptanalysis of current ciphers.

Linear cryptanalysis, in comparison, can't boast such successes. When Matsui originally published it [42], it gained a significant critical acclaim and was successful in breaking a number of then-current cryptosystems (particularly DES [42], but also a reduced-round PRESENT [16] or GOST [60]), and some interesting extensions have been found such as zero-correlation linear cryptanalysis [14], but it seems that the success rate is somewhat smaller than that of its older companion technique, despite the original expectations to the contrary [15].

We feel that the current research into linear cryptanalysis can be improved in several aspects. Particularly, we focus on actual experimental evaluation of Matsui's Algorithm 2 [42] as applied to Baby Rijndael [5], a reduced Rijndael-family cipher, where the technique exhibits several interesting properties: our experimental results show that the success rate of various linear approximations is highly dependent on their structure, not only on their probability bias, that certain key bits are more easily recoverable than other key bits, and that these key bits are easier to recover for some keys than for other keys.

In this chapter, revised and expanded from [A.1], we will briefly describe Baby Rijndael, Matsui's Algorithm 2 and our modifications to get a better performance out of them. We show that for Baby Rindael, there exist many different optimal linear approximations, and present the experimental results which demonstrate the different success rate of these approximations under Matsui's Algorithm 2. We then modify the algorithm to recover only selected key bits and show experimental data which demonstrate how the success rate of recovery of different bits differs. Finally, we adapt multiple approximation analysis [36] to our data and present experimental results demonstrating the varying difficulty of

recovering key bits for different keys.

3.1 Our Approach

In our study of the properties of linear cryptanalysis, we wanted to evaluate our subject cipher by experimental verification, to verify whether there are any unexpected parameters to the success rate of the technique used. Unfortunately, this is particularly difficult for modern ciphers which use key and block sizes much to large to allow for exhaustive tests of all possibilities. For this reason we decided to use a cipher which is not realistic in the sense that it allows for exhaustive processing, but at the same time relates to real-world ciphers in a significant way, so that the results obtained could potentially be extended to these ciphers. From among the available options, we chose Baby Rijndael by Cliff Bergman.

3.2 Baby Rijndael

Baby Rijndael is a block cipher proposed by Cliff Bergman [5] as an educational block cipher. It is modelled after Rijndael (AES), but with reduced key- and block-space: it uses 16-bit blocks and 16-bit keys. Its design, however, follows the design of the full Rijndael, respecting the requirements, implementations and design decisions set by Daemen and Rijmen in Rijndael proposal [19].

The state of the cipher is represented by a column-major 2×2 matrix A, where each element a_{ij} is a four-bit number. The state is initially filled by the plaintext and xor-ed by the key; then it undergoes three rounds of transformations consisting of a sequence of SubBytes, ShiftRows, MixColumns and AddRoundKey; the last, fourth round, omits the MixColumns transformation. The individual transformations, as well as the key schedule, are defined much like the same-named transformations of Rijndael, except for the size of the state and individual elements.

3.2.1 SubBytes

The SubBytes operation of Rijndael transforms each element of the state separately by replacing the original value by a new value found in a lookup table. The lookup table is formed in the following fashion:

A given byte $a = a_0 + 2a_1 + 2^2a_2 + ... + 2^7a_7, a_i \in \{0, 1\}$, represents a polynomial $a(x) = a_0 + a_1x + a_2x^2 + ... + a_7x^7$ in a Galois field $GF(2^8)$. We find a multiplicative inverse of $a(x)$ modulo irreducible polynomial $m(x) = 1 + x + x^3 + x^4 + x^8$ and apply an affine transformation in the form of a modular multiplication and addition in $GF(2^8)$:

$$b(x) = \left| |a^{-1}(x)|_{m(x)} \cdot c(x) \right|_{n(x)} + d(x), \tag{3.1}$$

where $c(x) = 1 + x^4 + x^5 + x^6 + x^7$, $d(x) = x + x^2 + x^6 + x^7$ and $n(x) = 1 + x^8$. Finally, we transform the polynomial $b(x)$ back to a byte $b = b_0 + 2b_1 + 2^2b_2 + ... + 2^7b_7, b_i \in \{0, 1\}$.

Baby Rijndael's SubBytes operation is defined in a very similar way. The nibble $a' = a'_0 + 2a'_1 + 2^2a'_2 + 2^3a'_3$ represents a polynomial $a'(x) = a'_0 + a'_1x + a'_2x^2 + a'_3x^3$ in a Galois field $GF(2^4)$, the output value is calculated as

$$b'(x) = \left||a'^{-1}(x)|_{m'(x)} \cdot c'(x)\right|_{n'(x)} + d'(x), \tag{3.2}$$

where $c'(x) = 1 + x^2 + x^3$, $d'(x) = x + x^3$, $m'(x) = 1 + x + x^4$ and $n'(x) = 1 + x^4$. Then we transform $b'(x)$ to nibble $b' = b'_0 + 2b'_1 + 2^2b'_2 + 2^3b'_3$.

The differences between Baby Rijndael and Rijndael are:

- A different Galois field, which is a necessary result of the reduced space of Baby Rijndael and is expected.
- A different irreducible polynomial, which is a necessary result of the changed Galois field. Note that even though Bergman in [5] changed this aspect of the cipher, the actual choice of the polynomial follows the Rijndael's design motivations: According to the design specifications [19], Rijndael can use any irreducible polynomial of the eighth order. The authors chose the one above because it is the smallest one. In the same way, the irreducible polynomial of Baby Rijndael is the smallest irreducible polynomial of the fourth order.
- A different affine transformation is also a result of a different size of the cipher. Baby Rijndael does use the same operations as Rijndael, but with different values. According to [19], the transformation should be invertible and should lead to a complex algebraic description of the SubBytes step; particularly, it should use a large number of high-order powers of x, and the additive polynomial should invert exactly half of the bits. We can see these requirements are fulfilled for Baby Rijndael as well.

Furthermore, we verified by calculating all 16 values that the substitution table from [5] was indeed calculated according to these specifications.

3.2.2 ShiftRows

The ShiftRows transformation is intended to mix values between columns. To this end, it is designed as a series of cyclic shifts, where each row is shifted by a certain number of columns to the left. The cipher specification doesn't require that a given row shifts by a certain number of columns; it just places the following constraints on the individual shifts: each row should shift by a different number of columns, one row should remain unshifted (or shift by zero columns).

In case of Rijndael with 128-bit block, these requirements alone force the shifts to 0, 1, 2 and 3 columns, though the assignment to rows is arbitrary; Rijndael assigns a smaller shift to the earlier row, i.e., the first row doesn't shift at all, the second row shifts by one column etc.

In the same fashion, Baby Rijndael's 2x2 state matrix forces the shifts to 0 and 1 columns, with the first row not shifted. Therefore, the operation is the same as with Rijndael, except for the size of the state.

It should be noted that for blocks larger than 128 bits, the shift choices are not fixed — we have four rows and 5-6 possible shifts. In such a case, the Rijndael specification provides additional requirements to the ShiftRows operation. It's not relevant to the Baby Rijndael, though we would have to consider these requirements if we used a state matrix of a different shape (4 rows, 4 columns, for example).

3.2.3 MixColumns

The MixColumns transformation is intended to mix values within a column, for all columns independently. For this purpose, Rijndael understands each column of the cipher's state as a polynomial $b(x) = b_0 + b_1x + b_2x^2 + b_3x^3$, where b_i is the column's element in row i, and calculates the output column $d(x)$ as $d(x) = |b(x) \cdot c(x)|_{1+x^4}$, where $c(x) = \texttt{0x02} + \texttt{0x01}x + \texttt{0x01}x^2 + \texttt{0x03}x^3$ and all coefficients are treated as polynomials in $GF(2^8)$ modulo $m(x)$.

Baby Rijndael's definition superficially appears to be completely different, because it calculates each bit separately by multiplying the input by a binary matrix t [5]. It can be shown, though, that this matrix multiplication can be expressed equivalently to a polynomial multiplication in the form of $d(x) = |b(x) \cdot c(x)|_{1+x^2}$, where $b(x) = b_0 + b_1x$ is the input polynomial with coefficients b_i coming from the i-th row of the state matrix, $c(x) = \texttt{0x05} + \texttt{0x0d}x$ and all coefficients are treated as polynomials in $GF(2^4)$ modulo $m(x)$.

The differences between Rijndael's MixColumns and Baby Rijndael's are:

- The order of $b(x)$ is necessarily smaller in Baby Rijndael due to the smaller number of elements in the state's column. The same is true for the polynomial $1 + x^4$.

- The coefficients of $c(x)$ for Rijndael are chosen so that $c(x)$ has an inverse modulo $1 + x^4$, the multiplication is fast on 8-bit computer, and the whole transformation has a high relevant diffusion power.

  Baby Rijndael satisfies the first requirement with $|c(x)^{-1}|_{1+x^2} = \texttt{0x04} + \texttt{0x0b}x$ and ignores the second as irrelevant.

  The relevant diffusion power, calculated as the minimum of branch numbers (see [19]), comes to 5 in case of Rijndael, the best possible value for a 4-element column. We calculated branch numbers for all 256 possible column values in Baby Rijndael and found that the minimum is 3, the best possible value for a 2-element column. We conclude that Baby Rijndael satisfies even the third requirement.

Despite the different representation of the cipher's description, we have shown that the MixColumns transformation of Baby Rijndael follows the specifications and the design criteria of the Rijndael itself.

3.2.4 AddRoundKey

The AddRoundKey transformation is defined as a simple `xor` between bits of the state and bits of the round key. This is true for both Rijndael and Baby Rijndael; the only difference is the length of the state and key.

3.2.5 The Number of Rounds

The number of rounds for Rijndael was chosen somewhat arbitrarily (see [19]) by testing which number of rounds rendered shortcut attacks inefficient[1] and adding four extra rounds for security against future attacks.

Baby Rijndael specifies the number of rounds as 4, without any explicit reason given. We believe this is due to the high diffusion power of the cipher, which gains full diffusion in two rounds, i.e., every state bit depends on all state bits two rounds earlier, so that at least two full diffusions are performed during the encryption.

While this suggests a possible security flaw in the design of Baby Rijndael, it should be noted that unlike Rijndael itself, Baby Rijndael is not constrained by performance concerns and can increase the number of rounds arbitrarily. If we encounter a technique which breaks Baby Rijndael, we can add two or more extra rounds and check whether the attack still works.

3.2.6 The Key Schedule

The key schedule expands the 128-bit master key into 11 128-bit round keys needed by Rijndael. The first 128 bits are simply copied from the master key, the others are formed iteratively:

$$\begin{aligned}(w_1, w_2, w_3, w_4) = \\ (k_1 \parallel k_2 \parallel k_3 \parallel k_4, k_5 \parallel k_6 \parallel k_7 \parallel k_8, k_9 \parallel k_{10} \parallel k_{11} \parallel k_{12}, k_{13} \parallel k_{14} \parallel k_{15} \parallel k_{16}),\end{aligned} \tag{3.3}$$

$$\begin{aligned}(w_{4i+1}, w_{4i+2}, w_{4i+3}, w_{4i+4}) = \\ (w_{4(i-1)+1} \oplus f(w_{4(i-1)+4}), w_{4(i-1)+2} \oplus w_{4i+1}, w_{4(i-1)+3} \oplus w_{4i+2}, w_{4(i-1)+4} \oplus w_{4i+3}),\end{aligned} \tag{3.4}$$

where k_i is the i-th byte of the master key, w_i is the i-th 32-bit word of the expanded key, $\parallel$ is the operation of concatenation and $f(x)$ is a non-linear function which first rotates the input word x left by one byte, performs `SubBytes` on all its bytes, and then `xor`s the result with a constant Rcon$[r]$, where r is a round number and Rcon$[r] = |(x^{r-1}|_{m(x)}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00})$.

[1] compared to exhaustive key search

Baby Rijndael uses the same schema, except that the words w_i are only 8-bit and the sizes of f and Rcon are reduced accordingly:

$$(w_1, w_2) = (k_1 \parallel k_2), \tag{3.5}$$

$$(w_{2i+1}, w_{2i+2}) = (w_{2(i-1)+1} \oplus f(w_{2(i-1)+2}), w_{2(i-1)+2} \oplus w_{2i+1}), \tag{3.6}$$

The inner workings of f and the values of Rcon are defined the same way as with Rijndael.

3.3 Linear Cryptanalysis

Our approach to linear cryptanalysis follows Matsui's specifications for Algorithm 2 [42]:

We select such a linear approximation of Baby Rijndael which has the highest probability bias. Using this approximation, we describe the first three rounds of the cipher in the form

$$P[i_1, i_2, ..., i_a] \oplus D(C, K_4)[j_1, j_2, ..., j_b] = K[k_1, k_2, ..., k_k] \tag{3.7}$$

where $P[i_1, i_2, ..., i_a]$ is the xor of some bits of the plaintext, C is the ciphertext, K_r is the key in the last (r-th) round of the cipher, $D(C, K_r)[j_1, j_2, ..., j_b]$ is the xor of some bits of the cipher's inner state which is generated by performing one decryption round of the ciphertext using key K_r and $K[k_1, k_2, ..., k_k]$ is the xor of some bits of the (master) key. This approximation is satisfied with probability $P = 0.5 + \epsilon$, where ϵ is the linear probability bias of the approximation.

Then we can apply all candidates for key K_r^i against a given set of N plaintext-ciphertext samples encrypted by a fixed key K and for each candidate calculate the number of times T_i the equation 3.7 was satisfied. Let T_{max} be the maximum of all T_i and T_{min} the minimum. Then:

1. If $|T_{max} - \frac{N}{2}| > |T_{min} - \frac{N}{2}|$, then the recovered key is the key corresponding to T_{max} and furthermore guess that $K[k_1, k_2, ..., k_k]$ is 0 if $\epsilon > 0$ or 1 if $\epsilon < 0$.

2. If $|T_{max} - \frac{N}{2}| < |T_{min} - \frac{N}{2}|$, then the recovered key is the key corresponding to T_{min} and furthermore guess that $K[k_1, k_2, ..., k_k]$ is 1 if $\epsilon > 0$ or 0 if $\epsilon < 0$.

3.3.1 Adaptation of Baby Rijndael to Matsui's Algorithm 2

Early in our research when we adapted Baby Rijndael to Matsui's Algorithm 2, we noticed that can significantly reduce the number of non-linear operations used if we modify the description of the cipher so that 1) we switch the order of SubBytes and ShiftRows, and 2) combine two SubBytes and one MixColumns operation into one larger transformation "BigSub":

- We can perform this modification because SubBytes and ShiftRows are completely independent of each other: Each element of the state matrix is processed by SubBytes the same way regardless of its position, and ShiftRows only changes the positions of the whole elements, never of their parts, and never mixes the content of two or more elements.
- Each transformation block of a cipher can be considered a non-linear transformation, even though this is usually undesirable as it increases the number of non-linear elements and thus decreases bias.
- A sequence of transformations can be replaced by a composite transformation, as long as all input and all output bits of all member transformations are included in the composite transformation.

When we performed the linear analysis of the "BigSub" transformation, we discovered that the maximum reachable bias for it is $\pm\frac{1}{4}$, same as for the SubBytes transformation; that means that we can replace two non-linear transformations with one non-linear transformation (which improves the overall bias) while maintaining the bias of the components of the transformation (which doesn't change the overall bias), thus improving the overall bias.

Furthermore, we noticed in our earlier work that the success ratio of the recovery of the last-round key — that is, the probability that a key selected by the Algorithm 2 is the key actually used to encrypt the set of plaintexts — is relatively low. For that reason we decided not to use the guess for $K[k_1, k_2, ..., k_k]$ in our current work; that will be used only when the success rate becomes significantly high.

The overall algorithm for performing linear cryptanalysis is shown in Alg. 3.1.

3.4 Results

During our research, we achieved several relevant results. We created a list of all linear approximations for the Baby Rijndael cipher. We evaluated all of them as to their ability to successfully recover the correct key. We then chose a set of approximations with the highest success rate and tried to improve the probability of success, finding several interesting facts along the way.

3.4.1 Success Rate of Baby Rijndael's Linear Approximations

We created a program which could generate all possible linear approximations of Baby Rijndael using an exhaustive search, and list those approximations with the highest achievable bias — we will call these the *optimal approximations*. We did, however, limit the search in these aspects:

- We only generated approximations for the first three rounds of the Baby Rijndael, in accordance with the requirements of Algorithm 2, where the last round is run "backwards" with all possible candidate keys.

Algorithm 3.1 Linear Cryptanalysis using Matsui's Algorithm 2

```
1: function LinearCryptanalysis(PlainTextMask, InnerStateMask, Samples)
2:     CandidateKeys ← CalculateCandidateKeys(CipherTextMask)
3:     Approximations ← ∅
4:     for all CandidateKey in CandidateKeys do
5:         SatisfiedCount ← 0
6:         for all (PlainText, CipherText) in Samples do
7:             State ← ShiftRows^-1(SubBytes^-1(CipherText xor CandidateKey))
8:             Left ← (PlainText and PlainTextMask)
9:             Right ← (State and CipherTextMask)
10:            if Parity(Right xor Left) = 0 then
11:                SatisfiedCount ← SatisfiedCount + 1
12:            end if
13:        end for
14:        add (CandidateKey, SatisfiedCount) to Approximations
15:    end for
16:    sort Approximations by abs(SatisfiedCount − length(Samples)/2) descending
17:    return Approximations
18: end function
```

- We only focused on approximation whose active bits end in two SubBytes blocks — we will call these the *active SubBytes*. This optimization is also done to facilitate Algorithm 2, as it requires an exhaustive search of all candidate keys and then an exhaustive search of the remaining bits of the full key. With key N bits long, the optimum separation for one-pass linear cryptanalysis is into $N/2$ bits for the candidate keys and $N/2$ bits for the remainder of the key, leading up to the overall complexity of $2 \cdot 2^{N/2}$. It is possible to perform multiple passes of linear cryptanalysis in sequence with fewer active SubBytes in each, but we left this approach for future study.

For Baby Rijndael with four existing SubBytes in each round, there are 6 different classes of linear approximations if we distinguish them by the active SubBytes: counting from left, first two SubBytes are active (we will denote this as the "1100" class), the first and the third is active ("1010") etc. The number of approximations belonging to each class is surprisingly regular, probably owing to the way the SubBytes transformation is constructed.

Then we generated 65536 sets of plaintext-ciphertext samples, one for each existing key, which we will call the *correct key*. In each sample set, there were 65536 plaintext-ciphertext pairs, i.e. our sample sets contained all possible plaintexts and respective ciphertexts for a given key.

Finally, we applied Alg. 3.1 to every sample set to recover a last round key, which we transformed (using a look-up table) to an actual *recovered master key*. We then compared the recovered master key to the correct key, and calculated the *rank of the key* as the

number of recovered master keys we would have to try before we could reach the correct key; that is, if recovered master key equals the correct key, then the rank of the key is 1; if the keys do not match, the rank of the key would be 2, 3 and so on, up to 256 (the correct key was the last key suggested by Alg. 3.1, in other words, the algorithm completely failed). The ranking algorithm is shown in Alg. 3.2

Algorithm 3.2 Rank Candidate Keys generated by linear cryptanalysis

```
1: function RankCandidateKeys(PlainTextMask, InnerStateMask, Samples)
2:     Approximations ←
   ← LinearCryptAnalysis(PlainTextMask, InnerStateMask, Samples)
3:     PreviousApproximation ← NULL
4:     RankedApproximations ← ∅
5:     for all Approximation in Approximations do
6:         if PreviousApproximation = NULL then
7:             Rank ← 0
8:         else if PreviousApproximation.SatisfiedCount =
   = Approximation.SatisfiedCount then
9:             Rank ← PreviousApproximation.SatisfiedCount
10:        else
11:            Rank ← PreviousApproximation.SatisfiedCount + 1
12:        end if
13:        add (Approximation.CandidateKey, Approximation.SatisfiedCount, Rank)
   to RankedApproximations
14:        PreviousApproximation ← Approximation
15:    end for
16:    return RankedApproximations
17: end function
```

This calculation was done for all approximations of the "1010", "1001", "0110" and "0101" classes and to 200 randomly selected[2] approximations of the "1100" and "0011" class. The results are shown in Tab. 3.1.

We can see that the success rate of the approximations, expressed as the rank of the correct key, significantly depends on the active SubBytes of the approximation: While approximations of the "0011", "0110", "1001" and "1100" class are only a little better than a random guess (which would yield the rank of 128), approximations of the "0101" and "1010" class achieve a much better rank. This is contrary to the expectation that all linear approximations with the same bias should be interchangeable as far as their success rate is concerned. It is as yet unclear why Baby Rijndael's approximations with alternating active and passive SubBytes should prove so much better than the other approximation.

For our subsequent tests, we selected the class of approximation "0101", which performed equally well as the "1010" class in the average case and marginally better in the median case.

[2]Using a cryptographically secure PRNG

Table 3.1: Success rate of Baby Rijndael's linear approximations. The lower the average rank, the better can Algorithm 2 recover the correct key. The value of one-half of the number of candidate keys is the worst case, the linear approximations can't determine the correct key better than a random guess.

	Active SubBytes					
	0011	0101	0110	1001	1010	1100
Optimal approximation's probability bias	$\pm\frac{1}{256}$	$\pm\frac{1}{256}$	$\pm\frac{1}{256}$	$\pm\frac{1}{256}$	$\pm\frac{1}{256}$	$\pm\frac{1}{256}$
Nr. of opt. approximations	3840	48	48	48	48	3840
Nr. of candidate keys	256	256	256	256	256	256
Average rank of the correct key	114.75	49.58	111.91	111.90	49.58	114.72
Median rank of the correct key	114.91	49.85	111.77	111.65	49.88	115.08
Std. deviation of the correct key	2.89	6.03	2.23	2.32	6.03	2.77

Table 3.2: Success rate of Baby Rijndael's linear approximations of the "0101" class. The lower the average rank, the better can Algorithm 2 recover the correct key. "Inner state" represents the bits at the beginning of the last round, after performing the ShiftRows transformation. The active bits are counted from the right, that is, the left-most bit is number 15, the right-most bit is number 0.

Active bits		Average	Std. dev.
Plaintext	Inner state	rank	of rank
Best approximations			
0, 2, 3	1, 2, 9, 10, 11	40.27	46.72
12, 14, 15	1, 2, 9, 10, 11	40.37	46.82
8, 10, 11	1, 2, 3, 9, 10	40.38	46.84
4, 6, 7	1, 2, 3, 9, 10	40.43	46.85
Worst approximations			
4, 6, 7	0, 1, 8, 11	57.20	69.50
4, 6	0, 3, 9, 11	57.20	70.50
12, 14, 15	0, 3, 8, 9	57.25	69.55
0, 2, 3	0, 3, 8, 9	57.40	69.65
Average over all 48 approximations			
		49.58	61.04

3.4.2 Success Rate of the Approximations in the "0101" Class

For all linear approximations in the "0101" class, we evaluated the success rate over all existing keys. The best approximation achieved the average rank of 40.27, significantly better than the 57.40 rank of the worst approximation. However, this difference only translates to 0.52 bits of complexity saved by the best approximation over the worst approximation. You can see the best and the worst approximations in Tab. 3.2.

It is interesting to note that if we consider all 48 approximations, the standard deviation

Table 3.3: The average number of bits of the correct key recovered by Baby Rijndael's linear approximations of the "0101" class. The higher the number the better. "Inner state" and "active bits" are defined as in Tab. 3.2.

Active bits		Average nr. of
Plaintext	Inner state	recovered bits
Best approximations		
0	0, 3, 8, 9	4.895
0, 2	1, 3, 8, 11	4.877
8, 10	0, 3, 9, 11	4.876
8	0, 1, 8, 11	4.875
Worst approximations		
12, 14	3, 9, 10	4.421
4, 6	1, 2, 11	4.414
8, 10	1, 2, 11	4.399
0, 2	3, 9, 10	4.398
Average over all 48 approximations		
		4.632

of the rank tends to grow with the average value of the rank. The correlation between the two statistics is not perfect, but it is very high, with Pearson's correlation coefficient of 0.9865.

However, when we consider the average number of correctly recovered bits of the key, that is, the number of bits in the recovered master key which match their respective bits in the correct key, the results are rather discouraging, the number of recovered bits varies between 4.896 (best) and 4.398 (worst) — in other words, on average we are only able to recover a little over one half of the bits correctly, which compares poorly to random guessing. See Tab. 3.3 for details.

We observed that the rank of the correct key often is not 1, as expected by the linear cryptanalysis theory, but rather a different value. It is not unreasonable to expect, though, that the candidate keys with better ranks would have more of their bits correctly guessed than candidate keys with worse ranks. In order to verify this expectation, we tried to calculate a weighted average of up to 10 highest ranking candidate keys, with weights assigned according to the distance of each particular T_i from N (notation as per Matsui's Algorithm 2 above). The number of bits correctly guessed using this method is shown in Tab. 3.4.

The results rather convincingly show that the expectation above is probably not correct and averaging top-ranking candidate keys does not lead to improved accuracy of key recovery.

Table 3.4: The average number of bits of the correct key recovered by Baby Rijndael's linear approximations of the "0101" class, if we calculate a weighted average of bits of a given number of the top-ranking candidate keys. The higher the number the better.

Number of candidate keys used	Average number of bits recovered		
	Best case	Worst case	Average case
1	4.895	4.398	4.632
2	3.345	2.834	3.070
3	4.806	4.385	4.566
4	3.855	3.367	3.604
5	4.111	3.683	3.882
6	2.918	2.485	2.691
7	2.506	2.143	2.319
8	1.610	1.302	1.449
9	1.268	1.022	1.142
10	0.768	0.589	0.677

Table 3.5: The probability of recovery of individual bits of the key, when calculated as the probability that the given bit in a recovered last-round key is correct across all possible keys.

Active bits		Recovered	Probability
Plaintext	Inner state	bit	of recovery
Best approximations			
0, 2	1, 3, 8, 11	3	0.703
8, 10	0, 3, 9, 11	11	0.701
4	0, 1, 8, 11	3	0.683
12	0, 3, 8, 9	11	0.682
0	0, 3, 8, 9	11	0.682
8	0, 1, 8, 11	3	0.679
12, 14, 15	0, 3, 8, 9	11	0.677
10, 11	1, 2, 3, 9, 10	0	0.676
Worst approximations			
12, 14, 15	1, 2, 9, 10, 11	11	0.511
4, 6, 7	1, 2, 3, 9, 10	3	0.510
8, 10	1, 2, 11	11	0.493
0, 2	3, 9, 10	3	0.491

3.4.3 Recovery of individual key bits

We tried another approach to improve the accuracy of key recovery: We modified the Algorithm 2 to recover individual bits rather than the whole candidate key. The modification is straightforward: We perform Algorithm 2 as usual, but only consider a subset of the bits of the recovered master key. We then evaluate how these bits match against the respective bits of the correct key, hopefully improving the accuracy.

We performed exhaustive testing of this idea on all the "0101" approximations, all keys and a complete plaintext-ciphertext sample set for each combination of an approximation and a key. We measured the probability that a given single bit of the recovered last-round key is correct, i.e. equal to the respective bit of the correct last-round key, across all possible keys. Unlike the previous experiments, here we only consider the match between the bits of the recovered last-round key and the correct last-round key; the rank of the correct last-round key is without meaning in this approach.

Tab. 3.5 shows that this approach is promising, in several important aspects:

- There are significant variations between individual approximations, further supporting the expectation that there are other factors to an approximation's power than just the bias.
- Some key bits are easier to recover than other bits, which suggests that there may exist a set of keys in the Rijndael family of ciphers which are more susceptible to the linear cryptanalysis. When taken as a whole, bits 3 and 11 are the easiest to recover - among the 20 most successful approximations, 8 recovered bit 11, 7 recovered bit 3, 3 recovered bit 0 and 2 recovered bit 8. The most successful approximation for recovery of bit 10 was 29th, etc.
- With the best approximations, we can achieve a significantly higher success rate than when recovering the entire candidate key. This means that recovery of individual bits is indeed possible, and these recovered bits can then be used to facilitate recovery of other bits, possibly using different cryptanalytic techniques.

3.4.4 Using Multiple Approximations to Improve the Success Rate

Thanks to the small size of Baby Rijndael, we were able to perform an exhaustive analysis of a number of its aspects. We were able to find all optimal linear approximations of the cipher and discovered that there is a great number of these optimal approximations. This will be useful further on, when we consider the combined effect of multiple linear approximations on the same sample set — as Kaliski and Robshaw suggest [36], we can use multiple approximations to generate a new statistic for our set of candidate keys, one which would reduce variance of the result and thus decrease the size of the required sample set.

We adapted the idea to the concept of Baby Rijndael and the recovery of individual key bits. We used a simple Alg. 3.3 for estimating the value of a bit using majority voting

and then used it to try to recover every bit position available in the key mask individually as per Alg. 3.4.

Algorithm 3.3 Calculate average bit value

```
1: function CalcBit(BitSum, BitCount)
2:     if 2 · BitSum > BitCount then
3:         return 1
4:     else if 2 · BitSum < BitCount then
5:         return 0
6:     else
7:         Raise an error
8:     end if
9: end function
```

Algorithm 3.4 One key bit recovery using multiple linear approximations

```
1: function OneKeyBitRecovery(Approximations, MasterKey, BitPosition)
2:     Samples ← GenerateSamples(MasterKey)
3:     LastRoundKey ← KeyExpansion(MasterKey, NumberOfRounds − 1)
4:     BitSum ← 0
5:     BitCount ← 0
6:     for all Approximation in Approximations do
7:         ApproxBitSum ← 0
8:         ApproxBitCount ← 0
9:         RankedCandidates ← RankCandidateKeys(
   Approximation.PlainTextMask, Approximation.InnerStateMask, Samples)
10:        for all CandidateKey in RankedCandidates do
11:            if CandidateKey.Rank = 0 then
12:                if GetBitValue(CandidateKey.Key, BitPosition) =
   = CallGetBitValueLastRoundKey, BitPosition then
13:                    ApproxBitSum ← ApproxBitSum + 1
14:                end if
15:                ApproxBitCount ← ApproxBitCount + 1
16:            end if
17:        end for
18:        BitSum ← BitSum + CalcBit(ApproxBitSum, ApproxBitCount)
19:        BitCount ← BitCount + 1
20:    end for
21:    return CalcBit(BitSum, BitCount)
22: end function
```

We performed the test for bits 0, 3, 8 and 11, which were shown to be the most easily recoverable key bits in our previous tests. Five best approximations were used for each bit

and an average success rate was measured over all existing master keys. The results are stored in `Recovery_of_bit_??.log` files.

This approach, unfortunately, leads to a large number of indeterminate results in cases where the correct and incorrect results are evenly matched. To improve that, a modification was made which would work with "fuzzy" (indeterminate) bits, hoping that the remaining approximations would overcome these matched opposites. The bit-calculating algorithm is shown in Alg. 3.5, the key recovery algorithm in Alg. 3.6.

Algorithm 3.5 Calculate fuzzy average bit value

```
1: function CalcFuzzyBit(BitSum, BitCount)
2:     if 2 · BitSum > BitCount then
3:         return 1
4:     else if 2 · BitSum < BitCount then
5:         return 0
6:     else
7:         return 0.5
8:     end if
9: end function
```

This approach led to a marked decrease in the indeterminate cases, but still some remained, as can be seen in the `NonWeightedRecovery_of_bit_??.log` files. A further modification introduced a weight for each linear approximation, where the weight is given as the probability that the approximation would correctly calculate the bit over all possible keys. The algorithm is shown in Alg. 3.7.

This algorithm completely removes the indeterminate bits, as seen in the `Weighted-Recovery_of_bit_??.log` files. However, the overall efficiency is not significantly improved — the indeterminate bits simply split into the "correct" and "incorrect" groups without any apparent reason for either category.

Algorithm 3.6 One key bit recovery with fuzzy bits

```
1: function OneKeyBitRecoveryFuzzy(Approximations, MasterKey, BitPosition)
2:     Samples ← GenerateSamples(MasterKey)
3:     LastRoundKey ← KeyExpansion(MasterKey, NumberOfRounds − 1)
4:     BitSum ← 0
5:     BitCount ← 0
6:     for all Approximation in Approximations do
7:         ApproxBitSum ← 0
8:         ApproxBitCount ← 0
9:         RankedCandidates ← RankCandidateKeys(
   Approximation.PlainTextMask, Approximation.InnerStateMask, Samples)
10:        for all CandidateKey in RankedCandidates do
11:            if CandidateKey.Rank = 0 then
12:                if GetBitValue(CandidateKey.Key, BitPosition) =
   = CallGetBitValueLastRoundKey, BitPosition then
13:                    ApproxBitSum ← ApproxBitSum + 1
14:                end if
15:                ApproxBitCount ← ApproxBitCount + 1
16:            end if
17:        end for
18:        BitSum ← BitSum + CalcFuzzyBit(ApproxBitSum, ApproxBitCount)
19:        BitCount ← BitCount + 1
20:    end for
21:    return CalcFuzzyBit(BitSum, BitCount)
22: end function
```

3.5 Discussion and Conclusion

An interesting aspect of the optimal linear approximations is their varying ability to recover the master key. We can divide the approximations into several classes, and through exhaustive testing of all possible keys we showed that approximations from some of these classes are, on average, significantly more successful than approximations from other classes. This suggests that, when we consider linear cryptanalysis, we need to pay attention not only to the probability bias, but also the choice among several possible approximations. We don't as yet know why the classes "0101" and "1010" are so much better than the others, but we intend to find out.

Even within a class of approximations, the variations in key recovery abilities are quite surprising. We would definitely like to precisely measure, what makes the "good" approximations different from the "bad" ones. If we could discover some metric which would let us choose the best approximation in advance, without having to perform intensive calculations, it would certainly be a useful result in its own.

It came as a distinct surprise to us that individual key bits show quite a large difference

Algorithm 3.7 Weighted one key bit recovery with fuzzy bits

```
1: function OneKeyBitRecoveryFuzzy(Approximations, MasterKey, BitPosition)
2:     Samples ← GenerateSamples(MasterKey)
3:     LastRoundKey ← KeyExpansion(MasterKey, NumberOfRounds − 1)
4:     BitSum ← 0
5:     BitCount ← 0
6:     for all Approximation in Approximations do
7:         ApproxBitSum ← 0
8:         ApproxBitCount ← 0
9:         RankedCandidates ← RankCandidateKeys(
   Approximation.PlainTextMask, Approximation.InnerStateMask, Samples)
10:        for all CandidateKey in RankedCandidates do
11:            if CandidateKey.Rank = 0 then
12:                if GetBitValue(CandidateKey.Key, BitPosition) =
   = CallGetBitValueLastRoundKey, BitPosition then
13:                    ApproxBitSum ← ApproxBitSum + Approximation.Weight
14:                end if
15:                ApproxBitCount ← ApproxBitCount + Approximation.Weight
16:            end if
17:        end for
18:        BitSum ← BitSum + Approximation.Weight ·
   CalcFuzzyBit(ApproxBitSum, ApproxBitCount)
19:        BitCount ← BitCount + Approximation.Weight
20:    end for
21:    return CalcFuzzyBit(BitSum, BitCount)
22: end function
```

in their ability of being successfully recovered. We weren't able to formulate the reasons behind this behavior so far, but this is also one of our research targets. We are particularly eager to continue our research in this area, because we would like to explore our idea of using information revealed by one cryptanalytic technique (e.g. linear cryptanalysis in this case) to enhance the power of another technique (e.g. algebraic cryptanalysis) — and then inject the results back to the original technique. It seems possible to achieve synergistic effects here.

The ultimate goal, of course, isn't cryptanalysis of Baby Rijndael, however interesting it may be. We hope, though, that the principles we discover will eventually allow us to attack even the full Rijndael itself — or, failing that, at least give us some idea of which approaches do have a hope of succeeding and which do not.

It needs to be stressed that while our approach did reveal some interesting information, it was not successful in actually breaking the cipher — specifically, it couldn't recover the key with a lesser effort than brute force trial of all keys would. Quite the opposite, in fact — in order to achieve a reasonable level of success even in the best circumstances, we had

to much perform more work than if we simply tried out all keys, which would additionally use simpler operations (i.e. the exponential complexity would use a lower base), much less memory, only need one plaintext-ciphertext sample and in addition to all that it would be assured success after trying all the combinations, something the key recovery using linear cryptanalysis could not do. That, however, is not a bad thing — it convincingly demonstrates that Baby Rijndael is indeed resistant to linear cryptanalysis even when conditions are extremely skewed in its disfavor. We can expect even more resistance with the standard Rijndael which does not contain these additional weaknesses.

Chapter 4

Automatic Detection of AES Operations

This chapter is based on the results obtained in cooperation with Jonatan Matějka, first presented in his master's thesis [44] and then revised, expanded and modified for international presentation [A.2]. A further expanded version has been requested for a journal publication [A.3].

In modern IT, security is no longer considered an afterthought; quite the contrary, security is a mandatory feature of many hardware and software products. In recent months we have seen a major push for increasing security which manifested e.g. in the announcements by browser manufacturers that they plan to abolish or at least minimize unsecured web traffic in near future [20], or in the deprecation of TLS protocols versions 1.0 and 1.1 from all major browsers and other applications in early 2020 followed by IETF soon after [49]. At the same time, privacy concerns of the users are on the rise [1].

While the improvements in security often improve privacy as well, this is not always the case. In particular, the increased use of encryption to protect communication from outsiders can also remove control of the legitimate users over the transmitted data as they are no longer able to easily monitor the contents of the network traffic [62]. This is especially dangerous in case of applications which are considered legitimate by users but do not provide any means of verification what is being sent over the network. For example, an operating system or an application may very well propose to send telemetry data to the developer to improve the user experience, but if the source code of that software is not available, the user cannot easily verify what data is actually being collected. The user could, of course, resort to the techniques of reverse engineering, but that almost always requires so much effort as to make this approach prohibitively expensive.

In this chapter, we propose an alternative solution to this problem that makes use of the dynamic analysis to recover the sensitive data generated or used by a third-party application: By exploiting the fact that applications do have a significant level of control over their child processes which essentially creates a particularly powerful side channel, we propose a tool that would run an encrypted application (the target) as its subprocess and manipulate its control flow in such a way that the AES encryption keys and even the plaintexts are revealed. The process is fully automatic and highly universal, at the cost of

performance.

4.1 Common Implementations of AES

In this section we will briefly introduce the common ways of implementing AES on Intel-based architectures.

AES is probably the most commonly used block cipher in the world. Its design and structure is defined in [19] and can theoretically be implemented according to that as well. It is, however, common to tailor the implementation to the specific features in the target CPU. Note that since the key-expansion process is generally a one-time operation or at least is performed much more rarely than the actual encryption, it is usually not optimized on the algorithmic level and if any optimization is used at all, it's left to the compiler and its choice of instructions and their ordering. For that reason, we will only discuss the operations used in the actual encryption/decryption.

When targeting the Intel architectures (IA-32, Intel 64), one of the following approaches is usually taken:

4.1.1 The Naïve Implementation

The naïve implementation of AES follows the operations described in the cipher's specification [50], i.e. key-expansion, sub-bytes, shift-rows, mix-columns and add-round-key, in the designated order and the specified number of repetitions.

The sub-bytes operation is commonly implemented through a lookup table of 256 values where the input to sub-bytes is used as an index to the table and the output value is read from that location in the table; another such table is used for the inverse of sub-bytes. That removes the need for calculating inverses in AES's Galois field.

Shift-rows is typically implemented as described as reordering of the bytes in the encryption state.

Mix-columns may either be implemented as straight table multiplication or optimized by pre-calculating the necessary multiples for each possible input value. For encryption, we need multiples of two and three, for decryption multiples of 9, 11, 13 and 14. That can be done by using 6 pre-calculated tables with the same structure.

Add-round-key is again usually implemented in a straightforward XOR, although multiple bytes may be processed at once using e.g. 32-bit XOR instructions.

An example of this optimized approach can be found in [40].

4.1.2 Implementation Using T-tables

Assuming that the target CPU architecture supports 32-bit instructions, further pre-calculation allows us to optimize the encryption process to just four lookups and four XOR operations per column per round, as described in [19]:

Given an input state of $A = [a_{i,j}], 0 \leq i < 4, 0 \leq j < N_b$, expanded round key $K = [k_{i,j}], 0 \leq i < 4, 0 \leq j < N_b$ where N_b is the number of columns of A, we can express the encrypted output state $D = [d_{i,j}], 0 \leq i < 4, 0 \leq j < N_b$ as:

$$\begin{pmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \end{pmatrix} = \sum_{i=0}^{3} T_i(a_{i,j+C_i}) + \begin{pmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{pmatrix}, \tag{4.1}$$

where + is the operation XOR, C_i are the respective left shifts for the i-th row of the state (e.g. 0, 1, 2 and 3 respectively for the 128-bit key version of AES) and T_i are pre-calculated as:

$$\begin{aligned} T_0(x) = \begin{pmatrix} 2 \cdot S[x] \\ 1 \cdot S[x] \\ 1 \cdot S[x] \\ 3 \cdot S[x] \end{pmatrix}, T_1(x) = \begin{pmatrix} 3 \cdot S[x] \\ 2 \cdot S[x] \\ 1 \cdot S[x] \\ 1 \cdot S[x] \end{pmatrix}, \\ T_2(x) = \begin{pmatrix} 1 \cdot S[x] \\ 3 \cdot S[x] \\ 2 \cdot S[x] \\ 1 \cdot S[x] \end{pmatrix}, T_3(x) = \begin{pmatrix} 1 \cdot S[x] \\ 1 \cdot S[x] \\ 3 \cdot S[x] \\ 2 \cdot S[x] \end{pmatrix} \end{aligned} \tag{4.2}$$

for all possible byte values of x, given that $S[x]$ is the result of the sub-bytes transformation of x.

The T-tables can be further compressed by observing that they are in fact rotated versions of each other and as a result only one of them needs to be pre-calculated, the others can be obtained from it through the use of rotation.

Further compression is possible if an unaligned data access is possible because then the tables can be stored overlapped.[55]

4.1.3 Implementation Using Bit-Slicing

The bit-slicing implementation [37] is inspired by the hardware-based implementations of AES: The cipher's state is represented as a series of bits and the operations usually performed by hardware gates are simulated using logical operations. This approach has several significant advantages: it does not need any pre-calculated tables (the lookups are replaced by series of logical operations), reducing memory requirements of the algorithm, the algorithm takes a constant time in clock cycles as the operation sequences are fixed regardless of any variations in input data, and timing attacks on memory access are difficult if not impossible [43]. The chief disadvantage is the reduced speed of the algorithm, although that can be offset if vector instructions (such as those provided by the MMX, SSE, SSE2 etc. instruction sets) are used and we can process multiple blocks of the cipher in parallel, e.g. in the CTR encryption mode: we would "slice" bits from eight independent states into eight 128-bit registers and process them in parallel.

4.1.4 Implementation Using AES-NI

In 2008, Intel introduced a new instruction set extension for a hardware support of AES encryption and decryption [28]. It consists of 6 instruction which provide encryption and decryption of a single round of AES as well as support for key expansion and the inverse Mix-columns operation. Much like bit-slicing, this technique provides high security (e.g. resistance to timing attacks) and low memory footprint because it does not use any in-memory tables, and in comparison to bit-slicing provides a very high performance due to its hardware-based implementation. Another benefit is the very simple implementation, although care needs to be taken to verify that the AES-NI instruction set is actually available at the CPU where the code is running – customarily, the code would check for the presence of these instructions and then branch to either an AES-NI based version or a traditional version based on one of the approaches shown above.

4.2 Detecting and Recovering AES Through the Use of S-Box

Let's first tackle the purely software-based approaches, i.e. the naïve implementation and the T-tables implementation. In both cases the application makes use of pre-calculated tables during the actual encryption and decryption, although the tables themselves may be dynamically calculated using the cipher's specification during the crypto engine initialization. Our approach is based on attaching to the target application as a debugger and then making use of the debugging APIs [45] to monitor data accesses to these tables with the intention to deduce both the key and the data from the order and the precise location of the accesses.

In order to achieve this goal, four subproblems need to be resolved: First we need to locate the substitution tables in the process' memory. Then we need to establish a method for getting notified about the process trying to access these tables. When that is one, we need a technique for determining the type of the access (during encryption/decryption, during key expansion, and irrelevant accesses due to other concerns), and finally we need a method for putting it all together and extracting the actual sensitive data from the accesses. In fact, it turns out that the last two subproblems are closely related and only using them in combination can reveal the full information.

4.2.1 Locating Tables

In order to be able to monitor accesses to the tables, we need to locate them in the target application's memory first. To do that, we use `VirtualQueryEx` function to get the list of memory pages belonging to the analyzed process, copy these pages to our memory using `ReadProcessMemory` and then search them. Since the tables may be stored in a variety of fashion, we do not compare memory blocks to known values but rather study the relationships between bytes – we are looking for multiplications of the original SubByte

table[1] with common interleaving (1 byte for SubBytes, 4 bytes for T-Tables and 8 bytes for overlapping T-Tables).

With many applications, it is sufficient to perform the search only once at the beginning of the application because the tables are statically compiled into the application. Some applications, however, build these tables at least partially dynamically during their runtime – e.g. to calculate the T-tables from the statically stored SubBytes table or to load a dynamic library which contains these tables. To facilitate support for these applications, we perform the search repeatedly using a background thread; currently no performance optimizations are performed for this search, but it seems likely that some would be applicable.

4.2.2 Monitoring Access

Once we have located the substitution tables, we need to monitor access to them. Based on the specific hardware used, there may be different ways of doing so. On the Intel architecture, we could use debug registers [32] for this purpose, but unfortunately only for memory locations of up to 8 bytes each could be monitored, which is not enough to detect all accesses – even SubBytes is at least 256 bytes long, T-tables even longer.

Instead, we decided to make use the concept of memory paging and memory page protection: Once we know in which memory pages the substitution tables reside, we remove all access from these pages using `VirtualProtectEx` by adding the `PAGE_GUARD` flag. When that was done, any access to any location within the memory page causes a page fault exception before passing it to the application itself the active debugger – our tool – is notified about it through a debug event. Specifically, we learn of the actual memory location and the type of access (read, write, execute) that caused the fault. We can then verify whether the access was a substitution table access and if so, process it accordingly.

Obviously, we must allow the application to actually perform the table access so that it can continue in its execution. We achieve that by temporarily removing the `PAGE_GUARD` flag from the affected memory page, enabling the single-step (trap) flag in the thread's `FLAGS` register and resuming the thread; after a single instruction the single-step flag causes another debugging event which we capture and restore both the memory protection by adding the `PAGE_GUARD` flag to the memory page and the standard thread execution by clearing the single-step flag from `FLAGS`.

4.2.2.1 Special Considerations

While the monitoring process is fairly straightforward, care needs to be taken to facilitate several special situations.

In particular, we need to consider the possibility that the substitution tables are stored in the code segment rather than data segment, such as in [52]. In that case an attempt to execute an instruction from the same memory page will cause a page fault because the instruction itself cannot be read due to the protection settings. That can be solved by

[1] 1, 2 and 3 for encryption tables and 1, 9, 11, 13 and 14 for decryption tables

checking whether the access occurred inside a substitution table or whether it occurred somewhere else within the monitored memory page, and in such a case simply removing the protections, single-stepping the instruction and then restoring the protections. It will degrade the performance significantly but the code will function as expected.

Unfortunately, that is not the case if the instruction that accesses the substitution table is located within the same memory page as the substitution table itself: In this case, we would fail to detect the table access because we removed the protection in order to execute the instruction and will only restore it after the instruction has completed, i.e. after the table access. We solve this problem by decoding the instruction in software using a third-party library and determining whether it is this particular case; if it is, we emulate the instruction rather than execute it directly.

4.2.3 Monitoring Key Expansion

During key expansion, the substitution tables are used to perform a SubWord operation:

$$W_i = \begin{cases} K_i & \text{for } i < N_k \\ W_{i-N_k} + r(s(W_{i-1})) + rcon_{i/N_k} & \text{for } i \geq N_k, i \equiv 0 \pmod{N_k} \\ W_{i-N_k} + s(W_{i-1}) & \text{for } i \geq N_k, N_k > 6, i \equiv 4 \pmod{N_k} \\ W_{i-N_k} + W_{i-1} & \text{otherwise} \end{cases} \tag{4.3}$$

Here, K_i is the i-th column of the master key, N_k is the number of columns of the master key, W_i is the i-th column of the expanded key and functions r and w as well as the constant $rcon$ are defined as follows:

$$\begin{aligned} r(w) &= r\left(\begin{pmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \end{pmatrix}\right) = \begin{pmatrix} w_1 \\ w_2 \\ w_3 \\ w_0 \end{pmatrix} \\ s(w) &= s\left(\begin{pmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \end{pmatrix}\right) = \begin{pmatrix} S[w_0] \\ S[w_1] \\ S[w_2] \\ S[w_3] \end{pmatrix} \\ rcon_i &= \begin{pmatrix} 2^{i-1} \\ 0 \\ 0 \\ 0 \end{pmatrix}, \text{ for } i \in \mathbb{N}^+ \end{aligned} \tag{4.4}$$

In order to properly recover the key, we need to make several assumptions:

- While key expansion is being performed, no other access to substitution tables is performed except through the SubWord function.

- SubWord calls are performed in order of the columns in the expanded key.

- Accesses to the substitution tables are the same in all SubWord calls.

On the other hand, we do not make any assumption on the order in which the bytes in a word are being substituted – that might be influenced e.g. by aggressive optimizations on the part of the compiler while building the target application. We can, however, determine the proper ordering by verifying that the dependencies between columns do exist as expected.

The dependencies must be calculated separately for each size of the key. For example, with AES-128 we can make use of columns $\{x \mid x = 3 + 4k, k \in \mathbb{N}\}$ of the expanded key which depend on previous columns as shown in Fig. 4.1. Then:

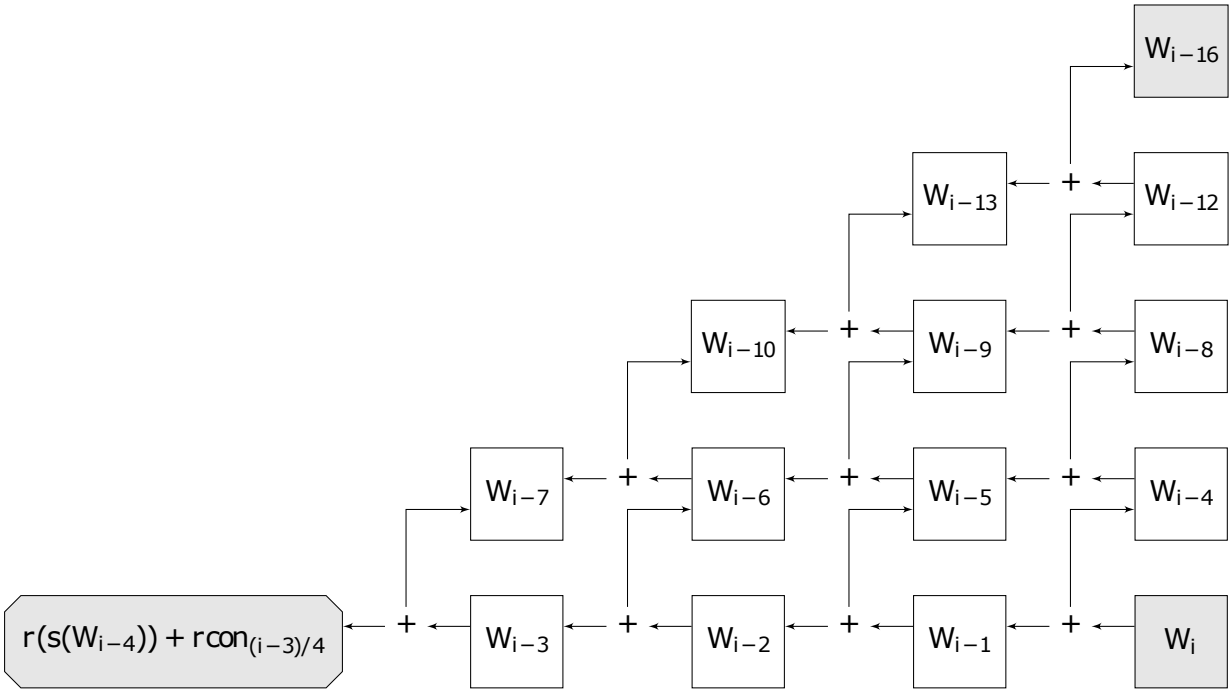

Figure 4.1: Dependency of columns in the expanded 128-bit key. The grayed boxes describe the final expression. [A.2]

$$\begin{aligned} W_i =& W_{i-4} + W_{i-1} = \\ =& W_{i-8} + W_{i-5} + W_{i-5} + W_{i-2} = \\ =& W_{i-8} + W_{i-2} = \\ =& W_{i-12} + W_{i-9} + W_{i-6} + W_{i-3} = \\ =& W_{i-16} + W_{i-13} + W_{i-13} + W_{i-10} + W_{i-10} \\ & + W_{i-7} + W_{i-7} + r(s(W_{i-4})) + rcon_{(i-3)/4} = \\ =& W_{i-16} + r(s(W_{i-4})) + rcon_{(i-3)/4} \end{aligned} \tag{4.5}$$

To detect these dependencies we then require five successive key columns. Since AES-128 performs 10 SubWord operations, we can produce six equations which describe the dependencies between all columns:

$$\begin{aligned} W_{19} &= W_3 + r(s(W_{15})) + rcon_4 \\ &\vdots \\ W_{39} &= W_{23} + r(s(W_{35})) + rcon_9 \end{aligned} \tag{4.6}$$

If the captured table accesses do not adhere to these expressions, then we know that the accesses are not a part of the key expansion process or the assumptions above have been violated. We can make use of this fact by finding the correct ordering of bytes in each word by simply trying them all and checking which leads to satisfying all the expressions.

In this fashion we can can recover every fourth column of the expanded key $W_3, W_7, ..., W_{35}$. Then we can make use of the algorithm of key expansion to calculate the remaining columns, e.g. $W_{i+3} = W_i + W_{i+4}$ for $i \in 3, 7, ..., 35$, $W_i = W_{i+4} + r(s(W_{i+3})) + rcon_{(i+4)/4}$ for $i \in 0, 4, 8$ etc., eventually recovering all the columns of the key.

The key for AES-192 can be recovered in a similar fashion, although only two equations can be used to verify that we are indeed performing key expansion:

$$\begin{aligned} W_{41} &= W_5 + W_{17} + W_{29} + r(s(W_{35})) + rcon_6 \\ W_{47} &= W_{11} + W_{23} + W_{35} + r(s(W_{41})) + rcon_7 \end{aligned} \tag{4.7}$$

With AES-256, the recovery is complicated by the fact that not all columns which entered SubWord can be used for the expression of dependencies – we know the value of W_{27} and W_{31}, but we can not express it using other columns:

$$\begin{aligned} W_{39} &= W_7 + s(W_{35}) \\ W_{47} &= W_{15} + s(W_{43}) \\ W_{55} &= W_{23} + s(W_{51}) \\ W_{43} &= W_{11} + r(s(W_{39})) + rcon_5 \\ W_{51} &= W_{19} + r(s(W_{47})) + rcon_6 \end{aligned} \tag{4.8}$$

As a result, we do not have sufficient information to calculate the correct ordering of the bytes in a word; W_{27} and W_{31} allow for $4! = 24$ different valid orderings each, giving us $24^2 = 576$ different keys which all satisfy the defined expressions. This obstacle can be overcome by deferring the final calculation of the ordering until the encryption phase, behavior of which will help us detect which specific key was actually used.

4.2.4 Monitoring Encryption

During encryption we will observe substitution table access in every round of the cipher. Assume the following:

- While encryption is being performed, no other access to substitution tables is performed except by that block's encryption.
- All substitution table accesses are ordered exactly as the rounds themselves.
- No two rounds overlap.
- The data is being encrypted with the key expanded in the last monitored Key Expansion phase.

We do not make any assumptions on the order of accesses within one round.

The first input to SubBytes within a round is created simply as a sum of the plaintext and the first round key. It is passed through SubBytes and the output is then processed according to the cipher's specifications (rows shifted, columns mixed, next round key added) and forms the input to the second round's SubBytes. Repeat the process for the rest of the rounds, skipping MixColumns in the last round.

With the assumptions above, we know the expanded key, except possibly the ordering in some of its columns. We can make use of this information to determine the proper ordering of the states. Given S the input state of one round's SubBytes, T the output state of the previous round's SubBytes and K the round key, we know that:

$$S = \text{MixColumns}(\text{ShiftRows}(T)) + K \tag{4.9}$$

Unfortunately, we do not know the ordering of SubBytes calls for the individual bytes of states S and T. We can, however, re-formulate and relax the expression as:

$$\forall x : x \in \text{InvMixColumns}(S + K) \iff x \in T \tag{4.10}$$

We could now check all 16! possible permutations of state S and locate the matching one, but that would require quite a lot of computational power. We can, however, further relax the expression and apply it to each column of the state separately:

$$\forall x : x \in \text{InvMixColumns}(S_i + K_i) \implies x \in T \tag{4.11}$$

Now we only need to check $4 \times \frac{16!}{(16-4)!}$ variations of the ordering of S. For each selection we verify that all of its bytes appear in T. If that is not the case, then we know that we are not using the correct key. Otherwise we can apply the same reasoning to the next (or previous) round and express the condition on the whole sequence. E.g. if S was the SubBytes output of the last round and T its input, we can now focus on U the output of the second-to-last round's SubBytes, T its input and L its key. Then:

$$\forall x : x \in \text{InvMixColumns}(U_i + L_i) \implies x \in V \tag{4.12}$$

We can substitute for U and express the left side as:

$$\text{InvMixColumns}(\text{InvSubBytes}(T)_i + L_i) \tag{4.13}$$

Substitute for T and again express the left side as:

$$\begin{aligned} &\text{InvMixColumns}(\text{InvSubBytes}(\text{InvShiftRows}(\\ &\qquad \text{InvMixColumns}(S + K))_i + L_i) \end{aligned} \tag{4.14}$$

And so on for all n rounds of the cipher, yielding $n - 1$ conditions. We can now use these conditions to find the correct ordering of the bytes in each intermediate state. Once we recover the first state, we can get the plaintext by adding the first round key to it.

In the previous chapter we noted that it may not be possible to get the correct ordering of the whole key, e.g. in AES-256. Instead, we only recovered a set of candidate keys. It's clear we could use them all in the state-ordering calculations above, but that would lead to a significant performance penalty. Instead, we can perform these calculations just for the fourth columns of each state, because we have recovered the most information for these: With AES-128, we know the fourth columns of all round keys except for the last, with AES-256 we know the fourth columns of all round keys except for the first and the last, and with AES-192 we know the fourth columns of each third round's key and we can calculate the others from them. We do not need to know the actual permutation of the key, because we can test for all of them if necessary. By applying these keys to the conditions on states, we can conclusively state which keys could not have led to the observed substitution table accesses.

4.2.5 Results and Discussion

In order to demonstrate this approach we created application `AesSniffer`. It is written in C++ and consists of three main parts: A system-dependent library for performing the debugging and memory access work, a system-independent library for recovering keys and plaintexts and a console tool for performing these tasks on third-party applications. This organization allows for a simple adaptation of the tool to different operating systems: while the supplied application is intended for Microsoft Windows, it is possible to adapt it to other OSes by reimplementing the debugging core and the user interface while keeping the recovery part unchanged – or improve the recovery part and apply it to all variants of the application.

4.2.5.1 Library and Application Tests

The tests were performed using Microsoft Windows 7 SP1 x86 in a virtual machine provided by Oracle VM VirtualBox with AES-NI and SSEx instructions disabled. Several popular cryptographic libraries were tested: OpenSSL[2], CryptoPP[3], Botan[4] and WinCrypt[5]; in all cases a simple application for encrypting and decrypting a sample block in the ECB operation mode with a random 128-bit, 192-bit and 256-bit key. As a further test, two existing third-party applications which use their own implementation of AES, were tested: 7-Zip[6] and Putty[7]; both of these applications use the CTR operation mode. Finally, we tested our application's ability to recover data sent and received by PowerShell's `Invoke-WebRequest` command over the HTTPS protocol using the CBC operation mode.

In all of these cases, the application was successful in recovering both the key and the plaintext, although with some limitations.

OpenSSL: All encryption and decryption of data was successfully detected and all keys and data were recovered. We did encounter 8 unrecognized accesses to the substitution tables due to the cache prefetch code which is a part of the OpenSSL implementation.

CryptoPP: The T-tables used by the library are calculated at runtime from the standard SubBytes tables. While these tables were eventually found by our application, accesses to them detected and data and keys recovered, this process did consume some time during which some encryption was already performed, leading to the loss of the early data. We also noticed 256 unrecognized accesses to the SubBytes table while the T-tables were being constructed.

Botan: Much like CryptoPP, Botan also calculates the T-tables at runtime, leading to the loss of early data before the calculated T-tables could have been found. Other than that, our application was able to detect all encryptions and recover both the keys and the data.

WinCrypt: WinCrypt is a part of the Windows family of operation systems. We were able to recover all keys and data, but we did encounter an error in the library shipped with Windows 7 and Windows 8.1: After the key expansion, four unexpected accesses to the substitution table were encountered, probably as a result of an unnecessary SubWord call for the last column of the expanded key, because the data were successfully recovered regardless. In Windows 10, no such accesses were observed.

7-Zip: A popular compression utility 7-Zip supports encryption of the archives using the AES cipher in the CTR mode. We performed a test with a file consisting of 32 zero bytes (two AES blocks) in the "Store" mode (without compression). Our application successfully detected two encryptions; both used the same key and the plaintext were

[2]https://www.openssl.org/
[3]https://www.cryptopp.com/
[4]https://botan.randombit.net/
[5]https://botan.randombit.net/
[6]https://www.7-zip.org/
[7]https://www.putty.org/

two successive counter values (`0x01, 0x00, 0x00, 0x00, ...` and `0x02, 0x00, 0x00, 0x00, ...`), as expected.

Putty: A GUI re-implementation of the SSH protocol for Windows, Putty uses AES (and other ciphers) to encrypt the transferred data between the client and the server. We attempted to recover data from a connection where AES-256 in the CTR mode was the agreed-upon cipher between client and server. Two distinct key expansions were detected and recovered as well as a lot of encryptions. After we XORed the recovered plaintexts with the encrypted data captured by WireShark, we were able to observe data structure expected in the unencrypted contents of the SSH protocol.

PowerShell: Microsoft PowerShell is a scripting language which, among other things, supports reading web data using the HTTPS protocol using the `Invoke-WebRequest` command. When we enforced the use of TLS version 1.0, the client and server agreed upon using the AES-128 cipher in the CBC mode. Our application successfully detected both the key expansion and the encryption as well as decryption of data and we were able to verify that the recovered plaintext contained the expected data of the HTTP protocol.

4.2.5.2 Performance Tests

During our tests we measured the speed of our application in different scenarios using 7-Zip. The tool was chosen because it allows precise specification of the size of the data as well as precise measurement of time; at the same time it is a real-world application and as such can provide a real-world benchmark, unlike a custom benchmarking application which would be heavily dependent on the actual organization of the AES code (e.g. whether the substitution tables were located in the same memory page(s) as some other frequently used data or code items). We created files of 256, 4096 and 65536 bytes and measured how long did 7-Zip take processing these files without compression but with AES-256 encryption in different scenarios based on our application's settings. The results can be seen in Tab. 4.1.

File size [B]	256	4096	65536
Time to process [s]			
Without AesSniffer	0.06	0.06	0.15
AesSniffer, no detection	148.83	150.96	166.93
AesSniffer, only AES-256	150.06	182.26	329.56
AesSniffer, AES-256 and 128	159.52	204.53	347.51
AesSniffer, full detection	199.97	662.81	8138.84
Simple tracer	38964.26	–	–

Table 4.1: Performance penalty of AES detection in 7-Zip based on the size of encrypted data and the settings for the detection. The "Simple Tracer" scenario represents a minimal trap-after-each-instruction implementation without any additional logic.

It is apparent that the presence of our application carries a significant performance penalty even if no key- and plaintext-recovery is being done. This is caused by the fact

that on any access to the memory page containing a detected substitution table causes a page fault, a number of context switches between the application, our AesSniffer and the operating system, and several `VirtualProtectEx` calls, not to mention the possible need for using a software decoder of the affected instruction. While this penalty can be reduced if the monitored application used a friendlier memory layout (e.g. the substitution tables would be located in dedicated memory pages), the opposite is also possible – if, for example, the substitution table occupies the same memory page as a virtual method table of some frequently used object class, the penalty could be much more pronounced, even more so if some frequently used code was located there as well.

Another significant increase in the processing time can be observed if the detection of AES-192 is activated. The reason for that is that with AES-192 there are far more possible permutations of the key than with AES-128 and AES-256 because the columns of the 192-bit key depend on five other columns rather than three columns with 128- and 256-bit keys.

Finally, the size of the data to be encrypted obviously increases the overall time because more accesses to the substitution table are required – 160 accesses per block in case of AES-128, 192 accesses per block in case of AES-192 and 224 accesses per block in case of AES-256.

While these penalties may seem overwhelming, it should be noted that they are still far more manageable than other dynamic approaches. We implemented a very simple tracer into our application, one which forces single-stepping of all instructions without any additional processing (i.e. no AES detection at all). Encryption of a 256-byte would then take more than 38900 seconds, or something like 200-times the worst case of our code with full detections enabled.

If better performance was desired, it is possible to separate the gathering of data (monitoring accesses to substitution tables) from the processing of the data (key and plaintext recovery): While the first phase must by necessity be performed at the time the accesses are done, the second phase does not need to – it is quite sufficient to process the data asynchronously, e.g. in a different thread or even offline from a record of the accesses in a file. That would at the very least resolve the penalty for using AES-192 which is unnecessarily incurred synchronously in the current implementation.

4.2.5.3 Limitations

From the presented tests it is obvious that the approach works in general. However, it does have certain limitations from the real-world-usage point of view:

The whole approach is based on a set of assumptions which seem to hold true in may real-world libraries, but that is certainly no guarantee that it would hold for all of them. In particular, with better compilers and more aggressive optimization techniques in them, we may well expect that loop unrolling could violate the “no mixing of rounds” requirement. Similarly, the use of true multithreading for encryption might violate the condition of blocks being processed sequentially, although in this case the use of thread identifiers could be added to the processing code to distinguish table accesses from different threads.

The key- and data recovery process is fairly slow to the point of being unusable in scenarios where a large amount of data is being processed, and it is certainly possible to write code in such a way that the slowdown might become even more pronounced. While the speed could be improved in the general case, against a targeted attack there is little to be done.

The major problem with our approach is that it is only suitable for AES implementations which use substitution tables located in the main memory. Bit-slicing implementations are completely immune to this approach, as is the usage of AES-NI instructions. Of course, if an application supports these hardware-assisted modes, it will tend to prioritize them over the S-box based approach. If we want to ensure that our monitoring technique will be successful, we must somehow convince the application that neither vector extensions (usually SSE or SSE2) nor the AES-NI extensions to the basic instruction set are available. That can sometimes be done in the computer setup (the older BIOS or the newer EFI setup) where the support for these features can be disabled, but with the advancing adoption of these features in new software that ability has been steadily disappearing over time and nowadays many systems do not provide it at all. Given the expected hardware requirements of the upcoming Microsoft Windows 11 operating system [48], we can assume that sooner rather than later it won't even be possible to start the operating system if these features are disabled.

It should be noted that even with a traditional software implementation of AES, our algorithm may run into trouble if the substitution tables do not exist in the actual executable and instead are precalculated during the program's runtime. The less time there is between the precalculation and the use of the tables, the more likely it is that some encryption may escape the detection. The applications which only perform one task and then quit may be particularly prone to this issue. Research is needed to establish what, if anything, can be done about it.

4.3 Detecting and Recovering AES by Intercepting the AES-NI Instructions

The approach described in the previous section is not suitable to applications which make use of the AES-NI instructions. At the same time, the growing availability of these instructions in new CPUs (e.g. the latest Intel CPU which does not support AES-NI is Intel Pentium A1020, launched in 2016 [33], AMD supports AES-NI in all of its architectures since Bulldozer, launched in 2011 [64]) make them increasingly interesting to applications that want to use AES for encryption. For this reason, it is imperative that a technique for recovering plaintexts and encryption keys used with these instructions is needed.

4.3.1 AES-NI Instructions

The AES-NI consists of six new instructions that were added to the instruction set [28]:

AESENC xmmA, xmmB/m128: This instruction performs one full round of the AES encryption. On input, the cipher's state is stored in one XMM register[8] and the round key is stored either in another XMM register or in a memory location. After execution, the first XMM register will contain the modified state, which becomes the input state for the next round of the cipher.

AESENCLAST xmmA, xmmB/m128: This instruction behaves similar to `AESENC`, except that it skips the MixColumns step. That makes the instruction suitable for use in the last encryption round which does not include that step. The output of the instruction becomes the final ciphertext.

AESDEC xmmA, xmmB/m128 and **AESDECENC xmmA, xmmB/m128** behave similarly to the respective `AESENC` and `AESENCLAST` instructions, except that the decryption steps are performed instead of the encryption steps.

AESKEYGENASSIST xmmA, xmmB/m128, imm8: This instruction is used during the key expansion to generate the round keys required by the cipher. It will take 128 bits of the previous round's encryption keys in the second argument and calculate 128 bits of the next round's encryption key to store into xmmA. The immediate value represents the first (non-zero) value of the *rcon* constant for that round.

AESIMC xmmA, xmmB/m128: This instruction performs the inverse MixColumns operation on an encryption key stored in the second argument and saves the result to the first argument. The instruction is intended for the generation of the decrypted key suitable for the `AESDEC` and `AESDECLAST` instructions as the instructions make use of the "Equivalent Inverse Cipher" ordering of round operations rather than the "Inverse Cipher" ordering provided by the official specification of the cipher.

4.3.2 Key and Plaintext Recovery

The specifications of the instructions allow for a very efficient key and plaintext recovery, provided that the instruction use can be detected.

4.3.2.1 Key Recovery

If we learn of the execution of `AESKEYGENASSIST`, we learn two round keys; by evaluating the third argument, we learn which round key was being generated because each round uses a different constant. Thus the key recovery algorithm can be very simple and self-sufficient: Wait for one (for a 128-bit key) or two (192-bit and 256-bit key) subsequent `AESKEYGENASSIST` operations, saving both their input values. Use the *rcon* value to determine the round key number and perform the respective number of inverse key generation steps to learn the master key. Optionally, the values of one additional instruction use can be saved in order to detect whether the same key was used for all calls. That would also allow us to determine the size of the key, by verifying how large a part of the round key can be calculated from its first 128 bits.

[8] A register used by the vector unit of the Intel architecture.

4.3.2.2 Plaintext Recovery

By capturing the execution of `AESDECLAST` instruction we directly learn the plaintext (the output of the instructions), regardless of the key size.

Recovering the plaintext during encryption is somewhat more complex and depends on the size of the key.

For a 128-bit key, it is sufficient to capture the `AESENCLAST` call and save both the final ciphertext and the round key used to generate it. By reversing the key generation process, we can then calculate the round keys for all previous rounds and perform decryption of the ciphertext. We need to know that 128-bit keys are used, though; that information can come from the keygeneration process or from some external application-specific source, e.g. the metadata of the TLS protocol.

For 192-bit or 256-bit keys, or for 128-bit keys where their size is not known, a different approach is needed: We need to capture at least two successive `AESENC` calls and verify that they are indeed successive, that is, the output of the first call becomes the input of the second call. That gives us 256 bits of round keys which we can once again use to recover the full AES key and determine its size and then use it to decrypt the plaintext.

However, in this scenario we also need to determine the round number of the round we captured. We can learn the information by monitoring the `AESKEYGENASSIST` instruction and matching the actually used keys to the values calculated from the key schedule. Another approach is to monitor all `AESENC` calls and assume that any call where the value of xmmA does not match the value of xmmA of the previous call is the beginning of the first round *after the initial AddRoundKey operation*. Yet another approach is to monitor for the sequence of `AESENC` and `AESENCLAST` where the output of the first instruction becomes the input of the second one, because then we know we have the data of the last two rounds; that does not necessarily provide us with the specific round number, but in the worst case scenario we can offer all three possibilities (9th and 10th, 11th and 12th, or 13th and 14th round) to the user to select the correct one.

Note that if an application uses AES-NI, it almost guarantees that the developer had to manually write code in assembler. If that was the case, it is to be expected that the developer performed multiple optimizations. For example, in the OpenSSL implementation of AES in CCM mode, the AES rounds for the encryption of data are intertwined with the AES rounds for calculating the MAC value, thus violating the assumption from the previous section about non-overlapping encryption operations. The recovery process needs to take that possibility into consideration.

4.3.3 Monitoring the Use of AES-NI

All of the recovery techniques discussed above, however, are predicated by our ability to detect that an AES-NI instruction was executed. Unfortunately, that is rather difficult because these instructions are not privileged in any way (to be detected by a non-permitted privilege instruction use) and can read all their data from registers (preventing any kind of

memory breakpoint, even if we could determine in advance which memory locations they would use, which is unlikely in any case).

In the paper [63], a very nice approach to this problem was suggested: The authors' technique involves the use of a CPU without the AES-NI instruction set while simultaneously tricking the application by faking the return value of the `CPUID` instruction (they achieve that by using a virtual machine hypervisor to capture the `CPUID` call because this instruction *is* privileged) to believe that AES-NI is in fact available. That convinces the application to use the AES-NI instructions which are unknown to the CPU and thus the CPU itself generates a hardware exception when these instructions are executed. The control application can capture this exception, decode the offending instruction, save all of its inputs and then emulate it in software, letting the target application continue as if nothing happened.

This approach is very simple, powerful and efficient, because it involves no modification of the target applications code (which could otherwise be detected) and the instruction detection is done by the CPU itself as a by-product of its standard functionality. As a result, the application runs full-speed most of the time and only when an AES-NI instruction is executed some processing is necessary. However, as explained above, this processing is quite simple as far as our needs are concerned, so we could adopt this approach to our tool almost without changes. In fact, we can even avoid faking the `CPUID` result value, because we have a solution for the software-only AES encryption and decryption. However, there is a significant limitation, and that's that we need a CPU that does not support AES-NI, which is becoming increasingly difficult these days, or a motherboard which can disable the functionality, which has the same problem.

In the absence of a CPU lacking AES-NI, there remain two main approaches to the problem:

Traditional *emulation* could be used to create such a CPU. Unlike virtualization, where the target application for the most part runs on the actual hardware (with some modifications done by the virtualization layer) and makes use of its features, emulation has a full control over the behavior of the CPU used by the target application. Thus it is definitely possible, especially in case of open-source emulators, to selectively remove some instructions from the emulated instruction set. The disadvantage of this approach is, of course, speed — emulation is extremely slow compared to native and even the virtualized execution.

In scenarios where performance is important, conventional *execution breakpoints* can be used. The monitoring tool would need to search the virtual memory space of the target application for the machine code of the specific instructions and then use either a hardware or a software breakpoint to stop execution when such an instruction is encountered. On the Intel Architecture, a software breakpoint is more viable in this usage because we will need to break on far too many instructions than the hardware breakpoint limits allow — e.g. a typical AES encryption will involve 9, 11 or 13 `AESENC` calls[9] followed by

[9]For performance reasons the rounds are almost always loop-enrolled if the AES-NI instructions are being used.

Instruction	Machine code
AESENC	0x66 0x0F 0x38 0xDC r/m
AESENCLAST	0x66 0x0F 0x38 0xDD r/m
AESDEC	0x66 0x0F 0x38 0xDE r/m
AESDECLAST	0x66 0x0F 0x38 0xDF r/m
AESIMC	0x66 0x0F 0x38 0xDB r/m
AESKEYGENASSIST	0x66 0x0F 0x3A 0xDF r/m imm8

Table 4.2: Machine code of the AES-NI instructions. [32]

an AESENCLAST call. Unfortunately, this approach carries all the drawbacks of software breakpoints, including an easy detection and the risk of overwriting an instruction which is a part of another instruction rather than an AES-NI instruction — although the latter is somewhat mitigated by the fact that the machine code of the AES-NI instructions is quite long and rather distinctive as shown in Tab. 4.2.

4.4 Detecting and Recovering AES by Intercepting the HW-assisted Bitslicing Algorithm

Compared to the support for AES-NI instructions, the need to automatically detect bitslicing is much reduced. It is supported by the encryption libraries (e.g. OpenSSL) and as such it can be used by applications to perform the encryption and decryption, but generally if AES-NI is available, the libraries tend to default to it because the performance is much better. Unless the user takes specific steps to prevent the use of AES-NI altogether (e.g. to alleviate concerns about the security of the instructions themselves), bitslicing will only get used on CPUs which do support vector instructions but don't support AES-NI; since all Skylake-based or newer Intel CPUs (introduced in 2015) and Jaguar-based or newer AMD CPUs (introduced in 2013) support AES-NI, and since AES-NI has been available in all but the least powerful CPUs since at least 2011, this is a dwindling concern.

However, we *can* encounter bitslicing implementations in the wild and we need to consider that they will remain in use even in the future. Could we detect their use automatically and could we extract the sensitive data (plaintexts, keys) used in them?

The answer to that question is unclear at the moment. It is definitely the case that bitslicing is much more difficult to detect than any other implementation covered in this chapter, because there is no universal detectable component in them: The traditional pure-software implementations depend on the availability of the substitution tables, which need to reside in the memory if the implementation should not be prohibitively slow. The AES-NI implementations make use of the AES-NI instructions which have a rather distinctive machine code. But bitslicing implementations do not depend on either: they don't use substitution tables (which are instead represented by logical operations) and while the instructions used are specific in that vector instructions (MMX or SSE) are used, these

instructions are also used for other purposes so they can't be summarily considered an indication of the AES use. It is only when the instructions are performed in a specific order that they comprise AES, but detecting such a scenario seems very difficult, if not impossible.

One viable option seems to be to focus specifically on popular implementations. These are relatively rare, probably due to the fact that a bitsplicing implementation requires manually writing a rather complex assembler code which is a daunting task for many developers. Thus it can be expected that most applications that use the technique will in fact contain the same or at least very similar code which may be detectable e.g. through the use of signatures. For example, the OpenSSL implementation of bitslicing AES builds around the code created by Mike Hamburg, and this code is possibly detectable through the use of constants, as seen in the sources [29]. Other implementations may be detectable through the use of signatures, much like antivirus tools detect malware. Code sequences discovered in this way could possibly be replaced with a call to a function which were injected into the target application's code by the control application, which would save the inputs and then simulate or even copy the original code. However, this approach seems to be highly specific and unreliable and we find it difficult to consider it on par with the much more generic nature of the approaches proposed previously.

It may be that a more generic scheme is available, but it's an open question whether it is even worth the effort to search for it, considering the usage aspects mentioned above. We should keep in mind that unlike a traditional attacker who gets his execution environment thrust upon herself, we are operating from a position of having a complete control over the environment. Thus, if we did encounter code which would prefer bitslicing to AES-NI, we could attempt to disable SSE instructions as well as AES-NI, thus forcing the use of a purely software-based implementation of AES.

4.5 Conclusion

In this chapter we proposed to introduce algorithms which can automatically detect the use of AES cipher and to automatically recover both the key and the plaintext. Our approach was primarily based on the observation that traditional software implementation of AES make use of precalculated substituted tables which can be detected in the application's memory, and that by evaluating the accesses to these tables we can deduce the desired information. While this approach carries a significant performance penalty and does not work against more hardware-based implementations such as Bit-slicing or the use of AES-NI, it still succeeds in a number of situations: We verified that we are able to recover key and plaintext with several commonly used encryption libraries using our own test applications, and we demonstrated that we could do the same with existing third-party applications, two of which use their own custom implementation of the AES cipher. It can be expected that other applications would be vulnerable to this approach as well, particularly so if they offload the encryption work to the libraries we tested.

With the software-based implementations thus resolved, we switched our focus to the

hardware-assisted AES implementations which are not vulnerable to our approach because they do not use a substitution table. We discussed options of achieving the same goals with them, albeit with different means, and found that AES-NI use may well be vulnerable to the use of traditional breakpoints. Certainly, once an AES-NI instruction has been identified and the code stopped at the beginning of the instruction, the process of extracting the key or the plaintext becomes relatively simple, especially compared to the rather complicated calculations of interactions between the key and the cipher state in our previous approach. Given the rather distinctive form of the AES-NI instructions' machine code, this approach seems viable.

Bitslicing approaches, on the other hand, present a much more difficult challenge. At the moment we are unable to propose a truly universal approach to their detection, much less to the universal recovery of sensitive data entering the algorithm. An approach exploiting the specific implementations was tentatively suggested, but whether it is viable in achieving the same results as the previous approaches remain to be seen.

However, it should be noted that even if were forced to limit the technique to the case of a pure-software implementation, the results are quite impressive: By the increasingly difficult but still somewhat possible expedient of disabling the AES-NI extensions of the CPU, we were able to automatically recover both the key and the plaintext from several commonly used encryption libraries as well as several applications sporting their own custom implementation of AES. That suggests that our tool can be used to do the same thing in many real-world scenarios and provide the users with a simple way of observing the encrypted traffic which would otherwise be difficult to replicate. We believe that is a worthwhile contribution.

Chapter 5

Insecure Use of AES

The underlying information used in this chapter is based on the results obtained in [2] and then revised, expanded and modified for this thesis. A publication has not yet been submitted internationally, pending fixes to the cryptography use by the developer of the application.

In the previous chapter we discussed a powerful side-channel represented by the processing environment in which a cipher was being used and we saw how the attacker how can control the environment can easily extract information about the cipher. Of course, that was facilitated by the fact that the attacker was at the same time the legitimate owner of the environment as well as the user of the cryptography, although the latter was achieved without his or her specific approval. The cipher itself could not properly defend against the attack.

A very similar effect can be achieved in other scenarios, where the cipher itself might be secure, but its security may be weakened or even completely broken by the improper use of the cipher: No matter how cryptographically secure a cipher is, if the developer who uses it in their program fails to use it properly — due to not only a malicious intent, but also misunderstanding of the properties of the cipher or the environment it is used in, or even due to a simple oversight — this misuse can dramatically decrease the security of the whole. While such a situation is typically not caused by the cipher itself and indeed, many or even all ciphers can be affected by the same mistake, the cipher's properties can facilitate it by leading a non-expert developer down a dangerous path. Unfortunately, all aspects of the use of the cipher need to be evaluated carefully if the whole should be secure, but many developers lack the expertise to do so. This happens quite often in case of applications which make use of cryptography but are not, in themselves, cryptographically oriented, but we have seen examples of these issues even in applications where cryptography is the major component, not just a minor feature.

5.1 Drive Snapshot

Drive Snapshot[1] is a Windows application for creation of backups of drive volumes and their recovery. It has been developed since 2001 and provides a number of features for its users in one tiny easy-to-use package. It uses encryption to protect the data from unauthorized use. Very few details on the actual implementation of that is available publicly and what is available is not always accurate — e.g. the german version of the documentation states that the application uses AES with 256 bit keys in the CBC mode [25], the english documentation states that the application uses AES with 128 bit keys in the CBC mode [24] and our security analysis using reverse engineering revealed that contrary to both of these official sources, AES with a 256 bit key in the CTR mode is actually used. [2]

While these discrepancies tend to cast doubt on the reliability of the application, they do not by themselves damage its security. Our analysis did find a number of alarming issues, though, even though Drive Snapshot tends to use standard algorithms in *mostly* the usual way. For example, when creating with password-protected backup images, the application follows the following high-level scheme:

1. The encryption key used to encrypt the actual data (the *data key*) is generated randomly. Each file will use a new encryption key, independent of all other previously used keys and independent of the user's password. The key is ultimately used for AES in the CTR mode to encrypt the disk data.

2. This data key is stored in the header of the generated file in an encrypted form, once again using AES, this time with a key (the *header key*) derived from the user's password.

This is a scheme which is commonly used by many encryption applications, including VeraCrypt [31]. The actual implementation in Drive Snapshot uses standard algorithms including AES, SHA256, PBKDF2 and the Diffie-Hellman key exchange algorithm and it tries to follow the recommendations for their secure use. Despite all that, several programming errors are present in the application which significantly weaken the security of the encrypted data.

5.1.1 Calculation of the Data Key in Drive Snapshot

The actual process for transforming the user's password into the *data key* suitable for encryption is this:

1. Read the user-provided password into an ANSI string: *password*.

2. Randomly generate a 128-bit salt: *salt*.

3. Append salt to the user's password: *passwordAndSalt*.

[1] http://www.drivesnapshot.de/

4. Initialize a SHA256 context, message block and the number of hashing iterations: $hashIterationCount$.
5. Repeat for 500 ms or more:
   a) Append $passwordAndSalt$ to the message block.
   b) Increment $hashIterationCount$.
   c) If the message block contains at least 64 bytes (512 bits, the size of the message block for SHA256), update the context using these 64 bytes and delete them from the message block.
6. Finalize the hash calculation to get the 256-bit hash value: $passwordHash$.
7. Calculate $publicA = \texttt{0x1267}^{passwordHash} \bmod m$, where $m = 2^{19937} - 1$, a Mersenne prime commonly used as a modulus for the Mersenne Twister.
8. Randomly generate a 256-bit secret value: $secret$.
9. Calculate $publicB = \texttt{0x1267}^{secret} \bmod m$.
10. Calculate the shared key as $sharedKey = publicA^{publicB} \bmod m$.
11. Calculate the header key by only keeping the lowest 256 bits of the $sharedKey$: $headerKey = sharedKey \bmod 2^{256}$.
12. Randomly generate a 256-bit data key: $dataKey$.
13. Initialize the $encryptedDataKey$ to contain a copy of the $dataKey$.
14. Initialize the number of encrypting iterations: $encryptIterationCount$.
15. Repeat for 100 ms or more:
    a) Encrypt the first block of $encryptedDataKey$ using AES in an ECB mode with the $headerKey$ as the encryption key and write the result back into the first block of $encryptedDataKey$.
    b) Increment $encryptIterationCount$.
16. Calculate the password verification value $passwordVerify$ as the first 20 bits of $dataKey$.
17. Use $dataKey$ to encrypt the drive data. Save $salt$, $hashIterationCount$, $publicA$, $publicB$, $passwordVerify$, $passwordIterationCount$ and $encryptedDataKey$ to the data file to facilitate future decryption to a user who knows the $password$.

The decryption process then can use the stored values and the user-supplied password to also reach the data key:

1. Read the user-provided password into an ANSI string: *password*.
2. Read *salt*, *hashIterationCount*, *publicA*, *publicB*, $passwordVerify$, *passwordIteration-Count* and *encryptedDataKey* from the data file.
3. Calculate *passwordHash* as per steps 3–6 above, except that *salt* and *hashIteration-Count* are not generated resp. calculated, but used directly as read from the file.
4. Calculate *publicA* as per step 7 above.
5. Verify that the calculated *publicA* matches the stored *publicA*. If a mismatch is detected, the password is incorrect and the process stops.
6. Calculate the shared key as $sharedKey = publicB^{passwordHash} \bmod m$. *publicB* was read from the data file, the calculation is the same as in the Diffie-Hellman key exchange.
7. Calculate the header key as per step 11 above.
8. Initialize the *dataKey* to contain a copy of the *encryptedDataKey*.
9. Repeat *passwordIterationCount*-times: Decrypt the first block of *dataKey* using AES in an ECB mode with the *headerKey* as the decryption key and write the result back into the first block of *dataKey*.
10. Verify that the first 20 bits of *dataKey* match $passwordVerify$. If a mismatch is detected, the password is incorrect and the process stops.
11. Use *dataKey* to decrypt the data.

The whole encryption process may seem overly complicated, but it is actually quite straightforward: At its core we can find an almost regular key derivation algorithm based on a repeated hashing [66] (steps 1–6) and an incorrectly implemented indirect encryption using a randomly generated key rather than a key directly provided by the user (steps 12–17). The one highly unusual part of the process is shown in steps 7–11; it represents a twist to the regular key derivation process which allows the user to optionally use public key encryption (based on the Diffie-Hellman key exchange) without having to completely redesign the whole application.

5.1.2 Security Considerations

The key generation process unfortunately contains a number of deviations from the standard approaches as well as indubitable programming errors. Some of these directly reduce the security of the stored data, others may become vulnerabilities in the future if specific problems are found in the underlying mathematical fundamentals. We would like to stress that while these issues are not inherent in the selected encryption algorithms themselves

and would be present even if other algorithms were chosen, and in this sense the algorithms are not at fault, still there are aspects that an average software developer cannot rightly be expected to understand and we must consider the possibility of similar mistakes being made in other applications.

5.1.2.1 Reading the Password

The first programming error is encountered at the very beginning of the encryption process, while reading the password from the user. Depending on the user interface chosen, Drive Snapshot uses either the `GetDlgItemTextA` WinApi function (if the application is used interactively in the GUI mode) or the `GetCommandLineA` WinApi function (if the application is used from a script). In either case, the output of these functions is an ANSI string which can then be used as usual in a C-style code.

What the developer did not realize is the fact that in both cases, the data can actually be entered in Unicode. In such a case, the Windows API itself will perform a conversion from Unicode to ANSI using the `WideCharToMultiByte` function with `CP_ACP` as the destination codepage and *without* the `WC_ERR_INVALID_CHARS` flag. As a result, any Unicode characters which cannot be represented in the OS's default ANSI code page will be replaced with question marks (characters with code `0x3F`) without the application realizing that any such conversion occurred. This can drastically reduce the complexity of the password if there's a mismatch between the preferred language of the user and the default codepage of the Windows operating system he or she is using. For example, should a Russian or Japanese speaker use English-language Windows (codepage 1252), then they will be able to enter passwords in their native language but none of the regular character of that language will have a representation in the system's code page and will be silently converted to question marks before the key derivation even starts! Note that some language combinations are more affected than the others — a Russian user on English Windows will encounter this issue but an English user on Russian Windows won't because all "regular" English characters are present even in the Russian codepage; a Czech user (code page 1250) on English Windows will encounter the problem in a reduced form because most of Czech characters (26 out of 42) are present in the OS's default codepage, but some characters will get converted. Etc.

Unfortunately, this type of error is difficult to detect and prevent without a prior knowledge of the fact that it can occur, because there is no indication of the error: Even if all characters of the password were indeed replaced, the operating system would not report any error (because no error actually occurred — the request was to convert the characters to the ANSI code page and replace any non-existent characters with a placeholder character). Once the replacement was done, the password would become a regular, although easily guessable, password and it would work properly in all tests — it would consistently generate a random-looking encryption key which would be able to encrypt and then decrypt any possible data.

The error can be easily fixed by switching to Unicode version of the functions (in this case, `GetDlgItemTextW` and `GetCommandLineW`) and if necessary, convert them to UTF8 by

explicitly using the `WideCharToMultiByte` function with the `CP_UTF8` codepage. That will generate a string which is compatible with ANSI functions but won't lose any information compared to the original input. Alternatively, a conversion to the regular `CP_ACP` codepage could be done, but with the `WC_ERR_INVALID_CHARS` flag enabled — then any conversion errors would get reported and could be acted upon. Unfortunately, both of these fixes can be difficult to use for developers who are not familiar with Windows API and resort to some high-level wrapper (such as Embarcadero's VCL), particularly if they are using a development environment which is not Unicode-enabled. All the more reason to switch to a more modern environment!

Note: While this problem is not related to the encryption algorithm itself, it demonstrates an important aspect: Due to a programming error, an encryption algorithm may very well receive significantly weakened inputs, especially the keys, that will greatly facilitate cryptographic attacks. Curiously enough, and contrary to what we would expect usually, this is a type of error which can easily affect cryptologists — perhaps even more easily than developers without a cryptology background, because the latter might be familiar with this unexpected behavior of the operating system. While the developers would do well to approach cryptology issues with humility, the same can be said about non-developers approaching developer problems.

Note: In the specific case of Drive Snapshot, this issue is further exacerbated by the incorrect logging of the command lines used to generate a data file: While the password[2] will be masked by asterisks, the length of the password would be visible to anyone who has access to the encrypted data file; if many or even all of the actually intended characters were silently replaced by question marks, that makes the attempt to break the password all that much easier.

5.1.2.2 Generating the Password Hash

After the password had been read into a local variable, Drive Snapshot uses it to calculate a hash value of it in steps 2–6. This process is called key derivation and is commonly used by applications which need to use encryption based on the user's input: A cipher uses a *key*, but the user provides a *password* or *passphrase*, which is a very different thing: The key is usually fixed length and should be of a maximum entropy, while passwords are generally variable lengths and with a much lower entropy per byte due to their use of natural languages and limited character sets. Key derivation is a technique which tries to convert a password into a key in such a way that the derived key receives all the entropy stored in the password and the process itself makes brute-force guessing of the correct password difficult.

A commonly used method for key derivation involves a repeated hashing of the password, such as in the PBKDF1 and PBKDF2 functions defined by PKCS #5 [35]. In addition to the password and the pseudorandom function, a random salt is used to prevent pre-calculation of derived keys and an iteration count is used to increase the time neces-

[2] Well — a part of it.

sary for the calculation of one derived key, which increases the cost of brute-force guessing. Newer key derivation functions, such as scrypt [51], introduce additional parameters to combat parallel calculations, e.g. by requiring a large amount of memory to operate.

Drive Snapshot uses a key derivation function similar to PBKDF2, but not the same. Let's consider the differences.

PBKDF2 is defined as follows: Given password P, salt S, iteration count c, key length d and a pseudorandom function PRF with output length h, calculate $n = \lceil d/h \rceil$ intermediate values $T_1, T_2, ..., T_n$ as $T_i = \mathrm{F}(P, S, c, i) = U_1 \oplus U_2 \oplus ... \oplus U_c$ with $U_1 = \mathrm{PRF}(P, S \parallel i)$ and $U_i = \mathrm{PRF}(P, U_{i-1})$. The derived key K is then the first d bits of $T_1 \parallel T_2 \parallel ... \parallel T_n$. [35]

Drive Snapshot's function, shown in steps 2–6 above, differs in the following aspects:

- The desired key length is fixed to $d = 256$ bits.
- The pseudorandom function used is fixed to SHA256 with $h = 256$ bits.
- Iteration count c is not a fixed value, but is dynamically adjusted to achieve a predetermined minimum execution time.
- PBKDF2 would, in this setup, call SHA256 c times on a concatenation of password and either the salt or the result of the previous call. Drive Snapshot only calls SHA256 once, on a much longer input $P \parallel S \parallel P \parallel S \parallel ... \parallel P \parallel S$, where the input is generated dynamically in one smaller buffer.

The first two differences simply specify precisely the settings of the key derivation function as a whole and do not influence its security, as long as SHA256 remains unbroken.

The dynamic selection of the iteration count is at the same time a reasonable choice and a possible security issue: It is a reasonable choice in that it allows for a dynamic adaptation to the growing performance of the CPUs without having to ask the user to provide their choice of a security margin. It is a potential security issue in the following aspects:

- There is no option for the user to increase the iteration time beyond 500 ms, which prevents the user from strengthening their derived key at the cost of reduced convenience. While it means that the user will never wait for more than approx. 500 ms before the decryption can start, it also means that the attacker will need only the same 500 ms for each password test, given the same resources as the user. But that is not generally the case: we can expect the attacker to have more resources at their disposal. That will tend to linearly reduce the required time-per-password, quite possibly below acceptable levels given the tendency of users to select non-random passwords (e.g. the rockyou.txt password list contains currently less than 13.5 million passwords, which would require 78 days to test them all at the regular speed; if the attacker had 10 times more resources than the user, brute-forcing through the list would take less than 8 days). A more cautious user might well want to increase their security by increasing the minimum time — after all, they will only be entering

the password once and the following backup process will likely take minutes, given the current disk sizes, so increasing the hashing time to 5 seconds presents only a minor inconvenience to them.

- There is no explicit lower bound on the number of iterations. In extreme cases, e.g. when pausing Drive Snapshot in a debugger in the middle of the generation, it is possible to reduce the number of iterations to 1. The brute-force testing of the passwords would then require just one SHA256 call; given that modern hardware is capable of calculating more than 4.6 billion SHA256 hashes per second [41], that would lead to checking the whole rockyou.txt password list in less than 3 milliseconds! Arguably, it is unlikely that this situation could occur without such a level of the attacker's control over the user's machine that they could just save all entered passwords in the file. Still, the attacker *can* reduce the iteration count with much smaller level of control, simply by making computation-intensive request on the user's machine at the time the Drive Snapshot's key derivation is executed. A simple expedient of having an explicit lower bound, e.g. one calculated for the least powerful supported CPU, would prevent this kind of attack.

The reduction of the number of SHA256 calls from c to 1 is also potentially dangerous, at least in these aspects:

- The decreased number of calls will decrease the effect of the setup costs of the hash function. While that doesn't seem to be an issue for SHA256, it might become an issue with other pseudorandom functions, such as those built around block ciphers, e.g. bcrypt.

- Similarly, the one-call key derivation function is prone to better optimization of the input data. Drive Snapshot's implementation, for example, is optimized for memory consumption rather than speed: It perform a context update the moment the hash function's block has been filled, then clears the block and starts filling it again. An attacker could prepare all input data in one go, then hash it all in one call, to save on function setup costs. They could, also, interleave input generation and context updates, to maximize the throughput.

- The one-call key derivation function may be more prone to shortcut attacks than multiple calls would, especially considering the regularity in the hash function's input composition (i.e. the repeated concatenation of password and salt, until the desired length had been reached). While we are not aware of any such attacks at the moment, there is no guarantee that they won't be found in the future. Such an attack on the multiple-call KDF seems much less likely due to the fact that the input differs in each iteration, being dependent on the results of all prior calls.

- Possibly the largest cause for concern is simply the fact that the Drive Snapshot's function deviates from what is usual. A large amount of resources was invested into

analyzing the security of standard key derivation functions and while that does not rule out that vulnerabilities would be found in the future, it is much less likely than in case of less-evaluated functions. In cryptography, one is always urged to use tried-and-tested approaches than applying their own ad-hoc modifications. — That said, the Drive Snapshot approach is definitely not unique, a very similar key derivation function is used e.g. in BestCrypt Volume Encryption version 3 [30].

5.1.2.3 The Diffie-Hellman Key Exchange

In steps 7–11 of the calculation of the data key, we can see what's clearly a Diffie-Hellman key exchange scheme [21] over a group with exponentiation modulo prime, with a prime $p = 2^{19937} - 1$ and a base $g = \texttt{0x1267}$. The purpose of this sequence may seem quite unclear and even suspicious, because usual key derivation schemes do not use anything like that — so why does Drive Snapshots?

The answer to that question lies in the changelog to version 1.47, which introduces a new feature: "Instead of a plain text password you can use a public encryption key now." [23]. These steps represent an additional code, not present in previous versions, which implements the public key encryption. The apparent idea behind the code is: We have a 256-bit value which we use as a key to decrypt the stored $dataKey$ from our data file. Let's not use it directly, but instead consider it a part of the private key component of the Diffie-Hellman key exchange. Then we can save the public key parts without an encryption and use them along with the private key, which will be provided as a password by the user, to generate a shared key which can then be used for the decryption of $dataKey$.

And that's exactly what the code does. For a Diffie-Hellman key exchange scheme over a group with exponentiation modulo prime, we need public parameters p, a prime number, and g, a generator of a large subgroup. Private parameters a and b are independently and randomly chosen by the two sides of the exchange. Then $A = g^a \bmod p$ and $B = g^b \bmod p$ can be exchanged over an insecure channel and both sides will be able to easily calculate a shared key $K = A^b \bmod p = (g^a)^b \bmod p = g^{ab} \bmod p = (g^b)^a \bmod p = B^a \bmod p$ while an eavesdropper who does not know either a or b must resort to solving either a Diffie-Hellman problem or a discrete logarithm problem, both of which are considered hard.

In this specific instance, we have $p = 2^{19937} - 1$, $g = \texttt{0x1267}$, $a = passwordHash$ and $b = secret$. If the public key encryption feature of the application is not used, both a and b are generated on the fly from the provided data and used to calculate the shared key K, whose lowest 256 bits then form the actual $headerKey$. The Diffie-Hellman key exchange represents an additional calculation effort.

If the public key encryption feature of the application is used, the user is required to first create a public key file, which contains the following information:

- The $salt$ and $hashIterationCount$ value which are used to calculate $passwordHash$ from the user's password.
- $publicA = \texttt{0x1267}^{passwordHash} \bmod p$

- SHA256 hash of the items above, used to verify the integrity of the file.

Then the process for generating the header key can be simplified by skipping steps 1–7 completely; the required values (*publicA* needed in step 10 and *salt* and *hashIterationCount* needed in step 17) are read directly from the public key file. In particular, the user's password is not used in this scenario and it is expected that even if the attacker got hold of the public key file, they would need to calculate the discrete logarithm of *publicA* to be able to decrypt the data file — and even then they wouldn't be able to recover the password itself, only its hash.

Unfortunately, the whole process is made insecure by the choice of parameters p and g. That choice was never explained and we can only guess at the motivation of the developer for his choice. It is likely that p was selected because it is a large prime known to the developer because of its use in Mersenne Twister pseudorandom number generators, commonly available in many programming languages. The choice of $g = \texttt{0x1267}$ is completely unclear; we were not able to find any specific use of this number in connection with the Diffie-Hellman algorithm and it doesn't seem to have any properties which would appeal to a developer (e.g. it's not prime, its binary representation is not "nice", it's not a simple mistake of entering a decimal number as hexadecimal because 1267_{10} is also neither prime nor "nice"). It is co-prime relative to $p - 1$, but that seems to be its only distinguishing characteristics.

The Diffie-Hellman key exchange in general, and the security of the Drive Snapshot's data file in particular, is only secure if the attacker cannot calculate a discrete logarithm of some of the public elements. For example, if the attacker got hold of the Drive Snapshot-generated data file and was able to calculate discrete logarithms quickly, they could read *publicA* and *publicB*, calculate a discrete logarithm of either (getting either *passwordHash* or *secret*) and then simply calculate the *sharedKey* by performing a modular exponentiation of the other value. Once the attacker has *sharedKey*, she can simply extract the lowest 256 bits of it to get a *headerKey* which then directly decrypts *dataKey* from *encryptedDataKey*.

The security issue lies with the use of a Mersenne prime in the context of the Diffie-Hellman key exchange: A Special Number Field Sieve (SNFS) is particularly effective for integers in the form of $r^e \pm s$ where r and s are both small. Mersenne primes $(2^e - 1)$ satisfy that requirement and are particularly vulnerable to this technique. [53]

At the moment, $p = 2^{19937} - 1$ might be large enough to generate a group which is computationally infeasible to break in this fashion, but it is definitely much more susceptible to an attack than the established groups, such as those provided in [38] or [39]. A developer should always use these well established groups to prevent an inadvertent use of a weak group such as the one used in Drive Snapshot. Here, the strength of the underlying AES algorithm is put at risk simply by making the encryption key more accessible to the attacker than should normally be the case.

5.1.2.4 Generating the Data Key

In step 12, Drive Snapshot generates a random 256 bit value to be used as an AES key for the encryption of the data. That is generally considered as long as a cryptographically secure random number generator is used, because the key generated in this fashion is unpredictable and contains the maximum entropy available in the system. In particular, it is important that the key is disconnected from the user's password, because even a poorly chosen password won't decrease the entropy of this key.

Upon verification, it can be seen that Drive Snapshot uses the `ADVAPI32.SystemFunction036` function as the random generator. That is in fact the implementation of the Windows internal `RtlGenRandom` function, which should not be used directly — instead, a developer should use the `CryptGenRandom` or `BCryptGenRandom` function. However, the `RtlGenRandom` is available on all versions of Windows supported by Drive Snapshot without additional dependencies (unlike `CryptGenRandom`, which actually calls `RtlGenRandom` to generate its output) and is declared as cryptographically secure by Microsoft [47], which makes it suitable for applications that try to avoid unnecessary code.

Unfortunately, Drive Snapshot does not end here. If the `RtlGenRandom` function should not be available (which is extremely unlikely, but may possibly occur in extremely low memory conditions), and alternative PRNG is used, and this one is not cryptographically secure: the result of `time64`, or the number of seconds elapsed since midnight, January 1st, 1970, is used as a seed for the standard C `rand` generator, which is then used to generate the random data. The return value of the Drive Snapshot's PRNG function contains the information about which generator was used, but this value is not checked anywhere in the program — in effect, the generated data will be used in the same way regardless of whether their source is or is not cryptographically secure.

While the risk of that happening is extremely low in current versions of Windows, it might conceivably occur in a future version if Microsoft decides to remove the `RtlGenRandom` function completely. If that happened, the user would not receive any indication that from that moment on his keys are not generated securely; in fact, any encrypted files created in such a version of Windows would be very easy to break due to the entropy of the key being reduced to 32 bits — or less, if the attacker was able to estimate the approximate time of the creation of the file. For example, if the attacker could guess the moment with a precision of plus or minus a year (extremely probable), then the entropy of the AES key would be less than 26 bits, easy enough to brute force on consumer hardware.

This problem is, unfortunately, quite common. Many developers provide a fallback solution for their random generator as a standard practice, one that is in fact recommended in other areas of software development. It is, however, not recommended in key generation scenarios unless the fallback generator is also cryptographically secure. At the very least, the user should be clearly notified that a generator which is not cryptographically secure is about to be used and whether they want to continue.

5.1.2.5 Encrypting the Data Key

Steps 13–15 serve to encrypt the data key for the purpose of storing it within the data file securely. The underlying idea is that without the *password*, it's not possible to derive the $headerKey$, and without the $headerKey$ it's not possible to decrypt the $encryptedDataKey$ because AES is used for the encryption. This is a standard practice used by many popular encryption applications, including VeraCrypt, BestCrypt, Keepass, etc. Unfortunately, in case of Drive Snapshot the developer's oversight seriously harms the actual security.

The problem lies in the actual encryption in step 15a, where the encryption is performed in-place on the first block of the $dataKey$. Unfortunately, there's a size mismatch: $dataKey$ is 256 bits long, but the AES block size is only 128 bits. Because only one block is being encrypted, it means that out of the 256 bits of the $dataKey$, the first 128 bits get encrypted and the second 128 bits are left unchanged. In effect, only one half of $encryptedDataKey$ is actually encrypted, the second half contains an unencrypted data of the data key!

How could the developer make such a blunder? Unfortunately, all too easily, particularly in ciphers such as AES where the block size is independent of the key size but it is a common enough scenario that they are in fact equal. In Drive Snapshot, for example, the application's documentation on the use of encryption suggests that it originally only used 128 bit keys, same size as the block size of AES. At that time, it made sense to simply use the one-block encryption function `EncryptBlock(InData, OutData, ExpandedKey)` without further considerations, because after all the size of the data block (i.e. the data key to be encrypted) matched the size of the data to be encrypted. Unfortunately, this approach created a nasty trap for the future, because it created an interdependency between the block size and the data size which was not explicitly expressed anywhere in the source code. Later, when the key size was increased beyond the original value, there was nothing to warn the developer that they should update their code. If the original code incorporated the size of the data in some way, e.g. by calling a function that required the size as another argument or even by using conditional compilation to verify that the size of the data matches the expected value, the problem would not occur because either more data would get encrypted automatically or a compilation error would be emitted and the developer would know that something wrong is going on.

Unfortunately, none of these precautions were taken and as a result, when the developer decided to increase the data security by switching from 128 bit keys to 256 bit keys, the security remained unchanged because the new 128 bits were provided to the attacker in plain unencrypted form.

It should be noted that this interdependency between a cipher's block size and the application's data size is not unique to Drive Snapshot. We have yet to encounter another application where it would cause revelation of the encryption keys, but we have encountered applications where the size of the data was specifically tailored to the size of the cipher's block. One example, also using encryption with AES, is the Yubikey's one-time-password algorithm, where all the data must necessarily fit into one AES block [67]. Another example is the Czech republic's Electronic Registration of Sales[3] which stores the

[3] https://www.etrzby.cz

critical information about a sale into one block of RSA. There are likely others — the lure of simplicity of implementation, of not having to deal with encryption modes, is a strong one. Unfortunately, as Drive Snapshot demonstrates, this simplicity can return at a later time with a devastating force.

5.1.2.6 Verifying the Password

As the developer states in their description of the cryptography use, they had to solve the problem of determining whether the user provided a correct password or an incorrect one so that the user could be informed if he or she mistyped. At the same time, the verification value must be such that it would not help the attacker in recovering the password. [24].

The way Drive Snapshot solves this problem is, according to the webpage, to calculate a hash of the password, store 20 bits of it and then verify that the 20 bits match the hash of the password provided by the user [24]. That would indeed be a viable solution, although with cryptographically secure hash functions there wouldn't really be any need to limiting the saved hash value to 20 bits.

Unfortunately, this description is not accurate. As can be seen in step 16 of the data key calculation process, the verification value is *not* a part of a hash of the password but simply *the first 20 bits of the actual data key*, which is then saved in plain text in the created data file. That effectively means the reduction of entropy of the key by another 20 bits because the value is known to the attacker.

It is unclear why did the developer choose this solution. There are tried and tested methods for solving the issue without providing the attacker with useful information: The data key, or better yet the header key, could have been hashed and that hash saved. Some fixed plaintext could have been encrypted with either of the keys and the ciphertext saved, giving an option to decrypt it and verify the value. In fact, given that *publicA*, which is directly dependent on the user's password, is saved into the data file, the verification could simply involve a re-calculation of this value and verification that it matches. All of these techniques would be secure provided that the underlying cryptographic mechanisms were secure. It seems the developer fixed their mind on the idea of *protecting the password* so much that they failed to consider that *protecting the key* is just as necessary.

5.1.2.7 Conclusion

Overall, the processing of the cryptographic information in Drive Snapshot reveals a number of serious vulnerabilities: The entropy of the password may be much lower than the user expects, simplifying the brute force recovery of it. That is further facilitated by another discovery of [2] that when executing the application from command line, the length of the password and possibly even a part of the password itself is stored in a readable form in the data file as a part of its log.

Focusing on the encryption keys, which are more relevant to this thesis, we find even more vulnerabilities. The key derivation function used may not consume as much time as intended, leading to the possibility of a brute force attack. Unsuitable group used in

conjunction with the Diffie-Hellman key exchange may cause the recovery of the key by using a particularly efficient method for solving the discrete logarithm problem. And in addition to all that, 148 bits of the key, or 57,8 %, is stored in plaintext form in the data file.

While it could be argued that the remaining 108 encrypted bits of the key are sufficient to protect the data against regular (non-quantum) attacks, the decrease in security is significant. It also illustrates a very important point, that while a cryptographic algorithm may very well be considered unbreakable, it secure use depends on aspects which can introduce devastating vulnerabilities. Let's not forget that the best current cryptanalysis of AES can shave less than 2 bits from the attack complexity while here we saved 148 bits even without considering other avenues of attack, with just the generated data file!

Chapter 6

Conclusions

6.1 Summary

In this thesis, we were studying selected security aspects of AES and its use.

In Chapter 2, we analyze the components of the Rijndael cipher and discussed the design principles and decisions involved in its creation. We show that while AES is a strictly defined special case of Rijndael, Rijndael itself allows for a large number of adaptations that, assuming that the design principles and requirements are followed, should all share the same security properties. That allows us to generate other variants of Rijndael an analyze them, with the purpose of addressing specific security aspects which may not be directly approachable in the regular Rijndael cipher but may be more accessible in a specific adaptation. In this way we can focus on a specific component of the cipher and analyze it more thoroughly than would normally be the case.

In Chapter 3, we present Baby Rijndael, a reduced model of Rijndael. We show that indeed Baby Rijndael is built on the same principles as Rijndael and that, excluding changes necessary for the reduction of scale, the design choices followed the same reasoning in both ciphers. With this background done, we perform exhaustive linear cryptanalysis of the reduced cipher, verifying all combinations of plaintext, key and linear approximations. Our results show that there is a great number of linear approximations of Baby Rijndael, all with the same probability bias, but that some groups of approximations are much more successfull in finding the correct key than others. We show that even within these groups variations between the success rate of approximations result and that some bits of the key are easier to recover than other, i.e. the "best" bit can be recovered with probability 70.3 % while the worst with probability 49.1 %. While this seems a promising result and certainly shows that the strength of linear cryptanalysis might be situation-dependent in that some plaintexts or keys could be easier to recover than others, we weren't able to find a specific rule as to which plaintexts, which keys, or which bits of them these vulnerable bits might be. The current results tend to indicate that while a certain variation in success rate is there, it's not exploitable — and definitely not so if we consider the computational effort involved when compared to simply trying all keys one by one. Which is, after all, what we

would like to see in the cipher — that there is no better way to break it than by trying all keys, because we can easily design a cipher to use such large keys that this brute-force test is not possible for them. In this sense, our results provide us with a peace of mind — if it isn't effective to linear cryptanalysis to break Baby Rijndael, it's almost certain that linear cryptanalysis of AES itself will not be effective either.

In Chapter 4, we introduce the different methods for implementing AES on Intel Architecture CPUs, commonly used in modern personal computers. We discuss the properties of software-based naïve implementation as well as optimized implementation using T-tables and propose an algorithm for using the properties of Rijndael, and specifically the fact that it uses a fixed S-box, to automatically detect that the cipher is being used an even to automatically recover both the encryption key and the plaintext. By performing tests on several popular cryptographic libraries as well as independent implementations of the cipher we demonstrate that the algorithm is effective in its purpose. We also discuss the possibility of performing the same tasks against hardware-assisted techniques using either the vector unit and bit-slicing or the specialized AES-NI instructions. While the former case seems difficult if not impossible to detect completely automatically and may require manual adjustment for each type of implementations, AES-NI use can be detected and attacked automatically, under certain conditions.

In Chapter 5, we analyze the key derivation process of Drive Snapshot, a tool for backing up user's data and encrypting it using AES. By using reverse engineering, we were able to understand the process in sufficient detail to perform an analysis of its security. It turns out that while Drive Snapshot uses AES with 256 bit keys to encrypt the user data and thus should be secure as long as AES remains unbroken, programming errors on the side of developer significantly reduce the security: Disregarding all other errors, the application incorrectly saves the encrypted data key in such a fashion that a large part of it is actually stored in the data file in an unencrypted form — specifically, out of the 256 bits, only 108 bits are encrypted; 20 bits are intentionally stored in plaintext to be used as a verifier that the password is correct, and another 128 bits are mistakenly skipped while encrypting. In addition to that, the entropy stored in the user's password may be significantly lowered under certain conditions because of conversion issues; in extreme cases, no entropy may remain at all. The key derivation function may potentially be subverted to reduce its security by reducing the number of iterations performed and thus allowing to test more passwords per second than would normally be the case. The use of the Diffie-Hellman key exchange, and specifically the use of an unsuitable group, can allow key recovery through solving a discrete logarithm problem by a particularly efficient algorithm. All of this shows that even with a cipher which is considered secure, a simple programming error can short-circuit that security and allow access to the data; unfortunately, many of the observed issues are of such a nature that they can be expected to exist in many other applications as well.

6.2 Contributions of the Dissertation Thesis

We consider these aspects the main contributions of the thesis:

- Analysis of Baby Rijndael reveals that it is an AES model designed along the same lines as AES itself, which makes it suitable to various analyses which would be difficult or even impossible with full AES. In fact, various cryptanalytic techniques have already been attempted, including algebraic cryptanalysis using Gröbner bases, differential cryptanalysis, cryptanalysis using impossible differentials, etc., with varying results.
- Analysis of Baby Rijndael shows that there are heretofore unknown properties of linear cryptanalysis, such as the varying performance of differing linear approximations used. This leads us to believe that there is more to discover in this area, with possible implications on other ciphers.
- Despite these discoveries, the practical verification shows that linear cryptanalysis is not sufficiently strong to break Baby Rijndael with a computational complexity lower than brute force, even in the most advantageous contexts. That result tends to scale to AES itself and assure us of its resistance to linear cryptanalysis.
- We show that side channels created by implementations of AES can be powerful enough to make attacks feasible. We present an algorithm which enable the attacker who can control the execution flow of an application to automatically detect the use of AES and recover both the encryption key and the plaintext, as long as a software-based AES implementation is used. We also show that some hardware-assisted implementations can also be attacked under the right circumstances.
- We performed a security analysis of Drive Snapshot, particularly its key generation process. We discovered a number of vulnerabilities which were reported to the developer for fixing. We also analyzed the causes of these vulnerabilities and proposed steps to be taken by developers to avoid these vulnerabilities in future applications.

6.3 Future Work

We would like to focus our future research particularly in these directions:

- Other types of cryptanalysis can be applied to Baby Rijndael, to select the most efficient ones. Once that is done, that technique can be applied to other models of AES, slowly increasing the size of the model and observing how does the technique scale. That should provide an insight into the behavior of the technique in AES.
- A more extensive study of the bit-slicing implementations of AES needs to be performed, with the intent to locate common grounds of these implementations that

would allow their automatic detection, much like with the software and the AES-NI implementation.

- Finally, the errors discovered in Drive Snapshot show that even a long-standing application can contain grievous errors that can allow an attacker to circumvent an otherwise secure encryption algorithm itself. We recommend continuing in these analyses, particularly for applications where encryption is not the main focus, as it can be expected that their developers are less aware of security issues inherent in the use of cryptography. We need to fix these applications!

Bibliography

[1] Auxier, B.; Rainie, L.; et al. *Americans and Privacy: Concerned, Confused and Feeling Lack of Control Over Their Personal Information*. [online], 2019. Available from: `https://www.pewresearch.org/internet/2019/11/15/americans-and-privacy-concerned-confused-and-feeling-lack-of-control-over-their-personal-information/`

[2] Bambuch, M. *Security Analysis of Drive Snapshot*. 2021.

[3] Bard, G. Algebraic Cryptanalysis. Springer US, first edition, 2009, ISBN 978-0-387-88756-2.

[4] Bellare, M.; Kohno, T.; et al. *The Secure Shell (SSH) Transport Layer Encryption Modes*. [online], 2006. Available from: `https://datatracker.ietf.org/doc/html/rfc4344`

[5] Bergman, C. *A Description of Baby Rijndael*. Technical report, Iowa State University, 2005. Available from: `http://www.math.iastate.edu/cbergman/crypto/homework/babyr/babyr.pdf`

[6] Bernstein, D. J.; Lange, T.; et al. *Dual EC: A standardized back door*. In The New Codebreakers, Springer, 2016, pp. 256–281.

[7] Biham, E. *Impossible cryptanalysis of Skipjack*. In CRYPTO '98, 1998.

[8] Biham, E.; Biryukov, A.; et al. *Miss in the Middle Attacks on IDEA and Khufu*. In International Workshop on Fast Software Encryption, Springer, 1999, pp. 124–138.

[9] Biham, E.; Shamir, A. *Differential cryptanalysis of DES-like cryptosystems*. Journal of CRYPTOLOGY, volume 4, no. 1, 1991: pp. 3–72.

[10] Biham, E.; Shamir, A. Differential Cryptanalysis of the Data Encryption Standard. New York: Springer-Verlag New York, first edition, 1993, ISBN 978-1-4613-9316-0.

[11] Biryukov, A.; Dunkelman, O.; et al. *Key recovery attacks of practical complexity on AES-256 variants with up to 10 rounds*. In Annual International Conference on the Theory and Applications of Cryptographic Techniques, Springer, 2010, pp. 299–319.

[12] Biryukov, A.; Khovratovich, D. *Related-key cryptanalysis of the full AES-192 and AES-256*. In International conference on the theory and application of cryptology and information security, Springer, 2009, pp. 1–18.

[13] Bogdanov, A.; Khovratovich, D.; et al. *Biclique cryptanalysis of the full AES*. In International conference on the theory and application of cryptology and information security, Springer, 2011, pp. 344–371.

[14] Bogdanov, A.; Wang, M. *Zero correlation linear cryptanalysis with reduced data complexity*. In International Workshop on Fast Software Encryption, Springer, 2012, pp. 29–48.

[15] Chabaud, F.; Vaudenay, S. *Links between differential and linear cryptanalysis*. In Workshop on the Theory and Application of of Cryptographic Techniques, Springer, 1994, pp. 356–365.

[16] Cho, J. Y. *Linear cryptanalysis of reduced-round PRESENT*. In Cryptographers' Track at the RSA Conference, Springer, 2010, pp. 302–317.

[17] Cid, C.; Murphy, S.; et al. *Small scale variants of the AES*. In International Workshop on Fast Software Encryption, Springer, 2005, pp. 145–162.

[18] Daemen, J.; Rijmen, V. *AES proposal: Rijndael*. 1999.

[19] Daemen, J.; Rijmen, V. The design of Rijndael: AES – the Advanced Encryption Standard. Berlin, Heidelberg: Springer-Verlag, first edition, 2002, ISBN 978-3-540-42580-9.

[20] DeBlasio, J. *Protecting users from insecure downloads in Google Chrome*. [online], 2020. Available from: `https://blog.chromium.org/2020/02/protecting-users-from-insecure.html`

[21] Diffie, W.; Hellman, M. *New directions in cryptography*. IEEE transactions on Information Theory, volume 22, no. 6, 1976: pp. 644–654.

[22] Edgar, T. W.; Manz, D. O. Research Methods for Cyber Security. Syngress Publishing, first edition, 2017, ISBN 978-0-12-805349-2.

[23] Ehlert, T. *Drive Snapshot - What's new*. [online], 2019. Available from: `http://www.drivesnapshot.de/en/news.htm`

[24] Ehlert, T. *Snapshot - Encryption Details*. [online], 2021. Available from: `http://www.drivesnapshot.de/en/icrypt.htm`

[25] Ehlert, T. *Snapshot - Verschlüsselung Details*. [online], 2021. Available from: http://www.drivesnapshot.de/de/icrypt.htm

[26] Fischer, V.; Drutarovskỳ, M. *Two methods of Rijndael implementation in reconfigurable hardware*. In International Workshop on Cryptographic Hardware and Embedded Systems, Springer, 2001, pp. 77–92.

[27] Google Corporation. Android Developers: Cryptography. Google Corporation, 2021. Available from: https://developer.android.com/guide/topics/security/cryptography

[28] Gueron, S. *Intel Advanced Encryption Standard (AES) New Instructions Set*. Technical report, Intel Corporation, 2010. Available from: https://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf

[29] Hamburg, M. *Constant-time SSSE3 AES core implementation*. [online], 2009. Available from: https://github.com/openssl/openssl/blob/master/crypto/aes/asm/vpaes-x86.pl

[30] Horňák, J. *Security Analysis of BestCrypt Volume Encryption*. 2016.

[31] IDRIX. *VeraCrypt Encryption Scheme*. [online], 2019. Available from: https://veracrypt.fr/en/Encryption%20Scheme.html

[32] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual. Intel Corporation, 2021. Available from: https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4.html

[33] Intel Corporation. *Intel Product Specification Advanced Search*. [online], 2021. Available from: https://ark.intel.com/content/www/us/en/ark/search/featurefilter.html?productType=873&0_AESTech=False

[34] Junod, P.; Canteaut, A. Advanced Linear Cryptanalysis of Block and Stream Ciphers. Amsterdam: IOS Press, first edition, 2011, ISBN 978-1-60750-843-4.

[35] Kaliski, B. *PKCS #5: Password-Based Cryptography Specification Version 2.0*. [online], 2000. Available from: https://datatracker.ietf.org/doc/html/rfc2898

[36] Kaliski, B. S.; Robshaw, M. J. *Linear cryptanalysis using multiple approximations*. In Annual International Cryptology Conference, Springer, 1994, pp. 26–39.

[37] Käsper, E.; Schwabe, P. *Faster and timing-attack resistant AES-GCM*. Cryptographic Hardware and Embedded Systems – CHES 2009, 2009: pp. 1–17. Available from: https://cryptojedi.org/papers/aesbs-20090616.pdf

[38] Kivinen, T.; Kojo, M. *More Modular Exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE)*. [online], 2003. Available from: https://datatracker.ietf.org/doc/html/rfc3526

[39] Lepinski, M.; Kent, S. *Additional Diffie-Hellman Groups for Use with IETF Standards*. [online], 2008. Available from: https://datatracker.ietf.org/doc/html/rfc5114

[40] Malbrain, K. *Higher performance AES C byte-implementation*. [online], 2007. Available from: https://www.geocities.ws/malbrain/aestable2_c.html

[41] MarioD. *Nvidia 3060ti benchmarks*. [online], 2020. Available from: https://hashcat.net/forum/thread-9694.html

[42] Matsui, M. *Linear cryptanalysis method for DES cipher*. In Workshop on the Theory and Application of of Cryptographic Techniques, Springer, 1993, pp. 386–397.

[43] Matsui, M. *How Far Can We Go on the x64 Processors?* Fast Software Encryption, 2006: pp. 341–358. Available from: https://www.iacr.org/archive/fse2006/40470344/40470344.pdf

[44] Matějka, J. *Recovery of the AES key by monitoring a program's flow*. 2019.

[45] Microsoft Corporation. Debugging Functions. Microsoft Corporation, 2018. Available from: https://docs.microsoft.com/en-us/windows/win32/debug/debugging-functions

[46] Microsoft Corporation. Microsoft AES Cryptographic Provider. Microsoft Corporation, 2018. Available from: https://docs.microsoft.com/en-us/windows/win32/seccrypto/microsoft-aes-cryptographic-provider

[47] Microsoft Corporation. *Microsoft SDL Cryptographic Recommendations*. [online], 2018. Available from: https://docs.microsoft.com/en-us/security/sdl/cryptographic-recommendations

[48] Microsoft Corporation. *Windows Processor Requirements*. [online], 2021. Available from: https://docs.microsoft.com/en-us/windows-hardware/design/minimum/windows-processor-requirements

[49] Moriarty, K.; Farrell, S. *Deprecating TLS 1.0 and TLS 1.1*. [online], 2021. Available from: https://datatracker.ietf.org/doc/html/rfc8996

[50] NIST. *Announcing the ADVANCED ENCRYPTION STANDARD (AES)*. Technical report, National Institute of Standards and Technology (NIST), 2001. Available from: http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf

[51] Percival, C.; Josefsson, S. *The scrypt password-based key derivation function*. IETF Draft URL: http://tools.ietf.org/html/josefsson-scrypt-kdf-00.txt, 2016.

[52] Polyakov, A. *crypto/aes/asm/aes-586.pl*. [online], 2016. Available from: `https://github.com/openssl/openssl/blob/master/crypto/aes/asm/aes-586.pl`

[53] Pomerance, C. *A tale of two sieves*. In Notices Amer. Math. Soc, Citeseer, 1996.

[54] Rescorla, E. *The Transport Layer Security (TLS) Protocol Version 1.3*. [online], 2018. Available from: `https://datatracker.ietf.org/doc/html/rfc8446`

[55] Rijmen, V.; Bosselaers, A.; et al. *Optimised ANSI C code for the Rijndael cipher*. [online], 2000. Available from: `https://github.com/openssl/openssl/blob/master/crypto/aes/aes_x86core.c`

[56] Russell, A.; Tang, Q.; et al. *Cliptography: Clipping the power of kleptographic attacks*. In International Conference on the Theory and Application of Cryptology and Information Security, Springer, 2016, pp. 34–64.

[57] Satoh, A.; Morioka, S.; et al. *A compact Rijndael hardware architecture with S-box optimization*. In International Conference on the Theory and Application of Cryptology and Information Security, Springer, 2001, pp. 239–254.

[58] Schneier, B. Applied Cryptography. New York: John Wiley & Sons, first edition, 1996, ISBN 0-471-12845-7.

[59] Schneier, B. *The Strange Story of Dual_EC_DRBG*. [online], 2007. Available from: `https://www.schneier.com/blog/archives/2007/11/the_strange_sto.html`

[60] Shorin, V. V.; Jelezniakov, V. V.; et al. *Linear and differential cryptanalysis of Russian GOST*. Electronic Notes in Discrete Mathematics, volume 6, 2001: pp. 538–547.

[61] Solil, L. *Reduced Models of Rijndael*. 2013.

[62] Supasatit, T. *TLS 1.3: Will Your Network Monitoring Go Blind?* [online], 2018. Available from: `https://www.extrahop.com/company/blog/2018/maintain-visibility-with-tls-1.3/`

[63] Takehisa, T.; Nogawa, H.; et al. *AES Flow Interception: Key Snooping Method on Virtual Machine - Exception Handling Attack for AES-NI -*. IACR Cryptology ePrint Archive, volume 2011, 01 2011: p. 428.

[64] Torres, G. *Inside the AMD Bulldozer Architecture*. [online], 2010. Available from: `https://hardwaresecrets.com/inside-the-amd-bulldozer-architecture/`

[65] Wagner, D. *The boomerang attack*. In International Workshop on Fast Software Encryption, Springer, 1999, pp. 156–170.

[66] Yao, F. F.; Yin, Y. L. *Design and analysis of password-based key derivation functions*. In Cryptographers' Track at the RSA Conference, Springer, 2005, pp. 245–261.

[67] Yubico. *OTPs Explained*. [online], 2021. Available from: https://developers.yubico.com/OTP/OTPs_Explained.html

Reviewed Publications of the Author Relevant to the Thesis

[A.1] Kokeš, J.; Lórencz, R. Linear Cryptanalysis of Baby Rijndael. In: *The Fourth International Conference on e-Technologies and Networks for Development (ICeND2015)*. Lodz: Lodz University of Technology, 2015. pp. 28-33. ISBN 978-1-4799-8450-3.

[A.2] Kokeš, J.; Matějka, J.; Lórencz, R. Automatic Detection and Decryption of AES by Monitoring S-box Access. In: *Proceedings of the 7th International Conference on Information Systems Security and Privacy*. Madeira: SciTePress, 2021. p. 172-180. ISSN 2184-4356. ISBN 978-989-758-491-6.

[A.3] Kokeš, J.; Matějka, J.; Lórencz, R. Automatic Detection and Decryption of AES Using Dynamic Analysis. In: *SN Computer Science*. Switzerland: Springer Nature, 2021. Requested expanded article, pending review.

Remaining Publications of the Author Relevant to the Thesis

[A.4] Kokeš, J. *Block Ciphers' Resistance to Linear and Differential Cryptanalysis.* In: *Sborník příspěvkøu PAD 2014.* Liberec: TUL, Fakulta mechatroniky a mezioborových inženýrských studií, 2014. pp. 38-43. ISBN 978-80-7494-027-9.

[A.5] Kokeš, J. *Praktické aspekty lineární kryptoanalýzy blokových šifer.* In: *Sborník příspěvkøu PAD 2015.* Zlín: Universita Tomáše Bati ve Zlíně, 2015. pp. 25-30. ISBN 978-80-7454-522-1.

[A.6] Kokeš, J. *Practical Aspects of Linear Cryptanalysis of Reduced Models of the Rijndael Cipher.* Ph.D. Minimum Thesis, Faculty of Information Technology, Prague, Czech Republic, 2015.

[A.7] Kokeš, J. *Cryptanalysis of Baby Rijndael.* Masters Thesis, Faculty of Information Technology, Prague, Czech Republic, 2013.