KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

White-Box Cryptography
Analysis of White-Box AES Implementations

Yoni De Mulder

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor in Engineering

February 2014

White-Box Cryptography

Analysis of White-Box AES Implementations

Yoni DE MULDER

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor in Engineering

February 2014

D/2014/7515/22
ISBN 978-94-6018-798-8

To my parents and my girlfriend.

Acknowledgements

When it comes to life,
the critical thing is whether you
take things for granted or
take them with gratitude.

– Gilbert Keith Chesterton

From the moment I started writing this doctoral thesis, which seems already ages ago due to underestimating the writing process for which I was warned in advance, I was looking forward to write these acknowledgements[1] since I knew this was going to be one of the last items on my todo list. However, now the time finally has come, it seems much harder than first anticipated. It even seems harder than writing the PhD text itself, since it is most likely the only part of my thesis that will actually be read by 'many' people. Therefore, it appears to me the perfect opportunity to summarize my main research contributions. No no, just kidding. But seriously, if you are planning to read this thesis, I should probably already warn you that the PhD text is rather long. The reason is that I had the impossible idea to write a reference book about white-box AES implementations for all (read: all ten, which is probably already overestimated) white-box cryptographers out there in the crypto world. Since secure white-box implementations seem hard to achieve (read this thesis!), I assume that the life expectancy of white-box cryptographers is not that high.

Pursuing a PhD and working towards the ecstasy of writing this doctoral thesis have been made possible by a diverse mix of people who influenced me (both professionally and personally) over the past six years. Therefore, I start by thanking you all for your support, inspiration and guidance. However, some people deserve some special attention, but don't feel offended if your name is not mentioned below because I didn't want to make these acknowledgements even longer than my PhD text.

[1]I thank Annemie Deiteren for proofreading these acknowledgements.

First of all, I thank my supervisor Prof. Bart Preneel for giving me the opportunity to start (and finish) a PhD at one of the most fascinating research groups called COSIC, and for always finding funding to get me paid and to let me travel abroad for international conferences or workshops. I also thank him for giving me the enormous amount of freedom to perform research in my own way, and for replying with his famous "Ok, Bart." mails (my guess is that he uses a hotkey for this, like F4). I am also very grateful that he carefully proofread this thesis and provided me with numerous insightful comments. Apart from approaching Bart as my supervisor, I always enjoy his interesting and pleasant travel stories from all over the world, and his dry humor.

Next, I thank Prof. Vincent Rijmen, Prof. Frank Piessens, Dr. Peter Roelse, Prof. Lars Knudsen and Prof. Koen De Bosschere for agreeing to act as a jury member, for being (or trying to be) present at my preliminary defense at the impossible hour of 8 in the morning, for asking me clever and challenging questions during the defense, and for providing me with comments that improved the quality of this thesis. I sincerely apologize for their 'suffering' while reading through the more than 200 pages that I have produced 'accidentally'. Further, I thank Prof. Pierre Verbaeten for chairing the jury. The support of the KU Leuven for financing my research is warmly appreciated.

More or less six years ago, I started my PhD journey at COSIC. When I gradually got to know all the colleagues one by one, either by sharing offices or by occasional chats in the hallways or at social events, it soon became clear that 'colleague' is some kind of synonym for 'friend'. Therefore, I thank many (ex-)COSICs for the nice time we spent together: Filipe and Anthony for the necessary and fair competition during the karting races; Begül and Nicky for spending a sleepless night in Athens after missing our flight connection; Deniz, Kerem, Qingju, Elmar, Atul and Sebastiaan for a pleasant working environment while sharing offices; Gabriel, Elke and Andrey for all the fun we had during the ECRYPT summer schools; Jan, Nessim and Nikos for the chaotic SoPro meetings; and finally Stefaan, Dave, Roel P., Dries and Karel for being the crew of the black-shirt-red-tie events and for jointly paying for our mistake by letting Karel decide on the movies. A special thanks goes to Brecht Wyseur for introducing me into the fascinating and magical world of white-box cryptography and for guiding me during the first years of my PhD. This guidance was most efficient by email since 'West-Vlaams' was and is still not supported by Google translate in a text-to-speech manner.

I express my deep gratitude to Prof. Vincent Rijmen for always being there when I needed him. Although not on paper, I consider Vincent as my second supervisor for all his guidance and support I enjoyed over the past few years. Of course every cryptographer knows Vincent as one of the designers of the famous AES cipher deployed worldwide, but I got to know Vincent in person when I

switched internal research subgroups at COSIC. Even though as a white-box cryptographer I didn't fit nicely into any of the subgroups, Vincent warmly welcomed me at his symmetric-key group and gave me the feeling that I finally found a group to which I belong. At COSIC, the door of his office is always wide open. And when you knock on the door, he always makes time for you. Further, I thank Vincent for his great sense of humor; I always enjoyed our (technical) conversations with the necessary dose of laughter and his subtle jokes during the symmetric-key group meetings. Finally and most importantly, Vincent showed me that not only 'colleague', but also 'supervisor' is a synonym for 'friend'. Thank you for that!

The person who influenced my research the most, and moreover had a tremendous positive impact on me as a researcher, is Dr. Peter Roelse. He was my mentor during my research internship at the Irdeto company from May 2011 until February 2012 and is the well-deserved co-author of my two most important publications. I gracefully thank him for his enriching guidance both during as well as after the internship. Over the past years, I have learned a lot from him and he has been truly inspirational to me. He influenced the way I perform scientific research and improved my writing skills significantly (though I still fail to write down only the essential part). Because of Peter, I now (try to) pay more attention to details. Further, he broadened my knowledge with numerous endless but fruitful discussions about white-box cryptography and its purpose. Next, since humor is important to me, I knew immediately that I would like Peter as a person because of his unique sense of humor. I don't think we ever had a discussion during which I laughed for less than 50% of the time (?!, ok, maybe that's an exaggeration), even if it was a serious technical discussion. Finally, I thank Peter for always being honest with me and for never beating around the bush when giving me feedback. So, for all of the above, I could not have imagined a better mentor than Peter for my PhD.

COSIC would not be the place as it is today without its driving engines underneath the hood. Therefore, I personally thank Péla Noë for being the best secretary, not only in the world, but in the entire universe! She showed me that 'secretary' definitely is a synonym for 'friend'. She was always unconditionally prepared to help me. Her office was the one place in the building where you could always go for answers to your questions and for a relaxing chat (and the occasional gossip). I am also grateful to Elvira Wouters, Elsy Vermoesen and Wim Devroye for taking care of the financial aspects, and to Saartje Verheyen for organizing everything behind the scenes.

In particular, one of the ex-colleagues at COSIC deserves some special attention. Some know him as the 'Pin Master', others as Dr. K., but his true name will never be revealed! I cannot remember anymore how fast the transition went from being a colleague to being a best friend, but I think it must have

approached the speed of light. As a colleague, he took me under his wing. At some point in my PhD journey, Karel noticed that I was going through a phase of severe demotivation. Without asking, he arranged the research internship at Irdeto, where I met Peter who brought me back on track. For that, I am eternally grateful to Karel; implicitly because of him, I can now write these acknowledgements. As a best friend, he enriched my life in so many different aspects. For example, I am now a fan of Guinness as well, or how Karel calls it: "Glazen boterhammekes." Recently I read the acknowledgements of his doctoral thesis dating back to 2012, in which he thanks me, and I quote: "for letting me (= Karel) win our games of squash for more than three years now (don't worry Yoni, the age difference and sports injuries will resolve that in the end)." Well, Karel, I am still waiting!!! I also thank Karel for pointing out to me that I've spent approximately 1% of my life on writing this thesis. But that is definitely not the worst 1%. No, I've spent the worst 1% on writing a project deliverable about VPAN, supervised by Karel, who sent me back to the writing table for over half a year because the deliverable needed to have 'meer vlees'.

I further warmly thank all my friends who have supported me throughout my PhD journey, and especially at the end when I disappeared from the planet Earth for almost half a year to write this thesis. So, a big thank you goes to Jan, Heleen, Jonathan, Hanne, Dieter, Jasmien, Bart, Peter, Willemien and others who I forgot to mention here. A special thank you goes to Eva because of her motivational words: "Maakt dat f*cking doctoraat af!" Up to now, these words are still hanging above my desk at home.

Now, I've finally arrived at those people who I carry in my heart the deepest.

I start by expressing my eternal gratitude to my parents. First of all, I thank them for transferring all their moral values, wisdom and love to me what made me the person I am today. Second, I thank them for all the opportunities they gave me in life. Because of them, I could study for civil engineering, which led me to pursue a PhD, which in its turn led me to write these acknowledgements as one of the first pages of this doctoral thesis. I know they are (and always have been) proud of me, with a current peak now that I finally became a Doctor (the second in the family, though not of Medicine but in Engineering), but what they don't know is that I have been proud of them my entire life as well. They are fascinating people and they will always be my role models in life. They have the heart at the right place, they *unconditionally* believe in me, and I can count on them not only in the good times, but more importantly also in the bad times. Trust me, my PhD experience has been a crazy roller coaster ride with lots of highs and lows. During the lows, there has been always one address to which I could return for support, and that was home. *Thank you* for being such great parents, and for creating such a warm place that I know as *my* home!

I know it is not customary to do so, but I also thank my parents' dog Liesel who has more nicknames than the number of people at COSIC. During the last week of writing this thesis before submitting to the examination committee, my parents were on holiday and I was dogsitting at home. Even though the writing days were long and shifted more to the night time, she remained faithful to sit next to me so I was never alone and always had someone to talk to. I will reward the 'Schmutse' with 'meaty sticks', her favorite candy!

Further, I warmly thank the craziest sister in the world, which happens to be my sister, Iris. Even though we don't see each other so often because of busy agendas, the bond that we share is for life and she is there for me when it matters the most. I can tell by the little things. Normally I don't get many text messages, but I was very surprised with all the encouraging messages she sent me before every important deadline during the final phase of my PhD. Thank you for supporting me in your peculiar way!

I saved the best for last: my girlfriend, Marie-Laure, with whom I share all tiny aspects of life. Her love, support and patience have been (and still are) limitless. First of all, an enormous thank you for being as steady as a rock (I know it is cliché to say, but it is the truth). Marie-Laure always had a listening ear during my countless emotional glitches, and time after time she had the perfect pep talk to put a smile on my face again. I consider being able to put a smile on someone's face regardless of the situation to be an extraordinary gift. Now, since Marie-Laure has this gift, she truly is one out of 7 billion people! Second, I gracefully thank Marie-Laure for giving me all the time I needed to finish my PhD. There were many times I was busy and stressy, such as before submission deadlines or when writing this thesis, during which she took care of all household tasks but more importantly took care of me in a very thoughtful and beloved way such as bringing me a hot chocolate before she went to sleep. I can continue to give reasons why Marie-Laure is an amazing woman, but let's just say that I am very grateful that 7 years ago, she chose me to share her life with. In my eyes, *she is like a flower that blossoms into eternity.*

Yoni De Mulder
Mortsel, February 2014

Abstract

Cryptographic algorithms are designed to protect data or communication in the presence of an attacker. If these algorithms make use of a secret key, then their security relies on the secrecy of the key. Hence, the primary objective of an attacker typically is to extract the key. In a traditional black-box environment, the attacker has only access to the inputs and outputs of a cryptographic algorithm. However, due to the increasing demand to deploy strong cryptographic algorithms within software applications that are executed on untrusted open platforms owned and controlled by a possibly malicious party, the black-box environment becomes inadequate. Therefore, a new realistic white-box environment is introduced in which an attacker has complete access to a software implementation of a cryptographic algorithm and furthermore has full control over its execution environment. Real-world examples of a white-box environment can be found in digital content protection systems such as Digital Rights Management or Pay-TV systems, where key-instantiated cryptographic algorithms are implemented on e.g. a smartphone, tablet or set-top box. The extraction of the secret key would compromise the content protection.

White-box cryptography aims to protect the confidentiality of the secret key of a cryptographic algorithm in a white-box environment. It is a technique to construct software implementations of a cryptographic algorithm that are sufficiently secure against a white-box attacker. In the academic literature, the focus has been mainly on the design of white-box implementations of block ciphers, an important subclass of symmetric-key cryptographic algorithms. In 2002, Chow, Eisen, Johnson and van Oorschot proposed the first published white-box implementation of the Advanced Encryption Standard (AES), one of the most prominent block ciphers at this time. However, two years later, Billet, Gilbert and Ech-Chatbi presented an efficient attack on this implementation, which motivated the design of three new white-box AES implementations offering more resistance against key extraction: the ones by Bringer, Chabanne and Dottax in 2006, by Xiao and Lai in 2009 and by Karroumi in 2010.

This doctoral thesis covers the design and analysis of white-box implementations of block ciphers, where the main contributions address the analysis of white-box AES implementations. Starting from the initial improvement of Billet et al.'s attack proposed by Tolhuizen in 2012, we present several additional improvements considerably reducing the overall work factor. Our improved version leads to some useful observations with respect to the design choices made in Chow et al.'s white-box AES implementation. Further, this doctoral thesis describes the analysis of the three newly proposed white-box AES implementations mentioned above. First, we show how to efficiently extract equivalent keys out of Bringer et al.'s white-box AES implementation; these equivalent keys yield functionally equivalent implementations. Second, we present a practical cryptanalysis of the white-box AES implementation proposed by Xiao and Lai. The cryptanalysis uses a modified variant of the linear equivalence algorithm presented by Biryukov, De Cannière, Braeken and Preneel as a building block. Additionally, we consider design generalizations of the Xiao-Lai white-box AES implementation and their impact on our cryptanalytic result. Third, we show that Karroumi's white-box AES implementation belongs to the class of white-box AES implementations specified by Chow et al. Consequently, Karroumi's implementation remains vulnerable to the attack it was designed to resist, i.e., Billet et al.'s attack and our improved version of this attack.

Based on the cryptanalytic results presented in this doctoral thesis and outlined above, it is shown that in early 2014 there does not exist a practical and secure white-box AES implementation published in the academic literature, even though AES is still considered to be a secure black-box block cipher. However, at the end of this thesis we discuss a new design principle proposed by Michiels and Gorissen that may lead to the construction of secure white-box AES implementations. All white-box AES implementations appeared in the academic literature so far are fixed-key; we present a new dynamic-key white-box technique that allows to update the cryptographic key in a more secure way than the known techniques.

Samenvatting

Cryptografische algoritmes beschermen data en communicatie in de aanwezigheid van een aanvaller. Indien deze algoritmes gebruik maken van een geheime sleutel, hangt hun veiligheid af van de geheimhouding van deze sleutel. Een aanvaller heeft daarom meestal als doel het bekomen van de geheime sleutel. In een traditionele black-box omgeving heeft de aanvaller slechts toegang tot de in- en uitgangen van een cryptografisch algoritme. Er is een toenemende vraag om cryptografische algoritmes te gebruiken in software applicaties die worden uitgevoerd op onbetrouwbare platformen; deze platformen zijn in het bezit en onder controle van een partij die slechte bedoelingen kan hebben. In het licht hiervan schiet de veronderstelling van een black-box omgeving duidelijk te kort. Daarom is er een nieuwe, meer realistische, white-box omgeving geïntroduceerd waarin een aanvaller volledige toegang heeft tot een software implementatie van een cryptografisch algoritme en bovendien ook volledige controle heeft over het platform waarop de implementatie wordt uitgevoerd. Praktijkvoorbeelden van een white-box omgeving kunnen gevonden worden bij digitale informatie beschermingssystemen zoals bijvoorbeeld Digital Rights Management of Betaal-TV systemen waarbij cryptografische algoritmes worden geïmplementeerd op bijvoorbeeld een smartphone, tablet of set-top box. Het uitlekken van de geheime sleutel zou leiden tot het teniet doen van de toegepaste beschermingstechniek.

White-box cryptografie richt zich op de geheimhouding van de sleutel van een cryptografisch algoritme wanneer dit algoritme wordt uitgevoerd in een white-box omgeving. In essentie is het een techniek om software implementaties van cryptografische algoritmes te bekomen die voldoende weerstand bieden tegen een white-box aanvaller. In de wetenschappelijke literatuur komen hoofdzakelijk white-box implementaties van blokcijfers, een belangrijke subklasse van de symmetrische sleutel cryptografische algoritmes, aan bod. In 2002 stelden Chow, Eisen, Johnson and van Oorschot de eerst gepubliceerde white-box implementatie van de Advanced Encryption Standard (AES) voor, waarbij AES één van de meest vooraanstaande huidige blokcijfers is. Twee jaar later echter presenteerden Biller, Gilbert en Ech-Chatbi een efficiënte aanval op deze implementatie. Dit

motiveerde de zoektocht naar nieuwe white-box AES implementaties die meer weerstand bieden tegen sleutelextractie. In de wetenschappelijke literatuur werden er drie systemen beschreven: de constructie van Bringer, Chabanne en Dottax in 2006, van Xiao en Lai in 2009, en van Karroumi in 2010.

Dit doctoraat behandelt het ontwerp en de analyse van white-box implementaties van blokcijfers, waarbij de belangrijkste bijdragen van toepassing zijn op het gebied van het analyseren van white-box AES implementaties. Uitgaande van de initiële verbetering van de aanval van Billet et al. voorgesteld door Tolhuizen in 2012, geven wij enkele aanvullende verbeteringen aan, die leiden tot een aanzienlijk vermindering van de totale werkfactor. Daarnaast leidt onze verbeterde versie van de aanval tot enkele nuttige inzichten in verband met de ontwerpstrategieën van de white-box AES implementatie van Chow et al. Verder behandelt dit doctoraat in detail de analyse van de drie nieuwe white-box AES implementaties uit de literatuur. Ten eerste tonen we aan hoe equivalente sleutels op een efficiënte wijze kunnen onttrokken worden uit de white-box AES implementatie van Bringer et al.; deze sleutels resulteren in implementaties die functioneel equivalent zijn. Ten tweede beschrijven we een praktische aanval op de white-box AES implementatie van Xiao en Lai, waarbij we een aangepaste versie van het lineaire equivalentie-algoritme voorgesteld door Biryukov, De Cannière, Braeken en Preneel gebruiken. Daarnaast beschouwen we ook veralgemeningen van de Xiao-Lai white-box AES implementatie en de impact ervan op ons resultaat. Ten derde tonen we aan dat Karroumi's white-box AES implementatie behoort tot de klasse van white-box AES implementaties zoals gespecifiëerd door Chow et al. Daarom blijft Karroumi's implementatie kwetsbaar voor de aanval waarvoor hij specifiek ontworpen was tegen bestand te zijn, namelijk de aanval van Billet et al. en onze verbeterde versie ervan.

De cryptanalytische resultaten bekomen in dit doctoraat tonen aan dat er begin 2014 geen enkele praktische en veilige white-box AES implementatie bestaat gepubliceerd in de wetenschappelijke literatuur, ook al is AES nog steeds een veilig black-box blokcijfer. Toch wordt er op het einde van dit proefschrift een veelbelovende nieuwe ontwerpstrategie besproken, geïntroduceerd door Michiels en Gorissen, die kan leiden tot de constructie van veilige white-box AES implementaties. Alle huidige white-box AES implementaties gepubliceerd in de wetenschappelijke literatuur zijn enkel gebaseerd op een vaste sleutel; wij stellen een nieuwe dynamische sleutel white-box techniek voor, die toelaat om de cryptografische sleutel te vernieuwen op een veiligere manier vergeleken met de bestaande technieken.

Contents

List of Figures

List of Tables

List of Symbols

General Notations

$\mathbb{F}_q$	the finite field with order q
$\mathbb{F}_q^n$	the n-dimensional vector space over $\mathbb{F}_q$
I_n	the n-bit identity matrix over $\mathbb{F}_2$
$0_{n\times n}$	the $n \times n$ zero matrix

AES-specific Notations

S	the AES S-box (unless otherwise specified)
`MC`	the 4×4 circulant `MixColumns` matrix over $\mathbb{F}_{256}$
$mc_{i,j}$	the coefficients of `MC` (indexed by i and j with $0 \leq i, j \leq 3$)
`SR`	the 128×128 non-singular matrix over $\mathbb{F}_2$ representing the `ShiftRows` operation (unless otherwise specified)
sr	the `ShiftRows` operation $sr : \{0,1,2,3\} \times \{0,1,2,3\} \to \{0,1,2,3\}$ is defined by $sr(i,j) = (j - i) \mod 4$
isr	the `ShiftRows` operation $sr : \{0,1,2,3\} \times \{0,1,2,3\} \to \{0,1,2,3\}$ is defined by $sr(i,j) = (j + i) \mod 4$

Operators and Functions

x, y	two n-bit words (for illustration purposes only)
X, Y	two permutations on $\mathbb{F}_2^n$ (for illustration purposes only)
$x \| y$ or (x, y)	the concatenation of the bit strings x and y (unless referred to as a pair of n-bit words in the case of (x, y))
$Y \circ X$	$Y \circ X : \mathbb{F}_2^n \to \mathbb{F}_2^n$ is defined by $Y \circ X(x) = Y\big(X(x)\big)$
$\big(X, Y\big)$	$\big(X, Y\big) : \mathbb{F}_2^{2n} \to \mathbb{F}_2^{2n}$ is defined by $\big(X, Y\big)(z) = \big(X(x), Y(y)\big)$ for each $2n$-bit word $z = (x, y)$ with $x, y \in \mathbb{F}_2^n$. In other words, $\big(X, Y\big)$ is the diagonal mapping with components X and Y (unless referred to as a pair of permutations on $\mathbb{F}_2^n$)
$\oplus$	the bitwise addition modulo 2 (exclusive-OR operation or XOR) and the addition operation in the AES polynomial representation of $\mathbb{F}_{256}$
$\otimes$	the multiplication operation in the AES polynomial representation of $\mathbb{F}_{256}$
$\oplus_l, \otimes_l$	the addition and multiplication operations in the polynomial representation of $\mathbb{F}_{256}$ associated with one of the 30 irreducible polynomials of degree 8 over $\mathbb{F}_{256}$ (indexed by l with $1 \leq l \leq 30$)
$\oplus_c$	$\oplus_c : \mathbb{F}_2^n \to \mathbb{F}_2^n$ with $c \in \mathbb{F}_2^n$ is defined by $\oplus_c(x) = x \oplus c$
$\langle z \rangle^L, \langle z \rangle^R$	the left and right nibble of the byte z in a graphical representation, i.e., the left and right nibble contain the four more significant bits and the four less significant bits within z, respectively, such that $z = \langle z \rangle^L \parallel \langle z \rangle^R$

Part I

Introduction: from Black-Box to White-Box

Chapter 1

Introduction: the Need for White-Box Cryptography?

Our society has always been information centric. Already in the early days, it was well understood that along with processing and exchanging information also comes the importance of its security. In the beginning, information security was primarily demanded by the government and the military as they dealt with information that was confidential and sensitive by nature. For example, the Roman emperor Julius Caesar is among other things still very well known for the use of his famous *Caesar cipher*.

The demand for information security and the way how information is secured have dramatically changed over the last decades. This is due to the penetration of the personal computer in our living rooms and the success of the Internet in the early 1990s; these phenomena caused a shift of the processing and exchange of information to the digital domain. Furthermore, due to the vast advancements in telecommunications, such as the growth of the Internet and the rapidly expanding digital mobile/wireless networks, an enormous number of parties all over the globe got interconnected through (mobile) communication devices. Billions of end-users started to use their electronic devices (connected to a network) for various tasks such as phone calls, e-mails, instant messaging and online financial transactions. Furthermore, the interconnectivity of billions of end-users made organizations aware of its enormous commercial value. Digital entertainment services arose such as digital television and virtual multimedia stores to purchase music or rent a movie. These trends as a result of the digital information infrastructure started to play a significant role in our daily lives.

Although interconnecting many different parties on a global scale clearly has its advantages, it also has its disadvantages. For example, digital information such as a bank transaction sent over a wireless connection is easy to intercept. Furthermore, these parties can be either *trusted*, such as government officials or legitimate organizations, or *untrusted*, such as end-users who may or may not behave maliciously. Therefore, it has become crucial to protect digital information. The science that studies this problem is called *cryptology*. Modern cryptology shows that the need for information security is no longer restricted to the government and the military.

The evolution of cryptology. Cryptology consists of two complementary branches: *cryptography* and *cryptanalysis*. Cryptographers design methods to protect information sent over an insecure communication channel, whereas cryptanalysts analyze the strength of these methods. Close interaction between both branches is important since the security of cryptographic methods often depends on the results obtained through cryptanalysis during many years.

Cryptography is the art of designing *cryptographic algorithms* to protect data or communication in the presence of an attacker (cf. Rivest [89]). The traditional setting considered in cryptography is discussed in the following, although the interpretation of *"in the presence of an attacker"* has strongly evolved over time as explained later. Two parties, referred to as sender and receiver, wish to exchange a confidential message over an insecure communication channel in such a way that the message is unintelligible for any third party (called the *attacker*) eavesdropping on the channel. In order to achieve this goal, both parties execute a cryptographic algorithm using some secret information to either transform (i.e., *encrypt*) the original message called the *plaintext* into an unreadable form called the *ciphertext* (at the side of the sender) or recover (i.e., *decrypt*) the plaintext from the ciphertext (at the side of the receiver). So, instead of the original message, the ciphertext is transmitted from which it is assumed to be impossible to recover the plaintext without knowing the secret information. This secret information is called *cryptographic keys*. If the cryptographic keys used for encryption and decryption are identical or can be derived from each other through a simple transformation, one speaks about *symmetric-key cryptography*. Throughout this doctoral thesis, symmetric-key cryptographic algorithms with identical keys are considered.

Now, when designing cryptographic algorithms, it is crucial to estimate the capabilities of the attacker as realistically as possible in the form of so-called *attack models*. Until the late 1990s, modern cryptography assumed that the end-points of the communication channel are trusted and that the cryptographic algorithms are executed in a secure environment. In this traditional model, known as the *black-box model*, the attacker has at most access to input/output behavior of the algorithm since he is only able to manipulate or eavesdrop on the channel between the trusted parties. So, until the late 1990s, the main idea was to employ a strong publicly known cryptographic algorithm on a trusted platform and rely on the secrecy of the cryptographic key.

However, starting in the second half of the 1990s, the attack landscape changed dramatically and the black-box model started to fall short. This was mainly due to the breakthrough of the computer and (mobile) communication devices, combined with the enormous success of the Internet and mobile networks interconnecting billions of users. This way, cryptography-enabled applications deployed on physically insecure devices (e.g., being susceptible to malware or viruses) were brought closer to a very broad audience of end-users, who may potentially behave maliciously. Therefore, the end-points of the communication channel could no longer be assumed to be trusted (or can even be considered to be the opponent) and naturally the traditional black-box assumption was no longer satisfied. This led to a need for more realistic attack models capturing the capabilities of more powerful attackers, as is explained in the following.

In practice, a cryptographic algorithm is implemented either in hardware or software and is often executed on an untrusted open platform, i.e., the electronic device on which the implementation is executed is in possession of and under control of the end-user. As a result, the end-user has (limited) access to the hardware/software implementation of the cryptographic algorithm, which clearly exceeds the capabilities of the attacker in the black-box model. In the academic literature, the shortcomings of the black-box model were highlighted by the appearance of the so-called *implementation attacks* in 1996 (cf. Kocher [59]). These attacks exploit *implementation-specific information*; if this information is leaked unintentionally out of the cryptographic implementation during its execution, one speaks about *side-channel information*. Examples of side-channels are execution time and power consumption. The attack model incorporating implementation-specific information is called the *grey-box model*.

However, in 2002, an even stronger attack model was introduced, called the *white-box attack context* (cf. Chow et al. [23]) or simply *white-box model*. This model, considered to be the strongest attack model from the perspective of the attacker, focuses solely on the software implementation of cryptographic algorithms executed on untrusted open platforms, i.e., platforms that are in possession of and under control of a potentially hostile end-user (e.g., a laptop, smartphone, tablet or set-top box). An attacker in the white-box model is assumed to have full access to the software implementation as well as full control over its execution environment. This allows the attacker for example to use debuggers with breakpoint functionality in order to observe and manipulate intermediate results of the implementation. Furthermore, the attacker is allowed to search the memory for stored keys (cf. Shamir and van Someren [97]), even after cooling the memory in order to preserve its state (cf. Halderman et al. [48]). For conventional cryptographic software implementations intended to be deployed in the black-box model only, such capabilities of the attacker form a real threat. Now, since it is often the case that all internal details about the cryptographic algorithm are known except for the secret key, the way how it is implemented in software becomes crucial. *White-box cryptography* focuses on this problem.

The existence of the more realistic grey-box and white-box attack models shows that a secure cryptographic implementation (either in hardware or software) can be considered as at least as important than the secure black-box properties of a cryptographic algorithm since the attacker will always try to exploit the weakest link in the security chain. If the attacker has access to the cryptographic hardware or software, then this weakest link can be found in either weak black-box design properties or weak implementation properties. Note that the untrusted end-points of the communication channel assumed in the grey-box and white-box models are not necessarily hostile end-users; for example, a plausible scenario would be that the physical device of a legitimate end-user on which the cryptographic algorithm is executed, is infected by malware.

1.1 White-Box Cryptography: a Use Case

White-box cryptography aims to construct software implementations of cryptographic algorithms in such a way that they offer a sufficient level of robustness against a white-box attacker. The part *'a sufficient level of robustness'* refers to protecting the confidentiality of the secret cryptographic key, which is also the primary goal of white-box cryptography. Later in this thesis, other white-box security goals are mentioned (see Sect. 3.4.2 for a discussion). Ultimately, an attacker in the white-box model should not have any advantage over an attacker in the black-box model with respect to extracting the secret cryptographic key, i.e., either having full access to and full control over the cryptographic software implementation or having solely access to the input/output behaviour of the implementation should not make any difference. Implementations obtained through the application of white-box cryptography are called *white-box implementations.*

The frequent occurrence of the white-box model in the real world, and thus the need for white-box cryptography, emanates from the ever increasing demand to deploy strong cryptographic algorithms in software applications that are executed on an untrusted open platform. As an illustration, we discuss the deployment of white-box cryptography in Digital Rights Management.

1.1.1 Digital Rights Management

Due to the digital revolution starting in the 1990s, copying and (illegally) distributing digital content has never been so easy. Therefore, content providers needed new technologies to protect their digital assets and to control the access and distribution of their copyright protected content. Such content protection schemes are known as *Digital Rights Management* or *DRM*. As expected, DRM can be found in many popular online digital multimedia (such as video, music, ebooks, apps etc.) stores nowadays. As an example, refer to the online Apple iTunes and iBooks Stores using Apple's FairPlay DRM system. Although Apple made music DRM free in 2009 [4], videos and ebooks purchased through the iTunes and iBooks Store still use Apple's FairPlay DRM system. Besides Apple, there are many other companies using DRM technology as well, such as Microsoft using Windows Media DRM for the Windows Media Player [76].

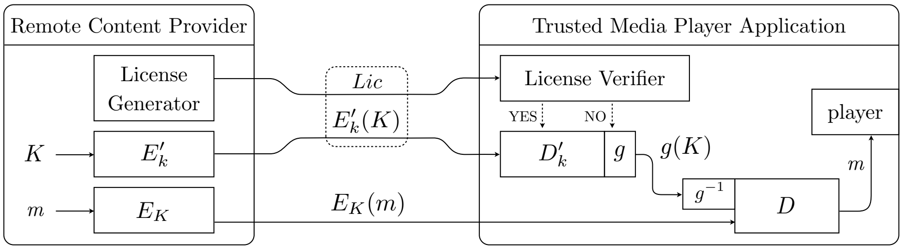


Figure 1.1: Use case of white-box cryptography: a simplified DRM model.

Cryptography typically forms one of the basic building blocks to enforce a DRM system. A simplified DRM model is depicted in Fig. 1.1, that serves merely as an example to sketch an environment that can benefit from white-box cryptography and is not intended to represent real-world deployed DRM architectures. This simplified DRM model comprises two parties: the *remote content provider* and the *trusted media player* (e.g., iTunes) executed on the end-user's untrusted platform (e.g., a PC). Here, it is assumed that the trusted media player is a solely software based application.

Now, the remote content provider delivers the copyright protected media content m to the authorized end-users in an encrypted form, consisting of the following three items:

1. the encrypted media content $E_K(m)$, where $E_K(\cdot)$ denotes a known symmetric-key encryption algorithm E using the secret content key K;
2. the encrypted content key $E'_k(K)$, where $E'_k(\cdot)$ denotes a (possibly different) known symmetric-key encryption algorithm E' using the secret

end-user's personal key k. The corresponding decryption algorithms of E and E' are denoted by D and D', respectively;

3. the DRM license *Lic*, comprising the restrictions (conditions) under which the end-user is allowed to access the digital content. Such a DRM license can for example specify a limited time frame (e.g., for movie rentals), or a maximum number of copies that can be made.

Typically, items 2 and 3 are sent simultaneously only upon request (i.e., purchase) of the end-user, whereas item 1 is available for download. Upon receipt of the above three items, the media player performs the following tasks. First, it verifies through the DRM license whether the end-user is allowed to gain access to the media content or not. After a positive confirmation ('YES'), the media player first decrypts the content key K using the end-user's personal key k and immediately applies an invertible encoding g to K, and then decrypts the media content using K after first applying the inverse encoding g^{-1} to $g(K)$.

Clearly, in the DRM setting (Fig. 1.1), the attacker (i.e., either a maliciously behaving end-user or malware executed on the end-user's device) steps out of the traditional black-box model and complies with the white-box model; he is in possession of and has control over the platform on which the media player application is executed. The attacker has the incentive to circumvent the restrictions posed by the DRM license. Being able to do so, a movie rental becomes as it were a movie purchase. He may achieve his goal by successfully performing one of the following three actions:

1. extract one of both decryption keys, i.e., either the content key K or the end-user's personal key k;
2. tamper with the license verifier code such that it always outputs 'YES';
3. intercept the media content m.

Countermeasures against the above mentioned attempts to bypass the DRM system are given below:

1. ensure that the used cryptographic keys are never revealed in the code implementing the media player application (either static or dynamic) or in the memory of the device on which the application is executed;
2. make the license verifier code tamper-resistant such that reverse engineering becomes a complex task and any attempt to modify the code results in breaking the functionality of the media player;

3. fingerprint the media content that unambiguously identifies the end-user such that a traitor (i.e., a malicious end-user illegally distributing his media content) can be traced back.

With respect to the first countermeasure, this can be achieved by constructing white-box implementations of both decryption algorithms to prevent decryption key extraction. Observe that between both decryption algorithms, the content key K only appears in an encoded form, i.e., $g(K)$. In [73], Michiels and Gorissen describe a technique to combine countermeasures 1 and 2. For a discussion on the practical security aspects of DRM and the involvement of white-box cryptography, refer to Schultz [95].

1.2 Outline and Contributions

This introductory chapter showed that there is a need for secure white-box implementations of cryptographic algorithms in our modern information society due to the digital revolution and the related demand for strong cryptographic algorithms as building blocks of software applications executed on untrusted platforms. The cryptographic algorithms performing symmetric-key encryption or decryption are called *ciphers*. Depending on how they process the plaintext or ciphertext, two classes can be identified: *block ciphers* and *stream ciphers*. In the past decades, there have been two prominent block ciphers: the *Data Encryption Standard (DES)* [68], standardized in 1977, and its successor the *Advanced Encryption Standard (AES)* [69], standardized in 2001.

The remainder of this doctoral thesis is dedicated to the design and analysis of white-box implementations of block ciphers, and in particular of white-box AES implementations. Before describing the outline of this thesis, the state-of-the-art of white-box implementations of block ciphers relevant to the results considered in this thesis is briefly discussed.

State-of-the-Art of White-Box Implementations of Block Ciphers. In 2002, Chow, Eisen, Johnson and van Oorschot introduced the research field of 'white-box cryptography' [24, 23] and presented generic techniques to construct practical white-box implementations of block ciphers; they applied their techniques to both DES and AES and proposed an example white-box DES implementation [24] and an example white-box AES implementation [23]. However, subsequent cryptanalytic results [51, 64, 105, 47, 13, 75] showed that both white-box implementations are insecure by efficiently extracting the embedded secret cryptographic key. One of the negative results by Billet, Gilbert and Ech-Chatbi [13], i.e., a practical attack on the white-box AES implementation of Chow et al., acted as a trigger for research dedicated to designing new white-box AES implementations to reinforce its white-box security. This led to three new white-box AES designs published in the academic literature: the ones by Bringer, Chabanne and Dottax in 2006 [20], by Xiao and Lai in 2009 [107] and by Karroumi in 2010 [53]. All three proposals were claimed to be white-box secure by offering resistance against the attack of Billet et al.

For a brief overview about white-box cryptography and white-box implementations, refer to Joye [52], Michiels [70] or Wyseur [104]. For a detailed overview, refer to Wyseur [103] or Muir [78].

Outline and Summary of Contributions. The outline and the main contributions presented in the different chapters of this thesis are summarized below.

Chapter 2: Design and Analysis of Block Ciphers: the Evolution. This chapter discusses the most common design principles as well as the security of block ciphers. With regard to the design of block ciphers, the specification of the prominent and widely used Advanced Encryption Standard (AES) is given, which is necessary for the description of the white-box AES implementations in Chapters 3-6. With regard to the assessment of the security of block ciphers, the evolution of the attack models and their corresponding cryptanalytic techniques are discussed. This chapter concludes with the observation that the most appropriate attack model in which AES deployments should be considered, is the white-box model, and that this poses a major challenge for designing secure software implementations of AES (or of block ciphers in general). This challenge is addressed by white-box cryptography, that is the subject of the following chapters.

Chapter 3: Design and Analysis of White-Box Implementations. This chapter elaborates on the objective of white-box cryptography, namely the construction of secure software implementations of cryptographic algorithms employed in the white-box model, and describes the generic white-box techniques introduced by Chow et al. [24, 23] in 2002 to achieve this goal in a practical manner. Further, this chapter elaborates in detail on the design and analysis of the white-box AES implementation of Chow et al. [23]. With regard to the analysis part, the practical attack of Billet et al. [13] (referred to as the BGE attack) and the generic white-box attack of Michiels, Gorissen and Hollmann [75] are discussed. As mentioned before, in response to these attacks, three new white-box AES implementations appeared in the academic literature for which it was claimed that they withstand the BGE attack.

Chapter 4: Revisiting the BGE Attack. Starting from the initial improvement of the BGE attack presented by Tolhuizen in 2012 [101], in this chapter we present several additional improvements of the BGE attack and show that the original work factor of 2^{30} can be reduced to 2^{22} if Tolhuizen's and our improvements are combined. Additionally, we show that Karroumi's white-box AES implementation [53] belongs to the class of white-box AES implementation specified by Chow et al. in [23]. As a result, Karroumi's implementation remains vulnerable to the attack it was designed to resist, i.e., the BGE attack and our improved version.

The main part of this work was published in [36] (ePrint 2013/450) and is joint work with Peter Roelse (Irdeto B.V., The Netherlands) and Bart Preneel (KU Leuven, Belgium). This result also appeared in [63] (SAC 2013) as part of a merged paper with Tancrède Lepoint (École Normale Supérieure and CryptoExperts, France) and Matthieu Rivain (CryptoExperts, France).

Chapter 5: Cryptanalysis of the Xiao-Lai White-Box AES Implementation. In this chapter we present a practical attack on the white-box AES implementation of Xiao and Lai [107]. Although Michiels et al.'s white-box attack [75] can be applied to the Xiao-Lai implementation, the attack is generic by nature and hence is not optimized, which results in a rather large work factor (estimated at at least 2^{49}; note that Michiels et al. do not provide a clean discussion on the overall work factor of their attack). Therefore, we show how specific properties of both AES as well as of the white-box implementation itself can be exploited in order to obtain an optimized non-generic attack with a work factor of 2^{32}. Additionally, we consider design generalizations of the Xiao-Lai white-box AES implementation and investigate their impact on the work factor of our cryptanalysis.

The main part of this work was published in [35] (SAC 2012) and is joint work with Peter Roelse (Irdeto B.V., The Netherlands) and Bart Preneel (KU Leuven, Belgium).

Chapter 6: Cryptanalysis of Bringer et al.'s Perturbated White-Box AES Implementation. This chapter first describes the novel white-box technique introduced by Bringer et al. in 2006 [20] as well as its application to both AES and a variant of AES (denoted by AES*). Next, we present a practical attack on the white-box AES* implementation of Bringer et al.; the attack extends naturally to Bringer et al.'s white-box implementation of standard AES.

This work was published in [38] (INDOCRYPT 2010) and is joint work with Brecht Wyseur (Nagravision S.A., Switzerland) and Bart Preneel (KU Leuven, Belgium).

Chapter 7: State-of-the-Art and Q&A. This chapter presents the state-of-the-art of white-box AES implementations; first a comparison is given between the size and performance of software AES implementations deployed in the white-box and black-box model, and second the cryptanalytic results of white-box AES implementations are summarized and conclusions are drawn. Further, this chapter discusses a few directions for designing new secure white-box AES implementations. Additionally, some techniques are described to construct white-box implementations with the possibility to update the cryptographic key. As part of this, we present a new dynamic-key white-box technique different from the existing techniques.

This work was published in [91] (Patent 2013/139380) and is joint work with Peter Roelse (Irdeto B.V., The Netherlands).

Chapter 8: Conclusions and Future Research.

Other results obtained during this doctoral research, that are not related to white-box cryptography and thus not considered in this thesis, were published in [34, 37]. A list of all publications can be found on p. 217.

Chapter 2

Design and Analysis of Block Ciphers: the Evolution

The deployment of a cryptographic algorithm has the objective to achieve security requirements that are application dependent. The requirements listed below tend to be of general importance:

Confidentiality ensures that an attacker is unable to gain any information about the content of the communication while listening to the communication channel;

Entity authentication allows the identification of an entity such as a person, a physical device or a software program;

Data authentication allows the detection of any unauthorized modification (by an attacker with access to the communication channel) of the origin and/or the content of the communication. Data authentication is in fact a synonym for *data integrity* since it is considered hard to tamper with a message without changing its origin.

Since the cryptographic algorithm is typically only a small but important part of the application in which it is deployed, its success in achieving the above security requirements not only depends on its obtained level of security, but also on how the application itself is engineered. In this thesis, the focus is on the security of (the implementation of) cryptographic algorithms under the assumption that the application in which the algorithms are deployed contains no 'security' weaknesses. In order to evaluate this security, it is crucial to take into account the environment in which the application is deployed. This is done by means of an attack model that is assumed to capture the capabilities of an attacker as realistically as possible. In order to achieve a sufficient level of security, (the implementation of) the cryptographic algorithm should be properly designed with respect to the appropriate attack model.

The class of *symmetric-key cryptographic algorithms* covers those algorithms in which all entities participating in the communication share the *same* secret key material. The shared secret keys can be either identical or derived from one another via a simple transformation. Within this class, one can distinguish three different types of algorithms: block ciphers, stream ciphers and message authentication code (MAC) algorithms. The speed at which symmetric-key algorithms operate makes them a powerful tool for performing encryption/decryption or authentication. This chapter focuses on the class of *block ciphers*, which are widely deployed in many applications and thus can be considered as a crucial cryptographic primitive. Furthermore, block ciphers can be used as building blocks to construct stream ciphers and MAC algorithms; however this topic falls out of the scope of this thesis.

This chapter presents a broad overview of the design and analysis of block ciphers. Most inspiration for writing this introductory chapter comes from the books published by Knudsen and Robshaw [58] and by Daemen and Rijmen [31]. First, the class of block ciphers is defined and it is explained what is meant by an *ideally designed* block cipher. Next, the most common design principles are discussed in order to obtain practical block cipher designs approximating the ideal cipher. As an example, the design specification of a widely used block cipher called the Advanced Encryption Standard (AES) is given. Second, it is discussed when a practical block cipher is considered to be insecure. With respect to the assessment of the security of a block cipher, the attack models appearing in practice are discussed, as well as their evolution in time due to the dramatic shift of applications in which block ciphers are deployed. Additionally, the most common cryptanalytic techniques corresponding to the different attack models are described. As the attack models only become more severe, and hence the corresponding cryptanalytic techniques more powerful, this naturally leads to the importance of white-box cryptography, which is discussed in detail in the next chapter.

2.1 Defining Block Ciphers

A block cipher consists of a pair of cryptographic algorithms, one for encryption and the other for decryption, that operates on blocks of data (i.e., fixed-length bit strings). The class of block ciphers is defined as follows.

Definition 1 (Block cipher)**.** *An n_b-bit block cipher E is a deterministic function mapping an n_b-bit plaintext block m onto an n_b-bit ciphertext block c under the action of an n_k-bit secret key k:*

$$E : \mathbb{F}_2^{n_b} \times \mathbb{F}_2^{n_k} \to \mathbb{F}_2^{n_b} : (m, k) \mapsto c = E_k(m) \ , \tag{2.1}$$

where n_b and n_k denote the block size and key size of the block cipher, respectively. The key-dependent mapping $E_k(\cdot)$ is a permutation on n_b bits for each $k \in \mathbb{F}_2^{n_k}$, and each different value of k results in a distinct permutation. In this way, a block cipher E specifies a family $\mathcal{BC}$ of 2^{n_k} permutations on $\mathbb{F}_2^{n_b}$, where 2^{n_k} denotes the total number of possible key values.

Block ciphers are used to encrypt/decrypt n_b-bit message blocks in order to provide data confidentiality. The process of *encryption* is given by the n_b-bit bijective mapping $E_k(\cdot)$ defined by (2.1). The process of *decryption* is given by the inverse mapping $E_k^{-1}(\cdot) = D_k(\cdot)$ instantiated with the same secret key k such that it satisfies the correctness property: $\forall m \in \mathbb{F}_2^{n_b}, \forall k \in \mathbb{F}_2^{n_k} : D_k\big(E_k(m)\big) = m$. Typical values for the block size n_b are 64 or 128 bits.

There are $(2^{n_b})! \approx (2^{n_b}/2)^{2^{n_b}}$ distinct permutations on $\mathbb{F}_2^{n_b}$, hence a practical block cipher specifies only a small fraction of all permutations. The size and composition of the specified family $\mathcal{BC}$ is completely determined by the key size n_k and the design of the block cipher, respectively. Ideally, a random element of the set of all $(2^{n_b})!$ distinct permutations on $\mathbb{F}_2^{n_b}$ is assigned to each of the 2^{n_k} permutations $E_k(\cdot)$. This brings us to the concept of an *ideal block cipher* (cf. Black [16]).

Definition 2 (Ideal block cipher [16])**.** *A block cipher E is called ideal, if the family $\mathcal{BC}$ of 2^{n_k} permutations on $\mathbb{F}_2^{n_b}$ specified by E is selected uniformly at random from the set of all $(2^{n_b})!$ distinct permutations on $\mathbb{F}_2^{n_b}$.*

2.2 Block Cipher Design

The primary objective of a designer is to approximate ideal block ciphers, even though the design of block ciphers is assumed to be highly structured. Such structure within block ciphers is inevitable since it is impractical to store 2^{n_k} truly randomly chosen permutations on $\mathbb{F}_2^{n_b}$. However, even ideal block ciphers (Def. 2) remain susceptible to a certain class of black-box attacks known as the *generic attacks*, i.e., attacks that do not exploit the internal structure of the block cipher. Section 2.5.1 elaborates on the various black-box attacks.

2.2.1 Confusion and Diffusion

Within the field of block cipher design, Shannon [99] laid in 1949 the foundation by introducing the concepts of *confusion* and *diffusion*, concepts that up to now are still widely considered in the process of designing new block ciphers. The motivation behind these concepts was to hide the redundancy inherently present in the plaintext. Both concepts can be described as follows:

Confusion captures the complex way in which the ciphertext bits depend on the plaintext bits and key bits. The goal is to make this relationship as complicated as possible such that it is hard (and preferably impossible) to be exploited by an attack. Basic components achieving confusion are non-linear *substitution boxes* (*S-boxes*) that are typically implemented as lookup tables (see Def. 3 below). However, the storage requirement of lookup tables (see Property 1 below) imposes a restriction on their input size.

Diffusion captures the influence of each plaintext bit and each key bit on the ciphertext bits. The goal is to make this influence as large as possible. The block cipher should exhibit the property that flipping a single plaintext bit or key bit results in flipping each ciphertext bit with probability $\frac{1}{2}$. This propagation property is known as the *avalanche effect*. Basic components achieving diffusion are (i) linear *diffusion boxes* (*D-boxes*) comprising wide linear operations or (ii) permutations operating at bit-level or at the level of bundles of bits (e.g., at byte-level), also referred to as *bit transpositions* or *bundle transpositions* respectively. In contrast to the non-linear S-boxes, there is no restriction on the input size of these diffusion components.

Typically, a strong block cipher contains a high degree of confusion and diffusion; this requires a close interaction between the following three operations: *substitutions*, *linear operations* and *transpositions*. Later, in Sect. 2.5.1, some desirable properties of S-boxes and diffusion components are discussed with respect to differential and linear cryptanalysis.

Definition 3 (Lookup table). *A lookup table $\mathcal{L}$ mapping m bits to n bits is a specific representation of any given function $f : \mathbb{F}_2^m \to \mathbb{F}_2^n$, i.e., $\mathcal{L}$ is an array of 2^m n-bit entries, denoted by $\mathcal{L}[i]$ for $i = 0, 1, \ldots, 2^m - 1$, with $\mathcal{L}[i] = f(bin(i))$ where $bin(i) \in \mathbb{F}_2^m$ denotes the binary representation of i.*

Property 1 (Storage requirement of a lookup table). *A lookup table mapping m bits to n bits requires a total of $2^m \cdot n$ bits of storage. As this amount is exponential in the table's input size m (measured in bits), the storage requirement becomes quickly impractical for large m.*

2.2.2 Constructions

The mix of non-linear substitutions and linear diffusion operations is a crucial component of most block cipher designs. This mix can be obtained in various ways such as the following two prominent and efficient constructions adopted by many important block ciphers (e.g., DES [68], AES [69], Serpent [7], ...): *Feistel ciphers* and *Substitution-Permutation Network (SPN) ciphers.* Both constructions belong to the class of *iterative block ciphers*, also known as *product ciphers*, introduced by Shannon in [99]. A product cipher is a block cipher made by iterating a fairly simple key-dependent round function many times. While a single key-dependent round function acts as a weak block cipher, the iteration of several round functions may result in a strong block cipher. A formal definition is given in the following.

Definition 4 (Iterative block cipher/product cipher). *An n_b-bit block cipher E is called an iterative block cipher with R rounds if for each key $k \in \mathbb{F}_2^{n_k}$, the bijective mapping E_k on $\mathbb{F}_2^{n_b}$ comprises the iterative application of R key-dependent round transformations $E^{(r)}_{k^{(r)}}$ with $1 \leq r \leq R$, i.e.,*

$$E_k = E^{(R)}_{k^{(R)}} \circ \cdots \circ E^{(2)}_{k^{(2)}} \circ E^{(1)}_{k^{(1)}} \quad , \tag{2.2}$$

where each key-dependent round transformation $E^{(r)}_{k^{(r)}}$ is a permutation on $\mathbb{F}_2^{n_b}$ and where $k^{(r)}$ $(1 \leq r \leq R)$ denotes the r^{th} round key. All round keys are derived from the secret key k through the application of the key scheduling algorithm $\boldsymbol{ks}$, defined as

$$\boldsymbol{ks} : \mathbb{F}_2^{n_k} \to \mathbb{F}_2^{n_K} : k \mapsto K = (k^{(1)} \parallel k^{(2)} \parallel \cdots \parallel k^{(R)}) \quad , \tag{2.3}$$

where K is called the expanded key and represents the concatenation of all round keys and where n_K denotes the length of K.

Within the class of iterative block ciphers (Def. 4), two (not distinct) subclasses can be identified: the *iterated block ciphers* and the *key-alternating block ciphers*. For *iterated block ciphers*, all key-dependent round transformations are identical, i.e., $E^{(r)}_{k^{(r)}} = E_{k^{(r)}}$ $(1 \leq r \leq R)$. For *key-alternating block ciphers*, all key-dependent round transformations are a two-layered structure consisting of a XOR with the round key followed by a key-*in*dependent round transformation, i.e., $E^{(r)}_{k^{(r)}} = E^{(r)} \circ \oplus_{k^{(r)}}$ $(1 \leq r \leq R)$. The intersection of the two subclasses determines the class of *key-iterated block ciphers*, which essentially is defined as key-alternating block ciphers for which all key-independent round transformations are identical, i.e., $E^{(r)}_{k^{(r)}} = E \circ \oplus_{k^{(r)}}$ $(1 \leq r \leq R)$.

Substitution-Permutation Network (SPN) Ciphers. In the following, one of the most widely accepted constructions to build iterative block ciphers is described, i.e., the class of *Substitution-Permutation Network (SPN) ciphers.* The description below is generic, hence deviations from it are most likely to occur in modern block cipher designs.

Definition 5 (Substitution-Permutation Network (SPN) cipher)**.** *An n_b-bit SPN cipher with R rounds is an iterative block cipher where each key-dependent round transformation $E^{(r)}_{k^{(r)}}$ ($1 \leq r \leq R$) is a permutation on $\mathbb{F}_2^{n_b}$ and typically consists of three layers (not necessarily in the following order): (i) the confusion layer comprising (the parallel execution of) S-boxes, (ii) the diffusion layer comprising D-boxes and/or (bit or bundle) transpositions, and (iii) the round key $k^{(r)}$ addition layer.*

For SPN ciphers, decryption is performed by inverting the encryption process while taking the round keys in reversed order. As a consequence, each key-dependent round transformation $E^{(r)}_{k^{(r)}}$ ($1 \leq r \leq R$) needs to be a bijective mapping on $\mathbb{F}_2^{n_b}$, which means that all layers defined in Def. 5 need to be invertible as well. Because of the restriction on the input size of S-boxes due to the storage requirement (Property 1, p. 16), the n_b-bit confusion layer typically consists of s m_i-bit bijective S-boxes S_i ($i = 1, 2, \ldots, s$) (each representing a permutation on $\mathbb{F}_2^{m_i}$) in parallel with $\sum_{i=1}^{s} m_i = n_b$. The n_b-bit diffusion layer typically comprises a bijective affine/linear mapping on $\mathbb{F}_2^{n_b}$ combined with a (bit or bundle) transposition. Note that (bit or bundle) transpositions are invertible by definition.

An advantage of SPN ciphers is that they tend to have good diffusion properties since the diffusion layer typically operates on all n_b bits of the state simultaneously. Depending on the choice of the diffusion operations, *full diffusion* can already be achieved after two rounds (e.g., for AES). The concept of full diffusion is described in Sect. 2.5.1.

Example. The *Advanced Encryption Standard* or *AES* [69] abbreviated is a 128-bit key-iterated SPN block cipher. A detailed description of AES is given in the next section, i.e., Sect. 2.3.

2.3 Advanced Encryption Standard (AES)

Let us start by sketching the historical context of the AES. In 1973, the National Bureau of Standards (NBS), now called the National Institute of Standards and Technology (NIST), issued a request for proposals for a block cipher that can be used for the protection of sensitive government data. This led to the adoption of the Data Encryption Standard (DES), developed by IBM and the NSA and published in FIPS 46 [68] in 1977. However, due to the increasing computational power and the various cryptanalytic results (e.g., linear cryptanalysis [66] – see Sect. 2.5.1), it was believed that the DES key size of 56 bits became too small to provide a sufficiently high level of security in many applications. Additionally, the rather slow software performance of DES became unacceptable for many software applications. Therefore, in 1997, NIST issued a call for proposals for a new block cipher, i.e., the Advanced Encryption Standard (AES) as the successor of DES. In 2000, the block cipher Rijndael [28, 31], designed by Daemen and Rijmen, with a fixed block size of 128 bits and a variable key size of 128, 192 or 256 bits was selected as the AES and was published in FIPS 197 [69] in 2001. The lifespan of AES is expected to last for another 20 to 30 years.

2.3.1 Specification

The Advanced Encryption Standard (AES), published in FIPS 197 [69], is a 128-bit key-iterated SPN block cipher supporting three different key sizes, i.e., 128, 192, or 256 bits, denoted by AES-128, AES-192 or AES-256, respectively. In general, AES consists of R rounds and has $R+1$ 128-bit round keys that are derived from the secret AES key using the AES key scheduling algorithm; R depends on the key size, i.e., $R = 10, 12$ or 14 in the case of AES-128, AES-192 or AES-256, respectively. Each AES round and the operations within a round update a 16-byte state; the initial and final state are the AES plaintext and ciphertext, respectively. An AES state is represented by the conventional 4×4 byte array $[\text{state}_{i,j}]_{0 \leq i,j \leq 3}$, called the *state array.*

AES can be described elegantly by interpreting the bytes of the state as elements of the finite field $\mathbb{F}_{256}$, and by defining AES operations as mappings over this field. In the remainder of this thesis, the specific polynomial representation of the field elements of $\mathbb{F}_{256}$ as defined in [69] is referred to as the *AES polynomial representation.* Now, an AES round comprises the following operations:

`SubBytes` applies the AES S-box to every byte of the state. AES uses one fixed S-box, denoted by S, which is a non-linear bijective mapping from 8 bits to 8 bits, defined as $S(x) = A(x^{-1})$ with A a fixed bijective affine mapping on $\mathbb{F}_2^8$. The inversion x^{-1} is computed in $\mathbb{F}_{256}$, with $\texttt{00}^{-1} = \texttt{00}$. The AES S-box has been very carefully designed; e.g., it has a high algebraic degree (exploited by the *algebraic degree attack* discussed in Sect. 3.5.1) and very good linear/differential lower bounds.

`ShiftRows` applies a circular shift to the left by i byte positions to each row i of the state array for $0 \leq i \leq 3$. Observe that the row indexed by $i = 0$ remains invariant. With respect to the `ShiftRows` operation, the function $sr(i,j) : \{0,1,2,3\} \times \{0,1,2,3\} \to \{0,1,2,3\}$ is introduced that is defined as $sr(i,j) = (j - i) \bmod 4$ such that

$$\text{state}_{i,j} \quad \xrightarrow{\texttt{ShiftRows}} \quad \text{state}_{i,sr(i,j)} \qquad \text{for } 0 \leq i,j \leq 3 \ .$$

`MixColumns` is a linear operation on $\mathbb{F}_{256}^{16}$ that applies 4 instances of the `MixColumns` operation in parallel to the 16-byte state. The `MixColumns` operation itself is represented by an invertible 4×4 circulant matrix `MC` over $\mathbb{F}_{256}$. The 16 entries of $\texttt{MC} = [mc_{i,j}]_{0 \leq i,j \leq 3}$ are called the `MixColumns` coefficients $mc_{i,j}$; they are elements of the set $\{\texttt{01}, \texttt{02}, \texttt{03}\}$. To update the state, each column j of the state array is interpreted as a 4×1 vector over $\mathbb{F}_{256}$ and multiplied by `MC` for $0 \leq j \leq 3$:

$$\begin{pmatrix} \text{state}_{0,j} \\ \text{state}_{1,j} \\ \text{state}_{2,j} \\ \text{state}_{3,j} \end{pmatrix} \leftarrow \texttt{MC} \cdot \begin{pmatrix} \text{state}_{0,j} \\ \text{state}_{1,j} \\ \text{state}_{2,j} \\ \text{state}_{3,j} \end{pmatrix} \text{ with } \texttt{MC} = \begin{pmatrix} \texttt{02 03 01 01} \\ \texttt{01 02 03 01} \\ \texttt{01 01 02 03} \\ \texttt{03 01 01 02} \end{pmatrix} , \quad (2.4)$$

for $j = 0,1,2,3$. Throughout this thesis, it is assumed to be clear from the context whether the `MixColumns` operation refers to the linear operation on $\mathbb{F}_{256}^{16}$ or the 4×4 matrix `MC` over $\mathbb{F}_{256}$.

`AddRoundKey` is a bitwise addition modulo two (i.e., XOR) of a 128-bit round key $k^{(r)}$ $(1 \leq r \leq R+1)$, represented by a 4×4 byte array $[k_i^{(r,j)}]_{0 \leq i,j \leq 3}$, and the state array.

In accordance to the SPN structure, the confusion layer comprises the `SubBytes` step, and the diffusion layer consists of the combination of `ShiftRows` and `MixColumns`. As a result of the *wide trail strategy* (see Sect. 2.5.1, p. 39), the diffusion layer has been designed carefully such that it achieves 'full diffusion' after only two rounds.

Figure 2.1a depicts the conventional description of AES-128, which consists of 10 rounds and has 11 128-bit round keys $k^{(r)}$ $(1 \leq r \leq 11)$. There are several equivalent ways to describe AES-128 by defining the boundaries between rounds in different ways, one of which is depicted in Fig. 2.1b where $\hat{k}^{(r)}$ for $1 \leq r \leq 10$ is the result of applying `ShiftRows` to $k^{(r)}$. As is discussed in Chapter 3, the alternative description (Fig. 2.1b) is used as a reference for white-box AES-128 implementations. Observe that (i) a *pre-whitening* `AddRoundKey` step $(k^{(1)})$ is performed before the first round in the conventional description (Fig. 2.1a), and a *post-whitening* AddRoundKey step ($k^{(11)}$) is performed after the final round in the alternative description (Fig. 2.1b), and (ii) the MixColumns step is omitted in the final round. The pre/post-whitening step prevents the 'peeling-off' of parts of the first/final round, whereas the omission of the MixColumns step ensures a certain symmetry between AES encryption and decryption.

```
state ← plaintext
state ← AddRoundKey(state,$k^{(1)}$)
for $r = 1$ to 9 do
    state ← SubBytes(state)
    state ← ShiftRows(state)
    state ← MixColumns(state)
    state ← AddRoundKey(state,$k^{(r+1)}$)
end for
state ← SubBytes(state)
state ← ShiftRows(state)
state ← AddRoundKey(state,$k^{11}$)
ciphertext ← state
```

(a) Conventional description of AES-128.

```
state ← plaintext
for $r = 1$ to 9 do
    state ← ShiftRows(state)
    state ← AddRoundKey(state,$\hat{k}^{(r)}$)
    state ← SubBytes(state)
    state ← MixColumns(state)
end for
state ← ShiftRows(state)
state ← AddRoundKey(state,$\hat{k}^{(10)}$)
state ← SubBytes(state)
state ← AddRoundKey(state,$k^{(11)}$)
ciphertext ← state
```

(b) Alternative description of AES-128.

Figure 2.1: Equivalent descriptions of AES-128.

As white-box AES implementations typically do not implement the AES key scheduling algorithm, the description of this algorithm is not required. For details about the AES key scheduling algorithm, refer to FIPS 197 [69]. However, note that the AES key scheduling algorithm is invertible, and in the case of AES-128, it has the property that the 128-bit AES key can be computed if one of the round keys is known. In fact, the first round key $k^{(1)}$ is equal to the AES key in the case of AES-128.

2.3.2 Standard Software Implementation

One of the requirements during the AES competition was that the AES candidates should have a high performance in both hardware and software. This requirement was met by Rijndael, which eventually became AES. In [31], Daemen and Rijmen presented efficient AES implementations for various platforms. Here, the focus is on the performance-oriented software implementation on 32-bit (or higher) processors, the one that is used most often in practice (e.g., in OpenSSL [82] – see the `aes_core.c` file in "openssl-1.0.1e/crypto/aes/").

As pointed out in [31, p. 56], very fast implementations on processors with word length 32 can be obtained by merging SubBytes and MixColumns together into a single set of four lookup tables. Observe that in the alternative 'implementation-friendly' description of AES-128 (Fig. 2.1b) both steps are already adjacent in the round function which simplifies the merger. Concerning the `MixColumns` operation, the 4×4 matrix `MC` can be split into four 4×1 submatrices over $\mathbb{F}_{256}$: $\mathtt{MC}_l$ is defined as column l of `MC` for $l = 0, 1, 2, 3$. Using this notation, the `MixColumns` matrix-vector multiplication (2.4) is decomposed into a XOR of four 32-bit values and is given by

$$\begin{pmatrix} \mathrm{state}_{0,j} \\ \mathrm{state}_{1,j} \\ \mathrm{state}_{2,j} \\ \mathrm{state}_{3,j} \end{pmatrix} \leftarrow \bigoplus_{l=0}^{3} \mathtt{MC}_l \cdot \mathrm{state}_{l,j} \qquad \text{for } j = 0, 1, 2, 3 \ . \tag{2.5}$$

By taking the `SubBytes` step, which precedes the `MixColumns` step, into account, (2.5) becomes

$$\begin{pmatrix} \mathrm{state}_{0,j} \\ \mathrm{state}_{1,j} \\ \mathrm{state}_{2,j} \\ \mathrm{state}_{3,j} \end{pmatrix} \leftarrow \bigoplus_{l=0}^{3} \mathtt{MC}_l \cdot S(\mathrm{state}_{l,j}) \qquad \text{for } j = 0, 1, 2, 3 \ , \tag{2.6}$$

where S denotes the AES S-box. Now, `SubBytes` and `MixColumns` are merged together by constructing four $\mathtt{SMC}_l$ ($l = 0, 1, 2, 3$) lookup tables, each table mapping 8 bits to 32 bits and composing both the AES S-box and a quarter of the `MixColumns` operation: $\mathtt{SMC}_l = \mathtt{MC}_l \circ S$ for $l = 0, 1, 2, 3$. Note that in [31], these tables are denoted by T_l ($l = 0, 1, 2, 3$).

With the alternative description of AES-128 depicted in Fig. 2.1b, very fast implementations can be obtained as follows: (i) `ShiftRows` is implemented simply by means of a byte transposition, i.e., appropriately shifting the bytes, (ii) `AddRoundKey` of a 128-bit round key is implemented as four 32-bit XOR operations, and (iii) `SubBytes` and `MixColumns` are implemented by means of 16 table lookups (using the $\mathtt{SMC}_l$ tables) and 12 32-bit XOR operations:

$$\begin{pmatrix} \mathrm{state}_{0,j} \\ \mathrm{state}_{1,j} \\ \mathrm{state}_{2,j} \\ \mathrm{state}_{3,j} \end{pmatrix} \leftarrow \bigoplus_{l=0}^{3} \mathtt{SMC}_l(\mathrm{state}_{l,j}) \qquad \text{for } j = 0, 1, 2, 3 \ . \tag{2.7}$$

Observe that in the final round, the `MixColumns` step is omitted. One option is to implement the AES S-box as an additional lookup table mapping 8 bits to 8 bits. However, each $\mathtt{SMC}_l$ table already implicitly implements the AES S-box since each `MixColumns` submatrix $\mathtt{MC}_l$ contains exactly two `MixColumns` coefficients equal to `01`. Hence, the AES S-box can be extracted out of any of the $\mathtt{SMC}_l$ ($l = 0, 1, 2, 3$) tables by considering the appropriate eight output bits.

This standard software AES implementation is referred to as the *lookup-table-based AES implementation* by Daemen and Rijmen in [31]. The total implementation size equals 4 kB, which corresponds to the size required to store the four $\mathtt{SMC}_l$ ($l = 0, 1, 2, 3$) lookup tables (Property 1, p. 16): each $\mathtt{SMC}_l$ table requires $2^8 \cdot 32$ bits (= 1 kB) of storage space. As is discussed in Chapter 3, a somewhat similar lookup-table-based AES implementation as the one described above is used as a starting point to construct white-box AES implementations.

2.4 Security

During the process of designing a new block cipher, the desired security level needs to be taken into account. Below, different levels of security are highlighted. As it turns out, the achieved security of block ciphers is often evaluated by considering their corresponding state-of-the-art cryptanalytic results, which depends on a certain security notion comprising the following two factors: (i) the goal of the cryptanalyst and (ii) the attack model that represents the hostile environment in which the block cipher is deployed (based on the application in which it is used). Both factors, and more specifically the evolution of attack models over time, are discussed at the end of this section.

2.4.1 Perfect Security

The highest level of security is called *perfect security*, introduced by Shannon [99] and defined in the following.

Definition 6 (Perfect security)**.** *A cipher is called perfectly secure if the ciphertext does not reveal any information about the plaintext, or in other words, if the plaintext and the ciphertext are statistically independent.*

Perfect security is also often called *unconditional security against unbounded adversaries*, where an unbounded adversary refers to an attacker having access to an unlimited amount of computing power. Shannon showed that the Vernam cipher [102] (also known as one-time pad[1]) provides perfect security if the secret key is randomly chosen and only used once. Furthermore, he proved that in order to achieve perfect security, the entropy of the key needs to be at least as high as the entropy of the plaintext, which implies that the key needs to be at least as long as the plaintext and may never be reused again.

[1]The Vernam cipher, first described by Miller [77] in 1882 and later in 1917 (re)invented by Vernam, is a stream cipher with the following simple encryption scheme: $c_i = m_i \oplus k_i$, where m_i, k_i and c_i denote the i^{th} bit of the plaintext, key and ciphertext respectively.

In the field of block ciphers, perfect security is approached by *ideal block ciphers* (see Def. 2). However, as mentioned earlier, even implementing an ideal block cipher is impractical since it requires the storage of 2^{n_k} randomly chosen permutations on $\mathbb{F}_2^{n_b}$. As a result, practical block ciphers approximate ideal block ciphers while being highly structured, and thus containing less entropy. Therefore, practical block ciphers can achieve at most *computational security.*

2.4.2 Computational Security

Instead of assuming unbounded adversaries as is the case for perfect security, in practice, it makes more sense to assume *bounded adversaries*, i.e., attackers who have access to only a limited amount of computational resources. This brings us to the concept of *computational security.*

Definition 7 (Computational security)**.** *A block cipher E using an n_k-bit secret key is called computationally secure if there exist no attacks on E with a complexity less than the one of an exhaustive key search, i.e., 2^{n_k}.*

In the above definition, the *complexity* of an attack refers to the combination of the time (i.e, the work factor), memory (i.e., the storage requirement and the amount of memory accesses) and data (i.e., the type and amount of data) complexities required during the offline (i.e., precomputation) and online phase of the attack. It is crucial to consider all three different complexities since they all determine the actual cost, and hence the (im)practicality, of an attack. The generic black-box attack *exhaustive key search*, which is used as a reference in Def. 7, is described in Sect. 2.5.1.

However, as Def. 7 implicitly suggests, it is hard to evaluate (or even stronger, to prove) the (computational) security of a block cipher. Therefore it is common practice to consider a well-defined subset of all existing attacks and to make the proper design choices of the block cipher to resist this subset of attacks. For example, with regard to resistance against differential and linear cryptanalysis (discussed on p. 39), this often involves the provable lower bounds on the probabilities of differential/linear characteristics of certain components (e.g., S-boxes) and the differential/linear properties of a multiple round structure (e.g., the number of active S-boxes) of block ciphers.

At the end, the most convincing argument concerning the security of a block cipher relates to the concept of *ad hoc security*: a cipher is said to be ad hoc secure if no successful attacks (i.e., better than exhaustive key search) have been found by cryptanalysts for many years after the publication of the specification of the cipher. Making the design specification of a block cipher public falls under *Kerckhoffs' assumption*, which is discussed in the next section.

2.4.3 Kerckhoffs' Assumption

In 1883, Kerckhoffs [54] stated six principles that should be met by well designed military ciphers at that time. The second principle, which is now known as *Kerckhoffs' assumption* or *Kerckhoffs' principle*, is still of great importance nowadays and is quoted in the following:

> *"Il faut qu'il n'exige pas le secret, et qu'il puisse sans inconvénient tomber entre les mains de l'ennemi."* [54]

With the above quote, Kerckhoffs stated that a cryptographic algorithm should remain secure even if everything about the algorithm is known except for the secret key. In other words, the security of the algorithm should rely solely on the secrecy of the key and not on the secrecy of the specification of the algorithm itself. It should be emphasized that Kerckhoffs never made a statement about either 'publish the specification' or 'keep the specification confidential'. Therefore, it is best practice to always design a cryptographic algorithm (such as a block cipher) under the assumption that its specification is known to the attacker (e.g., if the attacker figures out how the algorithm works) and thus without relying on the secrecy of the design. After that, it remains the choice of the designer to either publish the specification or keep it confidential.

The former case (i.e., a public specification) might be interesting if a block cipher design lacks any security proofs such that it can gain trust through *ad hoc security* (Sect. 2.4.2). The first block cipher with a full public specification was Lucifer, the predecessor of DES. Now, also AES has a publicly known specification; moreover, given its simple algebraic structure, it was a target of many cryptanalysts during many years and will remain in the spotlights for years to come. The latter case (i.e., a confidential specification) creates an additional challenge that the attacker is facing, namely extracting the specification.

2.4.4 Cryptanalyst's Goal

As explained in Def. 7, a block cipher is called *computationally insecure* once a successful attack has been found with a complexity less than that of an exhaustive key search. In order to complement Def. 7, a 'successful attack' should be defined. Depending on the application in which the cipher is deployed, an attacker can be satisfied with various cryptanalytic results. A classification of the outcome of an attack [58, p. 7-8] is given below:

Total break: the full recovery of the n_k-bit secret key k;

Global deduction: the construction of an algorithm functionally equivalent to E_k (or D_k) without the knowledge of the secret key k;

Local deduction: the retrieval of $E_k(m)$ (or $D_k(c)$) for some plaintext m (or ciphertext c) without the knowledge of the secret key k;

Distinguishability: the ability to distinguish between (i) the permutation E_k specified by the block cipher E for a randomly chosen secret key k and (ii) a randomly chosen permutation on $\mathbb{F}_2^{n_b}$. Here, it is assumed that the distinguisher does not know the secret key k.

The order of the above classification is hierarchical, i.e., the successful execution of an attack ensures the successful execution of all subsequent attacks. Note that the inability to successfully execute the 'distinguishability attack' relates to the definition of ideal block ciphers (Def. 2).

Now, depending on the goal of the cryptanalyst, a successful attack does not necessarily imply the full recovery of the secret key. For example, as is discussed later in Sect. 3.4 in the context of white-box security, inverting the cryptographic algorithm (i.e., global deduction) can pose a serious threat without ever recovering the actual secret key. A detailed discussion of the above classification with respect to white-box cryptography (i.e., the objective of a white-box attacker) is given in Chapter. 3 (Sect. 3.4.1).

2.4.5 Attack Models

Attack models specify the capabilities of the attacker (i.e., the cryptanalyst) in order to attempt breaking a block cipher while achieving his goal, i.e., finding an attack with a complexity less than the one of exhaustive key search. Such attacks are also referred to as *shortcut attacks*. The existence of attack models formulating the hostile environment in which a block cipher will be deployed, is crucial when it comes down to designing a new block cipher or an implementation (hardware and/or software) of an existing block cipher with respect to assessing its security level.

Below, the three main attack models are discussed and a comparison (related to their evolution over time) is given at the end of this section.

Black-Box Model

The *black-box model* is the model assumed in traditional cryptography. In this attack model, the authorized end-users of the communication channel (i.e., the sender and the receiver(s)) are assumed to be trusted. Where this assumption was valid in the earlier years of cryptography, the wide range of applications in which cryptographic primitives are deployed nowadays clearly shows that the black-box model is inadequate.

The black-box model is the most conservative model in which the information available to the attacker is solely restricted to the input/output behavior of the block cipher, where the input corresponds to the plaintext and secret key, and the output corresponds to the ciphertext (in the case of encryption). According to which information (i.e., input and/or output) is available to the attacker, and to which operations (i.e., read and/or (adaptively) write) the attacker is allowed to perform on this information, a classification of black-box attack models is given below:

Ciphertext-only: the attacker has only read access to the ciphertext. This is considered to be the default and also weakest attack scenario such that failure to resist attacks within this model implies particularly insecure block cipher designs.

Known plaintext: the attacker has read access to plaintext/ciphertext pairs. Linear cryptanalysis (cf. Matsui [65]) belongs to this class of attack models.

Chosen plaintext: the attacker has write access to the plaintext and read access to the corresponding ciphertext. Choosing specific forms of plaintexts is a crucial requirement of differential cryptanalysis (cf. Biham and Shamir [8]).

Chosen ciphertext: the attacker has write access to the ciphertext and read access to the corresponding plaintext. This attack model requires that the attacker has (limited) access to the decryption routine.

Adaptively chosen plaintext/ciphertext: similar to the 'chosen variants' above, with the difference that the attacker is able to depend his choice on intermediate results obtained during the attack.

The above classification is not complete. First, every possible combination of the above mentioned classes results in a valid attack model. Second, as the secret key is part of the input of a block cipher, there exist other attack models such as *known key* and *chosen key* (cf. Knudsen and Rijmen [57]) that are of typical interest for block cipher based hash function designs where the key is either known to the attacker or to some extent under the attacker's control.

Grey-Box Model

In reality, cryptographic primitives such as block ciphers are always implemented either in hardware or software on physical devices. In the black-box model, it is assumed that these implementations behave as *ideal black boxes*, preventing any observation or tampering of internal operations/data. Hence, the capabilities of the attacker are restricted to observing the input/output behavior of the cryptographic primitives, which together with the knowledge of the (possibly) public specification often results in rather complex attacks with a high computational cost.

However, in practice, the assumption of ideal black boxes is unrealistic. This led to the introduction of a more realistic attack model, i.e., the *grey-box model*, that no longer assumes that the end-points of the communication channel are trusted (as was the case in the black-box model). In the grey-box model, (limited) access to the implementation belongs to the capabilities of the attacker, such that the information available to the attacker is, apart from the information available in the black-box model, further expanded by *implementation-specific information* that typically is (weakly) correlated to the cryptographic key. Attacks focusing on the implementation of cryptographic algorithms rather than on the algorithms themselves are referred to as *implementation attacks*, and can be categorized by the following two classes:

Intrusive nature: this relates to the fact whether the attacker tampers with the behavior of the system containing the implementation of the cryptographic primitive or not. Hence, the following distinction can be made: (i) *active attacks* that involve tampering by for example altering the environmental conditions (e.g., heat) and (ii) *passive attacks* that involve no tampering and are restricted to just observing.

Invasive nature: this relates to the fact whether the attacker tampers with the device of the cryptographic system in order to gain (direct) access to internal components or not. Depending on the degree of tampering, one can distinguish the following three attacks: (i) *invasive attacks* that involve direct access to the internal components of the cryptographic system, (ii) *semi-invasive attacks* that involve access to the system only through the authorized surface and (iii) *non-invasive attacks* that are limited to the externally available information (which is unintentionally leaked/emitted), i.e., the device remains unaltered.

In general, the subclass of passive non-invasive implementation attacks poses a serious threat to the security of cryptographic devices since they are undetectable and tend to be of low cost. This subclass is formed by the so-called *side-channel* *analysis (SCA) attacks*. The implementation-specific information exploited by SCA attacks is called the *side-channel information* and refers to additional information that is leaked/emitted out of a real-world implementation through unintended *side-channels* during its execution. Examples of side-channels are execution time, power consumption and electromagnetic radiation. SCA attacks were introduced by Kocher [59] in 1996.

According to how the obtained side-channel information (referred to as *side-channel trace(s)*) is analyzed, the following distinction can be made: simple and differential SCA attacks. In the case of *simple SCA attacks*, only one side-channel trace is measured and the attack relies on the correlation between the executed implementation-specific instructions and the side-channel output. In the case of *differential SCA attacks*, multiple side-channel traces are measured for different input data and the attack exploits the data-dependencies in the obtained side-channel traces based on statistical analysis. Additionally, a distinction can be made between *profiled* and *non-profiled SCA attacks*. Profiled SCA attacks contain an *a priori* profiling phase during which the attacker has full control over a training device that is equal or similar to the attacked physical cryptographic implementation (i.e., the target device). This allows the attacker to gather side-channel traces corresponding to known or chosen inputs and/or keys of the training device in the profiling phase, which later can be used during the actual SCA attack on the target device. An example of a profiled SCA attack is the *template attack*, introduced by Chari, Rao and Rohatgi [22] in 2002.

White-Box Model

The grey-box model as discussed above is the most severe attack model when it comes down to hardware implementations of cryptosystems, i.e., the attacker is allowed to tamper with the physical device in order to obtain some useful key-correlated side-channel information. However, with respect to software implementations, the model becomes significantly weaker as the attacker is only allowed to have access to the implementation through side-channels such as execution time (see the *cache timing attacks* discussed in Sect. 2.5.2). In practice, as indicated by the use case treated in Sect. 1.1, the extent to which the attacker has access to software implementations is often much more severe: i.e., in real world applications, software implementations are executed in an untrusted environment and hence are much more vulnerable than merely the unintentional leakage of side-channel information. As a consequence, in 2002, a new ‘realistic’ attack model was introduced by Chow, Eisen, Johnson and van Oorschot [24, 23] under the name of the *white-box attack context*.

Definition 8 (White-box attack context [23, p. 4]). *The white-box attack context (WBAC) captures the strongest capabilities of an attacker in the scenario of the execution of software implementations of cryptosystems in a hostile environment. The WBAC assumes that:*

1. *fully-privileged attack software shares a host with cryptographic software, having complete access to the implementation of algorithms;*
2. *dynamic execution (with instantiated cryptographic keys) can be observed;*
3. *internal details of cryptographic algorithms are both completely visible and alterable at will.*

To summarize, in the white-box attack context, also referred to as the *white-box model*, the end-users are considered to be untrusted as within the grey-box model, and moreover are assumed to have (i) full access to the cryptographic software implementation and (ii) full control over its execution platform. As a result, examples of what an attacker is allowed to perform on cryptographic software in the white-box model are the following:

- *static analysis* by means of disassemblers or decompilers: the attacker is allowed to reverse-engineer the software implementation and adaptively alter parts of it to his advantage, i.e., tampering with the software;
- *dynamic analysis* by means of debuggers with breakpoint functionality: the attacker can observe any intermediate result and alter it at will, i.e., injecting well-chosen faults. He is also allowed to start/stop executing the software at any given point;
- *search in memory* for cryptographic keys.

The list above is not complete, however, it contains the most common capabilities exploited by an attacker in the white-box model (i.e., a *white-box attacker*).

Within the field of software protection techniques, the white-box model is also known as the *malicious host attack model* (cf. Sander and Tschudin [94, 93]). Clearly, with respect to software implementations, the white-box model can be considered as a worst case attack model from the perspective of the designer. However, note that a white-box attacker is assumed to have only access to the cryptographic software, and hence not to the software generating the cryptographic software (also known as the white-box (WB) generator). If the attacker also has access to the WB generator, then he steps out of the white-box model; this is an example of an even more severe (maybe not practical) attacker.

Comparison Between Attack Models

The biggest change between attack models is determined by the lack of trust in the end-users, which takes place at the transition from the black box model to the grey-box/white-box model. While the mathematical strength of a cryptographic primitive is central to the black-box model, its implementation comes into play in the grey-box model (mainly hardware) and the white-box model (solely software). When going through all three different attack models, i.e., from black-box to grey-box to white-box, it is clear that the threat only increases in accordance to the increasing amount of information available to the attacker (Fig. 2.2). In fact, all models build upon each other in the order mentioned above, e.g., a white-box attacker complies to the grey-box and black-box models. As a consequence, there exists a certain duality concerning the security within the different attack models, where the black-box model is assumed to be the *weakest* attack model, and the white-box model to be the *strongest* attack model (here and below, the terms 'weaker', 'weakest', 'stronger' and 'strongest' are from the perspective of the attacker):

1. *Insecurity* within an attack model implies insecurity within all stronger attack models as well, but not necessarily in the weaker attack models;
2. *Security* within an attack model implies security within all weaker attack models as well, but not necessarily in the stronger attack models.

As an example, this thesis shows that in early 2014 there exists no secure white-box AES implementation in the academic literature, however, AES is still considered to be a black-box secure block cipher.

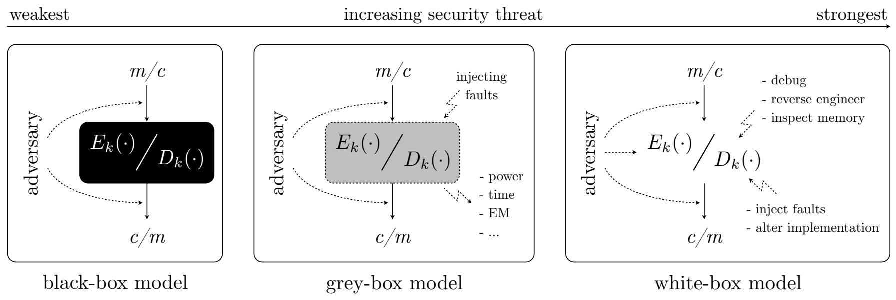


Figure 2.2: The evolution of the attack models.

The shift in attack models is largely caused by the era of the computer and communication devices, i.e., by the fact that cryptography-based applications on physically (insecure) devices have been brought closer to a very broad audience of end-users, hence also to potentially hostile end-users. Take for example the DRM setting where the end-user is considered to be the opponent. Interestingly, this shift is reflected in the times at which the various attack models have been introduced: the black-box model roughly since the beginning of cryptology, the grey-box model since 1996 and the white-box model since 2002. Note that these introduction times refer to their first appearance in the literature, and that it is likely that the various attack models already existed in practice in advance.

2.4.6 The Unbounded White-Box Attacker

Recall from Sect. 2.4.2 that the overall complexity of an attack could be divided into time, memory and data complexities. In the black-box and grey-box model, all three different types of complexity play a crucial role in determining the overall complexity (i.e., the practicality) of a black-box or grey-box attack. In particular, concerning the data complexity given by the amount and type of data (i.e., the number of known or (adaptively) chosen plaintexts/ciphertexts) required for the attack, the black-box or grey-box attacker is typically bounded by the number of queries he can make in order to collect the necessary data due to limited access to (the implementation of) the cryptographic algorithm. Often this results in a bottleneck with respect to the practicality of an attack. As is discussed in Sect. 2.5.1 (see the example on p. 38), even though the linear and differential attacks on DES show its computational insecurity, the attacks are still considered to be theoretic since the data complexity is impractical.

However, since a white-box attacker is in possession and in full control of the cryptographic software implementation, the number of queries he can make to the implementation is unbounded (though in practice this number is still limited by time and available storage). Similar reasoning can be found in the case of profiled SCA attacks, where the grey-box attacker is assumed to be in full control of a training device during the profiling phase. Furthermore, the white-box attacker is not solely restricted to the input/output of the cryptographic algorithm (i.e., the plaintext and ciphertext), but he can observe, alter or inject any intermediate result in the cryptographic software implementation as well.

Complexity of a white-box attack. Because of the above reasoning, the way how the overall complexity of a black-box or grey-box attack is determined becomes less relevant in the context of white-box attacks. First, due to the fact that the white-box attacker can compute at will and is not bounded in the number of queries he can make, data complexity is not meaningful anymore. Second, as will become clear in Part II of this thesis, white-box attacks tend to have negligible memory complexities. As a result, the overall complexity of a white-box attack is determined by its time complexity, which is referred to as the *work factor* in the remainder of this thesis.

2.5 Cryptanalytic Techniques

This section describes the most common cryptanalytic techniques used within the three different attack models discussed in Sect. 2.4.5.

2.5.1 Black-Box Cryptanalysis

Two different classes of black-box attacks can be distinguished: (i) the *generic attacks* that only exploit the core properties of the cipher such as the block and key size, and (ii) the *non-generic attacks* that exploit additionally the internal structure/specification of the block cipher. As mentioned earlier, the former class of attacks can be mounted against ideal block ciphers. Below, a selection of the most common black-box attacks is highlighted.

Generic Attacks

Exhaustive key search. The exhaustive key search attack, also known as the *brute force attack*, simply consists of testing all possible values of the secret key. This 'key test' can be performed in the ciphertext-only setting which relies on redundancy (i.e., a biased statistical distribution) in the plaintext in order to identify the correct key. However, in order to avoid the reliance on this plaintext redundancy, a known plaintext/ciphertext pair in the known plaintext setting is preferred (from the perspective of the attacker). The 'time' complexity (i.e., the work factor) of the brute-force attack to retrieve an n_k-bit secret key equals about 2^{n_k} encryption operations, where the computation cost of the key scheduling algorithm (if applicable) should be taken into account. If more than one key is given as output for the original plaintext/ciphertext pair, the attack needs to be repeated for additional plaintext/ciphertext pairs. Optimizations are possible through the parallelization of the exhaustive key search.

Dictionary attack. Related to the exhaustive key search attack is the *dictionary attack*. Where the former is executed in the *time* domain, the latter is executed in the *memory* domain. The dictionary attack is a chosen-plaintext attack: in order to identify the correct key, the ciphertext of a chosen plaintext is matched against a lookup table that stores the encryption of the chosen plaintext under all possible values of the secret key. When ignoring the cost of the offline phase (i.e., building the lookup table), for an n_b-bit block cipher using an n_k-bit secret key, the memory requirement is given by $2^{n_k} \cdot n_b$ bits, whereas the 'time' complexity equals only one single table lookup.

Time-memory trade-off attack. Between the above two extreme attacks, i.e., the 'time-based' brute force attack and the 'memory-based' dictionary attack, lies the *time-memory trade-off attack* proposed by Hellman [49]. The main idea behind this chosen-plaintext attack is to break down the entire key space into a collection of smaller sets and to perform the exhaustive search over these sets. Again, when ignoring the cost of the offline phase, the time-memory trade-off attack requires $2^{n_k \cdot 2/3}$ encryption operations and $2^{n_k \cdot 2/3+1} \cdot n_k$ bits of memory. In fact, many different time-memory trade-offs are possible.

Code book attack. Some generic attacks such as the *code book attack* exploit the block size of the cipher instead of the key size. A *code book* corresponds to a table storing the ciphertexts for all 2^{n_b} possible plaintexts encrypted using the same key. Hence, given such a table representing the code book, an attacker can easily decrypt any ciphertext by means of one single table lookup. Hence, for an n_b-bit block cipher, the 'time' complexity is only one single table lookup, whereas the memory requirement is given by $2^{n_b} \cdot n_b$ bits.

Example. As the block cipher DES [68] is a 64-bit iterated non-key-alternating Feistel block cipher using a 56-bit secret key, the exhaustive key search attack requires 2^{56} encryption operations. Although originally generic attacks were not considered as a serious threat, they gained significant interest with respect to DES due to (i) the increasingly growth of computation power and (ii) the relatively short key size of 56 bits. In 1998, the Electronic Frontier Foundation (EFF) [41] built a machine, called the Deep Crack, that performs a successful brute force attack on DES in about 56 hours. In 2009, SciEngines [96] introduced the RIVYERA computing-architecture (the successor of the COPACOBANA machine) that is able to exhaustively find a DES key in less than a single day.

Countermeasure against generic attacks. Because of the independence of the design specification of the block cipher, the single countermeasure is to increase the key space and message space. The length (measured in bits) of the secret key depends on the security level demanded by the containing application. When consulting the yearly ECRYPT II report of 2011-2012 [67, Table 7.4], a length of 80 or 128 bits offers sufficient security against a wide range of attackers. If long-term security is required, a key length of 256 bits is advised.

Non-generic Attacks

At this point, it is assumed that both the key and block size have been carefully chosen such that generic attacks are far from practical. Hence the next step is to exploit specific design characteristics (i.e., the internal structure) of the block cipher. Therefore, such attacks, classified as *non-generic*, do not apply straightforward to any block cipher. An exception to this rule is the so-called *meet-in-the-middle (MITM) attack*, which is a generic time-memory trade-off attack and is typically applied to block ciphers comprising multiple (double or triple) encryption using independent keys.

Typically, non-generic attacks comprise the following two phases:

Phase I concerns the construction of an *s-round distinguisher*, i.e., a pattern over s rounds that holds true with a relatively high probability. As mentioned before in Sect. 2.2, the designer's goal is to ensure that the highly structured block cipher E approximates an ideal block cipher (Def. 2), i.e., the key-dependent permutation $E_k(\cdot)$ for a randomly chosen key k should be indistinguishable from a random permutation. Conversely, the attacker's goal is to identify an s-round distinguisher that enables him to distinguish the s-round reduced version of $E_k(\cdot)$ from a random permutation.

Phase II concerns the recovery of round key(s). An s-round distinguisher allows an attacker to attack an $s + 1$ or $s + 2$-round (reduced) version of the key-instantiated block cipher by either appending a first round or final round or both before and/or after the distinguisher. Next, given a sufficient amount of plaintext/ciphertext pairs, the attacker is able to identify (parts of) the first/final round key(s) as follows: (i) guess the involved round key bits, (ii) (partially) encrypt (or decrypt) the plaintexts (or corresponding ciphertexts), and (iii) verify whether the s-round distinguisher holds true for that particular guess. Once (parts) of the round key(s) have been recovered, the attacker may obtain the actual secret key.

For an R-round block cipher, one speaks of a *full-round* attack if $s + 1 = R$ (or $s + 2 = R$). Otherwise, it is called a *reduced-round* attack.

If a full-round non-generic attack improves the exhaustive key search attack, i.e., it has a lower complexity, one speaks about a *shortcut attack*. The existence of shortcut attacks implies the computational insecurity of a block cipher (Def. 7). However, a distinction should be made between *practical* and *theoretical* shortcut attacks. In line with the desirable key size discussed above, the upper bound on the complexity of shortcut attacks is given by 2^{80} or 2^{128}, depending on the amount of computational resources available to the addressed attacker. Very loosely speaking, this can be considered as the boundary between practical and theoretical.

Differential cryptanalysis. Differential cryptanalysis is a chosen-plaintext attack, and was introduced by Biham and Shamir [8] in 1990. As turned out later by analyzing the design of DES, the attack was already identified by the designers of DES in the mid 1970s. Over the years, differential cryptanalysis has been shown to be a very powerful cryptanalytic technique for symmetric-key cryptographic primitives.

In a nutshell, differential cryptanalysis keeps track of the probabilistic behavior of a difference through multiple rounds of a block cipher. Assuming that the key addition operation of the block cipher is given by the bitwise addition modulo two (or XOR operation), which typically is the case for many common block ciphers, then a *difference* between two bit strings X and X' of equal length is defined as $\Delta X = \Delta(X, X') = X \oplus X'$.

Now, the s-round distinguisher exploited by differential cryptanalysis is an s-round differential (introduced by Lai, Massey and Murphy in [61] and defined below) that holds true with a high probability, i.e., a probability significantly higher than the random difference probability $1/2^n$ where n typically denotes the block size.

Definition 9 (Differential [61])**.** *An s-round differential is a pair of differences (α, β), where the input difference α is the chosen plaintext difference so that $\Delta P = \Delta(P, P') = \alpha$, and where the output difference β is the expected difference between the partially encrypted plaintexts P, P' after s rounds, denoted by C_s, C'_s respectively, so that $\Delta C_s = \Delta(C_s, C'_s) = \beta$. The probability of the differential is given by the conditional probability $Pr\big(\Delta C_s = \beta \mid \Delta P = \alpha\big)$, which is taken over the entire plaintext space and key space.*

Typically, the complexity of a (successful) differential attack is expressed in the number of chosen plaintext pairs (P, P') with $\Delta(P, P') = \alpha$ required to identify the correct key. This number is inversely proportional to the probability of the differential (α, β). As a consequence, the higher this probability and thus the better the differential, the lower the complexity becomes. Calculating the probability of an s-round differential is likely to be a complex task, however, a good approximation is given by Lai, Massey and Murphy in [61] under the following two assumptions: (i) the block cipher is a Markov cipher (Def. 10) with independent and uniformly random round keys, and (ii) the hypothesis of stochastic equivalence (Def. 11).

Definition 10 (Markov cipher [61])**.** *An iterated cipher with round function $C_r = E_{k^{(r)}}(C_{r-1})$ is a Markov cipher, with respect to the defined difference, if*

$$\Pr\left(\Delta C_r = \beta \mid \Delta C_{r-1} = \alpha, C_{r-1} = \gamma\right)$$

is independent of γ for all α and β when the round key $k^{(r)}$ is chosen uniformly at random, where ΔC_{r-1} and ΔC_r denote the input and output differences of the round function.

Definition 11 (Hypothesis of stochastic equivalence [61])**.** *For virtually all high probability s-round differentials (α, β)*

$$\Pr_{\mathbb{P}}\left(\Delta C_s = \beta \mid \Delta P = \alpha, k = k'\right) \approx \Pr_{\mathbb{P},\mathbb{K}}\left(\Delta C_s = \beta \mid \Delta P = \alpha\right)$$

holds for a substantial fraction of the key values k', where $\mathbb{P}$ and $\mathbb{K}$ denote the plaintext space and key space, respectively.

Although the above hypothesis is a plausible assumption for many ciphers, there also exist ciphers for which the hypothesis does not seem to hold. Take for example the so-called *plateau characteristics* introduced by Daemen and Rijmen [32] where the probability of differential trails is key-dependent in a highly structured way, i.e., it can only take two different values (one of which is zero). Such plateau characteristics should be taken into account when analyzing the resistance against differential cryptanalysis.

Linear cryptanalysis. Linear cryptanalysis is a known-plaintext attack. It was introduced by Matsui [65] in 1993, although a precursor to linear cryptanalysis was already proposed by Tardy-Corfdir and Gilbert [100] in 1991. When compared to differential cryptanalysis, linear cryptanalysis tends to be a less powerful and versatile cryptanalytic technique. However, observe that there is a strong analogy between the description of differential and linear cryptanalysis.

While differential cryptanalysis keeps track of the probabilistic behavior of a difference through a reduced s-round variant of a block cipher, linear cryptanalysis focuses on the probability of linear relations between plaintext bits and partially encrypted (after s rounds) plaintext bits. Linear relations are described by means of *linear masks* (or simply *masks*) which define a specific linear combination of bits; they make use of the scalar product over $\mathbb{F}_2$ (denoted by $\bullet$). With regard to the probability p of linear relations, two quantities are introduced: (i) the *bias* ϵ defined as $\epsilon = p - \frac{1}{2}$, and (ii) the *correlation* c defined as $c = 2p - 1 = 2\epsilon$. Typically, the absolute value or magnitude of the bias or correlation are considered, with the objective to have $0 < |\epsilon| \leq \frac{1}{2}$ or $0 < |c| \leq 1$ with $|\epsilon|$ or $|c|$ as large as possible.

Now, the s-round distinguisher exploited by linear cryptanalysis is an s-round linear hull (introduced by Nyberg [81] and defined below) with a significantly high bias $|\epsilon|$ or correlation $|c|$.

Definition 12 (Linear hull [81]). *An s-round linear hull is a pair of masks (α, β), where the input mask α defines a specific linear combination of the plaintext bits (i.e., $\alpha \bullet P$), and where the output mask β defines a specific linear combination of the bits of the partially encrypted plaintext C_s after s rounds (i.e., $\beta \bullet C_s$). The probability p of the linear hull is given by $Pr\big(\alpha \bullet P = \beta \bullet C_s\big)$, which is taken over the entire plaintext space and key space. The corresponding bias and correlation are given by $\epsilon = p - \frac{1}{2}$ and $c = 2p - 1$, respectively.*

Typically, the complexity of a (successful) linear attack is expressed in the number of known plaintext/ciphertext pairs, which is inversely proportional to the square of the bias (or correlation) of the linear hull (α, β). Observe that a linear hull with a higher bias (or correlation) is desirable as it lowers the complexity. Under the same two assumptions as mentioned in the case of differential cryptanalysis, i.e., (i) Markov cipher with independent and uniformly random round keys and (ii) the hypothesis of stochastic equivalence, the bias (or correlation) of a linear hull can be approximated using the *piling-up lemma.*

Example. In 1992, Biham and Shamir presented a differential cryptanalysis [9] that could break the full 16-round DES cipher with a total of 2^{47} chosen plaintext pairs and a work factor of 2^{47}. In 1994, Matsui [66] was able to break DES using linear cryptanalysis with a total of 2^{43} known plaintext/ciphertext pairs and a work factor of 2^{43}. Although both attacks show an improvement over an exhaustive key search when comparing their corresponding work factors, the attacks are still considered to be theoretical since it is impractical to obtain either 2^{47} chosen plaintext pairs or 2^{43} known plaintext/ciphertext pairs. Therefore, in practice, exhaustive key search is still the best attack on DES, especially when parallelizing the key search (e.g., RIVYERA [96]).

Countermeasure against differential & linear cryptanalysis. The goal of the designer of a block cipher is to ensure that there exist no high-probability differentials or no high-bias linear hulls. Suppose that the non-affine (with respect to the XOR operation) layer of an iterated SPN block cipher is given by the parallel execution of S-boxes. Next, an S-box is called *active* if it has a non-zero input difference or a non-zero output mask, otherwise it is called *inactive.* Since the affine layer, the key addition layer and the inactive S-boxes either all propagate a difference with probability 1 or all contain input-output linear relations with bias $|\epsilon| = \frac{1}{2}$, it is clear that only the active S-boxes influence the probability of a differential or the bias of a linear hull. As is captured by the *wide trail design strategy* [30], invented by Daemen and Rijmen and used in the design of Square [27] and AES [69], the following two factors play a crucial role in designing block ciphers resistant to differential and linear cryptanalysis:

1. minimize (a) the maximum difference propagation probability and (b) the maximum bias of input-output masks of the S-boxes;
2. maximize the minimal number of active S-boxes involved in any possible differential or linear hull. Technically speaking, this involves trails.

The first item can be achieved by carefully designing/choosing the S-boxes. The second item can be achieved by designing linear/affine layers with very good diffusion properties such that the output of an active S-box in round r affects as many S-boxes in round $r+1$ as possible. Closely related to this is the concept of *full diffusion*, i.e., the case when the output of an active S-box in round r makes all S-boxes in round $r+s$ active; for example, in the case of AES, full diffusion is achieved after two rounds ($s = 2$).

Multiset attacks. The fundamentals of the multiset attack were initiated by the *Square attack* (described below), proposed by Daemen, Knudsen and Rijmen [27]. Multiset attacks are chosen-plaintext attacks; they are at some abstract level comparable with differential cryptanalysis: i.e., instead of observing the probabilistic behavior of a chosen difference between a pair of plaintexts, one observes the deterministic behavior of a chosen set of plaintexts consisting of a concatenation of *multisets* each with their own property. As a result, the s-round distinguisher exploited by a multiset attack has a probability of one. Furthermore, multiset attacks exploit the overall structure of a reduced round version of a block cipher (i.e., how the different components mutually interconnect), and are independent of specific characteristics of the components as is the case for differential and linear cryptanalysis. Mainly, multiset attacks apply to highly structured block ciphers with respect to m-bit words: e.g., the design of AES is highly byte-oriented ($m = 8$).

At its core, a multiset attack consists of two phases:

Phase I**:** decompose the plaintext blocks into a concatenation of m-bit words that are aligned with the internal structure of the block cipher. Next, assign to each m-bit word of the plaintext a multiset of m-bit values with a specific property. A multiset is a set where each value can appear multiple times (expressed as its multiplicity). Some multiset properties are the following: C (*constant* – only one single value with arbitrary multiplicity), P (*permutation* – all possible 2^m distinct values with multiplicity 1), E (*even* – each value has an even multiplicity) and B (*balanced* – the XOR of all values (taking into account their multiplicity) equals 0). Observe that some properties automatically imply others, but not vice versa (e.g., $P, E \to B$).

Phase II**:** build a deterministic s-round distinguisher as follows: given the set of chosen plaintexts as a composition of multisets with mixed properties (carefully chosen by the attacker), partially encrypt this set through the reduced s-round version and keep track of the transformation of the multisets induced by the internal structure of the various rounds. These transformations follow certain *multiset propagation rules*, which are listed in [15, Lemma 1-2].

For a detailed description of the multiset attack, refer to Biryukov and Shamir [15]. Below, two applications of a multiset attack are highlighted, where the Square attack can be considered as the initial multiset attack (as mentioned above).

Example 1 – Square attack [27]. As its name suggests, the attack was discovered during the design of the Square [27] block cipher. Since AES is a successor of Square, and thus inherited many properties of Square, the Square attack also applies to AES. The 3-round multiset distinguisher depicted in Fig. 2.3 allows an attacker to break 6 out of 10 rounds of AES-128 with a work factor of 2^{72} encryption operations. Later, Ferguson et al. [43] reduced the work factor to only 2^{44} by identifying a very closely related 4-round multiset distinguisher. In both cases, the distinguishers are key-independent and exploit the bijectiveness of the AES S-box and the byte-oriented diffusion properties of `ShiftRows` and `MixColumns`.

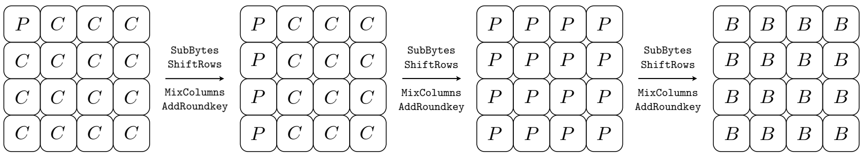


Figure 2.3: The deterministic 3-round distinguisher (with respect to AES) based on the multiset properties P (permutation), C (constant) and B (balanced).

In [23], Chow et al. use the first step of Ferguson et al.'s extended 4-round Square distinguisher to point out weaknesses in the design of a white-box AES implementation. This is discussed in Chapter 3 (Sect. 3.5.1).

Example 2 – Structural cryptanalysis of SASAS [15]. The multiset attack presented by Biryukov and Shamir in [15] successfully attacks an n-bit bijective five-layered structure $S_3A_2S_2A_1S_1$, where each S_i $(i = 1, 2, 3)$ denotes a layer of k non-linear m-bit bijective S-boxes in parallel with $n = k \cdot m$ and each A_i $(i = 1, 2)$ denotes an n-bit invertible affine transformation over $\mathbb{F}_2$. Apart from the knowledge of this general structure, it is assumed that all components are key-dependent and hence are kept secret, which led to the name of *structural cryptanalysis*. The attack relies on a 4-layer multiset distinguisher.

An interesting fact about this attack is that it finds equivalent representations of all involved components in the 5-layered structure, or in other words *equivalent keys*, that yield the same plaintext to ciphertext mapping as the original structure. Hence the attack succeeds without obtaining the actual key-dependent specifications of the S-boxes and the affine transformations. As described in Chapter 6, a similar result is obtained in the cryptanalysis of a white-box AES implementation, i.e., a set of equivalent keys is retrieved that yield functionally equivalent implementations.

Algebraic attacks. Within the field of symmetric-key cryptography, and in particular block ciphers, algebraic cryptanalysis is a fairly recent black-box cryptanalytic technique; the work by Courtois and Pieprzyk [25] drew attention to the use of algebraic attacks. The main idea is to represent a block cipher as an overdefined system of multivariate non-linear algebraic equations in the bits (or m-bit words) of the plaintext, ciphertext and secret key, where the latter act as the unknowns. Solving this system with potentially a relative small amount of plaintext/ciphertext pairs would eventually yield the secret key. However, algebraic cryptanalysis has not yet been proven as a successful or powerful technique. The main obstacle is that the algebraic equations tend to be of a high algebraic degree introduced by the non-linear components of the cipher and/or the combination of operations over different fields (e.g., the fields $\mathbb{F}_2$ and $\mathbb{F}_{256}$ for AES).

Due to the algebra-friendly design of AES, attempts have been made to represent AES by a structured and sparse overdefined system of multivariate quadratic equations over either $\mathbb{F}_2$ (see Courtois and Pieprzyk [25]) or $\mathbb{F}_{256}$ (see Murphy and Robshaw [79]). Working over $\mathbb{F}_{256}$ is tempting since all operations of AES were designed with the finite field $\mathbb{F}_{256}$ in mind, except for the $\mathbb{F}_2$-linear operation involved in the AES S-box in order to preclude the algebraic simplicity. In a different attempt by Ferguson, Schroeppel and Whiting [44], AES is described by one big equation over $\mathbb{F}_{256}$ in a form similar to that of 'continued fractions'.

Another interesting approach of algebraic cryptanalysis is the investigation of *dual ciphers*, which were introduced by Barkan and Biham in [3].

Definition 13 (Dual ciphers [3]). *Two ciphers E and E^{Δ} are called dual ciphers if they are isomorphic, i.e., if there exist fixed invertible transformations f, g and h such that*

$$\forall m, k \qquad f\big(E_k(m)\big) = E^{\Delta}_{g(k)}\big(h(m)\big) \ ,$$

where m and k denote the plaintext block and the secret key, respectively.

The main motivations for considering dual ciphers are (i) to improve cryptanalysis of the cipher, i.e., the dual ciphers may exhibit interesting linear/differential characteristics, and (ii) to improve implementations of the cipher, i.e., to increase its performance and/or to reduce its memory footprint. In [3], Barkan et al. identified a set of 240 dual AES ciphers, which is listed in *The Book of Rijndaels* [2]. In [14], Biryukov, De Cannière, Braeken and Preneel extend this set to a total of 61 200 dual AES ciphers based on the affine self-equivalences of the AES S-box. In Chapter 4, we present an efficient cryptanalysis of a dual-cipher-based white-box AES implementation.

Up to now, the above mentioned approaches to algebraically cryptanalyze the full AES have not formed any threat to the security of AES. The main reason is that the algebraic equations become quite complex after a sufficient number of rounds. However, as is discussed later in Chapters 3-4, algebraic cryptanalysis has been proven successful in the white-box setting by efficiently attacking a white-box AES implementation. The advantage an attacker has in the white-box attack context is the ability to access encoded reduced round variants (such as a single isolated round) of a block cipher, which significantly simplifies the algebraic equations.

2.5.2 Grey-Box Cryptanalysis

When introducing the *grey-box attack model* in Sect. 2.4.5, one can notice that a cryptographic primitive can be approached from two different points of view: (i) it can be seen as an abstract mathematical model, i.e., a key-dependent function transforming plaintexts into ciphertexts, and (ii) it is implemented in either hardware or software within a physical device executing the cryptographic primitive. The black-box cryptanalytic techniques discussed in the previous section assume the first point of view, while implementation attacks (also known as *grey-box attacks*) take the second point of view as a starting point. Grey-box cryptanalytic techniques exploit physical implementation-specific characteristics in order to break the cryptographic primitive. Although implementation attacks are not as general as black-box attacks as they rely on physical aspects specific to the implementation, they are considered to be a serious threat and are most often very powerful attacks.

Within the field of grey-box cryptanalysis, the first scientific publication of an implementation attack appeared in 1996 by Kocher [59]; he proposed a side-channel analysis attack called the *timing attack*. Below, a brief overview is given of some of the foundational grey-box cryptanalytic techniques.

Timing analysis. Timing attacks belong to the class of passive non-invasive attacks. They were introduced by Kocher [59] in 1996, where he considered the execution time of particular public-key cryptographic algorithms (e.g., RSA [90]). Due to performance optimizations, computations often occur in the form of conditional statements and hence execute in non-constant time. If these computations (and more specifically the branch selections) are key-dependent (e.g., the conditional repeated square-and-multiply implementation for modular exponentiation), variations in the overall execution time may leak key information which can result in key recovery. This is referred to as *branch timing attacks*.

In [59], Kocher also discussed the application of timing attacks with respect to symmetric-key cryptographic algorithms such as block ciphers. Although conditional statements typically do not occur, lookups in tables stored in memory can cause time variations due to cache/RAM hits, i.e., cache and RAM have different latencies with respect to memory access. This leaks information about which table lookups have been performed, which may be correlated with sensitive key information. This is referred to as *cache timing attacks*. As an example, cache-based software side-channel attacks (e.g., by Bernstein [6] or by Osvik, Shamir and Tromer [83]) have been presented against the lookup-table-based software implementation of AES (Sect. 2.3.2). In [5], Benadjila, Billet and Francfort show that countermeasures against such attacks are in fact closely related to lookup-table-based white-box AES implementations.

Power analysis. Another type of passive non-invasive attacks are the power analysis attacks discovered by Kocher, Jaffe and Jun [60] in 1999. A distinction is made between *simple power analysis (SPA)* and *differential power analysis (DPA) attacks*. SPA attacks make use of only one single power consumption trace; they exploit key-dependent operations each with different power consumption traces according to the key value. Typically, SPA is a successful technique in attacking public-key cryptographic algorithms where the key-dependent operations involve conditional statements (e.g., RSA – see above). On the other hand, DPA attacks are based on the statistical analysis of a large set of power consumption traces accompanied by the corresponding plaintexts/ciphertexts. The success of a DPA attack relies on the dependency of the power traces on part of an internal state of the cryptographic algorithm. Where SPA attacks depend strongly on the knowledge of the implementation, this is not the case for DPA attacks which makes them more powerful.

At the second advanced encryption standard (AES) candidate conference in March 1999, many results appeared concerning the vulnerabilities and possible countermeasures of smartcard implementations of the AES candidates (e.g., by Chari et al. [21] and by Daemen and Rijmen [29]) and their key scheduling algorithm (see Biham and Shamir [11]) against timing and power analysis attacks. Also after the selection of Rijndael as the new AES in 2001, the research on secure smartcard implementations of AES remained very active.

Closely related to the power analysis attacks are the *electromagnetic analysis (EMA) attacks*, proposed by Quisquater and Samyde [87] and Gandolfi, Mourtel and Olivier [45] in 2001. These attacks use *electromagnetic emanation* as a side-channel and hence can be helpful if *power* is not available as a side-channel.

Fault analysis. In 1997, Boneh, DeMillo and Lipton [18] announced the fault analysis attack, that exploits faults/errors occurring during the computation of particular public-key cryptographic primitives (e.g., RSA). Such faults, referred to as *computational faults*, are often induced by intentionally manipulating the environment of the hardware/software implementation in a semi-invasive way and can be categorized as follows: (i) *transient* faults by for example altering the power supply voltage or the working frequency, and (ii) *persistent* faults by for example cutting a wire or destroying a memory cell/register by means of a laser beam. Because of the manipulation of the behavior of the implementation, fault analysis attacks are classified as *active* attacks.

Biham and Shamir [10] introduced a fault analysis attack that, in contrast to Boneh et al.'s attack, is applicable to a wide range of symmetric-key cryptographic algorithms (such as block ciphers), namely the *differential fault analysis (DFA) attack*. In its essence, the attack analyzes the difference between outputs obtained through both the normal and abnormal computation of the block cipher, given the same input in both cases. The abnormal behavior is caused by injecting computational faults (at bit or byte level) in one of the last rounds of the cipher. In [10], the authors emphasize the strength of a DFA attack by extracting the full DES key with only very few pairs of (correct,faulty) ciphertexts. While Biham and Shamir [10] focused on Feistel structured block ciphers, Piret and Quisquater [85] showed that DFA attacks can also be successful against SPN block ciphers, with a practical application to AES. The feasibility of DFA attacks (or fault analysis attacks in general) in practice depends on how close the assumed *fault model* resembles reality, where a fault model captures the capabilities of an attacker with respect to (the accuracy of) injecting faults.

2.5.3 White-Box Cryptanalysis

This thesis covers cryptanalytic techniques within the white-box attack model. The entire Part II is dedicated to both the design and analysis of white-box implementations. Before diving into this topic, this section emphasizes the importance of constructing secure software implementations of a cryptosystem when employed in the white-box model. This is done by discussing two powerful and efficient attacks against naive software implementations of cryptographic algorithms with an embedded cryptographic key, where *naive implementations* refer to those used in the traditional black-box model (i.e., under the assumption of trusted end-users). Take the standard software implementation of AES (see Sect. 2.3.2) as an example of a naive implementation. Both attacks also clearly indicate the strength of the capabilities of a white-box attacker, which is typically underestimated.

Entropy attack. In 1999, Shamir and van Someren [97] considered the problem of efficiently locating cryptographic keys in large quantities of data stored in memory. Their motivation was the so-called *lunchtime attack*, where the attacker gains access to the memory (e.g., the hard disk) of the computer while the authorized user is away for lunch. This memory may contain the software implementation of a cryptographic algorithm with an embedded secret key. The attack for locating the embedded key relies on the (significant) difference in *entropy*[2] between the secret key and the structured design of the cryptographic algorithm:

Key = high entropy: the strength of cryptographic keys relies on their unpredictability, hence it is assumed that they are chosen randomly out of the entire key space. As a consequence, the section of the memory containing the secret key has high entropy.

Algorithm = low entropy: the section of the memory containing the binary of the cryptographic algorithm consists of a set of instructions implementing the well structured algorithm, hence it is likely to have significant less entropy than the section storing the cryptographic key.

Visually, sections with high entropy look *noisy*, while sections with low entropy look *structured*. An example is given in Fig. 2.4 which depicts a binary representation of a memory storing (part of) the implementation of an RSA signature verification algorithm. By visually inspecting Fig. 2.4, one can easily

[2]In 1948, Shannon [98] introduced the concept *entropy* in the information theoretic world. The *Shannon entropy* is used as a metric to measure the uncertainty or unpredictability of data. Entropy is said to increase with a higher degree of uncertainty of the data.

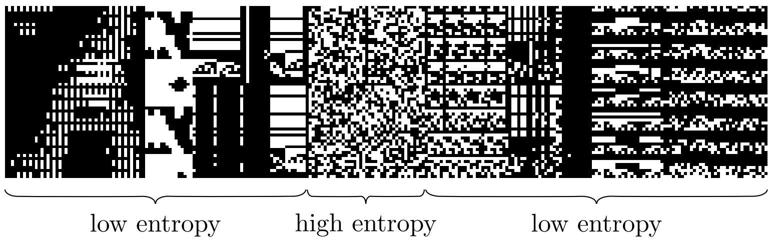


Figure 2.4: Visual identification of the location of cryptographic keys stored in memory based on the difference in entropy (source: [97]).

locate the RSA signature verification key. A more generic approach is the application of a *sliding window* over the memory, where in each iteration the entropy is calculated for the current window. Windows with unusually high entropy most likely indicate the location of the key.

Interestingly, Shamir and van Someren understood already back in 1999 the importance and difficulty of secure software implementations executed in an untrusted environment:

> *"If computer programs must be operated in an hostile environment, they need to have some form of protection. While it is relatively easy to build tamper resistant hardware, it is much harder to protect computer software."* [97]

Now, someone may wonder: *"Does the entropy/lunchtime attack really pose a threat in the real world?"*. Sadly, the answer is 'yes'. This has been shown in 2008 by Halderman et al. [48] who presented the so-called *cold boot attacks* on powered - though locked - computers with the objective to extract hard disk encryption keys out of DRAM memory images. They show that, although volatile memory (such as DRAM) is assumed to lose its stored information rapidly if the power is removed, the DRAM remanence can be increased dramatically by severely cooling the DRAM modules. Hence by rebooting (or cut the power + boot) a locked computer while keeping the DRAM modules cooled, they were able to obtain the DRAM memory images and search for the encryption keys in it. Their method for locating keys in memory images differs from the entropy method proposed by Shamir and van Someren; for example, in the case of a secret symmetric key k, they look for sequences in the memory closely related to the expanded key K (obtained from k through the key schedule) and exploit properties of the key schedule to correct for bit errors. This method can be justified since often the expanded key K instead of the secret key k is stored for performance reasons.

Countermeasures against the entropy attack. In [97], Shamir and van Someren proposed countermeasures in order to resist the entropy attack. Their idea was to ensure that the *entropy density* is uniformly distributed over the entire implementation, such that monitoring the difference in entropy does not yield any information about the location of the key. This can be achieved by the following two complementary techniques:

Lower the entropy density of the key: a trivial solution is to lower the entropy of the cryptographic key. However, this results in less random keys which triggers even worse security threats. A non-trivial solution is to spread the key over the entire implementation such that its otherwise local high entropy causes the global entropy density to increase. Shamir et al. already pointed out that the best way to achieve this goal is to fix the key value and to implement a key-instantiated cryptographic algorithm in an optimized way. Moreover, they made the following observation:

> *"Furthermore if the optimization process is thorough, it will likely be extremely hard to change the key without replacing the entire section of code which uses that key."* [97]

As is discussed in Chapter 7, this observation is very closely related to the fact why constructing *dynamic-key* white-box implementations seems an even more difficult task to achieve than constructing *fixed-key* white-box implementations.

Increase the entropy of the algorithm: one way to achieve this goal is to introduce randomness within the structure of the algorithm while preserving its overall functionality.

The two countermeasures above should not be seen separately, but as a coherent whole. Remarkably, they form the basis of the techniques presented by Chow, Eisen, Johnson and van Oorschot [24, 23] in order to obtain software implementations secure within the white-box attack context.

S-box blanking attack. In 2006, Kerins and Kursawe [55] present a powerful attack against simple software implementations of block ciphers. In contrast to the entropy attack that exploits the high entropy property of cryptographic keys, Kerins et al.'s attack focuses on the low entropy (i.e., structured) part of the implementation. In [55], the authors considered both Feistel and SPN block ciphers, though in the following only the SPN ciphers are discussed. In order for the attack to be successful, both the SPN block cipher (Def. 5) as well as its software implementation need to satisfy the following requirements:

- regardless of the boundaries of the round functions, the last three operations are (i) the confusion layer S consisting of the parallel execution of S-boxes, (ii) the diffusion layer D comprising a linear function (or the identity function if it is not applicable), and (iii) the final round key $k^{(R)}$ addition layer (see Fig. 2.5). The third operation can either be part of the final round (e.g., AES [69]), or is performed after the final round as a *post-whitening* operation (e.g., PRESENT [17]);
- the S-boxes (in particular the ones involved in the final round) are static and key-independent, i.e., they are included in the public specification of the cipher;
- the implementation is simple, i.e., the attacker is able to alter the definition of the S-boxes without affecting any key material.

The assumed attacker has more capabilities than in the entropy attack, i.e., the attacker is allowed to adaptively alter the implementation after performing a static analysis by means of reverse-engineering tools such as disassemblers or decompilers.

The workflow of Kerins et al.'s attack, referred to as the *S-box blanking attack*, is as follows. First, with the knowledge of the definition of the S-boxes, the attacker identifies the location of the (final round) S-box(es) within the software binary using static analysis tools. Typically, S-boxes are implemented as lookup tables. Second, all entries of the 'S-box' lookup tables are set to zero. This operation is called *S-box blanking.* Now, a single execution of the modified block cipher implementation for any given input reveals the final round key $k^{(R)}$. If the round keys are derived from the secret key through the use of a reversible key scheduling algorithm, the retrieval of $k^{(R)}$ eventually leads to the recovery of the secret key as well. The attack is depicted in Fig. 2.5.

Observe that the S-box blanking attack is very well suited for attacking simple software implementations of AES. An example of such a simple implementation is the lookup-table-based implementation for processors with word-length 32 bits proposed by Daemen and Rijmen (Sect. 2.3.2). Blanking the four $\texttt{SMC}_l$ $(l = 0, 1, 2, 3)$ tables, which contain the AES S-box, results in the recovery of the secret 128-bit key of AES-128.

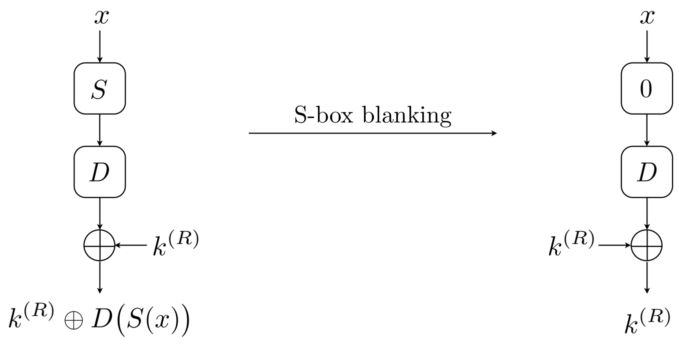


Figure 2.5: Blanking the final round S-boxes reveals the final round key $k^{(R)}$.

Countermeasures against the S-box blanking attack. In [55], Kerins and Kursawe proposed a number of countermeasures in order to preclude the S-box blanking attack:

Prevention of localizing the S-boxes: the most straightforward countermeasure to achieve this goal is to make the S-boxes key-dependent. However, one should be careful when altering the definition of S-boxes in standardized block ciphers as this change may introduce unexpected vulnerabilities, such as for example a low algebraic degree or bad linear/differential properties. Another countermeasure may be to implement S-boxes in a less straightforward way by means of S-box masking techniques or dynamically generation of the S-box during the execution of the program.

The use of key-dependent S-boxes is considered by Bringer, Chabanne and Dottax [20] in an attempt to construct a secure white-box implementation of a variant of AES. In Chapter 6, a cryptanalysis of this implementation is presented.

Integrity verification of the software binary: a different approach is to prevent an attacker from tampering with the software, i.e., overwriting the S-boxes. For example, only if the checksum over the binary validates, the code is executed. Much research has already been conducted in the field of Software Tamper Resistance (STR). In [73], Michiels and Gorissen proposed an application of white-box cryptography that enforces tamper resistant software: the program code (e.g., the license verification code in a DRM setting) is given a *dual* interpretation, i.e., it is both (i) executable code and (ii) part of a white-box implementation. Tampering with one of them breaks the code of the other and renders it unusable.

Modification of key usage/generation: the S-box blanking attack as discussed in [55] only recovers the final/post-whitening round key $k^{(R)}$. Hence, for independent round keys, the attack fails. Depending on the implementation, the attack may still be successful if the attacker is allowed to perform dynamic analysis with breakpoint functionality. Next, in the scenario of a double round key ($k_1^{(R)}$ and $k_2^{(R)}$) addition as the final operation of the cipher, the attacker would only obtain $k_1^{(R)} \oplus k_2^{(R)}$.

Reduced round attacks. As described in Sect. 2.5.1, the non-generic black-box attacks rely on reduced round distinguishers. Although a distinguisher that is insufficient to attack a full-round block cipher poses no real threat in the black-box model, it does in the white-box model. Typically, in the white-box setting, an attacker is able to access the reduced round versions of a block cipher in an encoded form. Even stronger, the white-box attacker often has access to encoded *single* round functions. Hence reduced (or even single) round distinguishers can be exploited by attacks against white-box implementations of block ciphers. For example, *differential cryptanalytic* techniques (see Link and Neumann [64], Wyseur et al. [105] and Goubin et al. [47]) have been proven successful by efficiently attacking Chow et al.'s white-box DES implementation [24].

As was stated before, although *algebraic attacks* have not (yet) been successful in attacking full-round block ciphers in the black-box model, they pose a real threat in the white-box model because of the same reason mentioned above; encoded reduced (or single) round versions of a block cipher vastly simplify the algebraic equations. For example, consider the algebraic attacks by Billet et al. [13] and Michiels et al. [75] on Chow et al.'s white-box AES implementation [23]. Chapters 3 and 4 elaborate on these attacks.

Grey-box cryptanalytic techniques (Sect. 2.5.2) should be taken into account as well. To some extent, DFA attacks can be seen as a form of differential cryptanalysis applied to a reduced s-round version of a block cipher, where most often $1 \leq s \leq 3$ as it is assumed (by the fault model) that the computational faults can be injected in one of the last rounds. Such attacks gain significant value in the white-box attack context. Whereas faults are injected probabilistically in the grey-box model, they can be injected deterministically in the white-box model. For example, consider the *differential fault analysis attack* by Jacob, Boneh and Felten [51] on Chow et al.'s white-box DES implementation [24]; in this attack, faults are injected in the input of the encoded final round of DES.

To conclude, one will notice that many white-box cryptanalytic techniques described in Part II find their origin in either the black-box and/or grey-box cryptanalytic techniques.

2.6 Conclusion

The primary goal when designing block ciphers is to approximate ideal block ciphers and hence to obtain a black-box secure cipher. However, in real-world applications, block ciphers are implemented in either hardware or software on physical devices. If the end-users of the communication channel who are in possession of these cryptographic physical devices can no longer be assumed to be trusted, the attacker no longer complies to the black-box model and thus the attack scenario changes drastically. The existence of grey-box and white-box cryptanalytic techniques shows that a black-box secure block cipher does not automatically lead to secure hardware/software implementations. In view of this additional threat, it clearly is insufficient to solely focus attention on designing block ciphers close to be ideal. As it turns out, an attacker will always look for the weakest link in the security chain in order to break the block cipher, hence a secure implementation plays an equally important role. This has been shown by the fairly simple *entropy attack* and *S-box blanking attack* with regard to software implementations of block ciphers employed in the white-box model.

As a consequence, a natural question is: *"Are there techniques to construct software implementations of block ciphers that offer a sufficient level of robustness against an attacker in the white-box model?"* The answer may be found in *white-box cryptography*, to which the second part of this thesis is dedicated. In particular, Part II elaborates on the design and analysis of white-box AES implementations.

Part II

Design and Analysis of White-Box Implementations

Chapter 3

Design and Analysis of White-Box Implementations

AES-128 as a Case Study

From Part I of this thesis it follows that it is crucial to implement a cryptographic primitive, such as a block cipher, with respect to the attack model in which it will be deployed. Especially since the introduction of the realistic grey-box and white-box attack models. An attacker within these attack models is considerably more powerful than a black-box attacker, i.e., instead of having only oracle access (i.e., input-output behavior) to the block cipher, the attacker furthermore has (limited) access to the hardware/software implementation of the block cipher. Hence, in the grey-box and white-box models, the attacker mainly focuses on weak implementation designs instead of weak cipher designs (in the case of a public specification of the block cipher). The research domain that specializes in developing secure software implementations of cryptographic primitives employed in the white-box model is called *white-box cryptography*. The resulting implementations are referred to as *white-box implementations*.

This chapter first elaborates on the objective of white-box cryptography and how this objective can be achieved in a practical manner with regard to block ciphers (both the encryption and decryption routine) by presenting a broad overview of the generic white-box techniques proposed by Chow, Eisen, Johnson and van Oorschot [23, 24] in 2002. Further, it describes in detail the first published white-box AES implementation presented by Chow et al. [23]. Second, this chapter elaborates on when a white-box implementation is insecure, followed by a complete overview of the analysis of Chow et al.'s white-box AES implementation. Finally, the concluding remarks outline an introduction to the following chapters.

3.1 White-Box Cryptography

White-box cryptography is a fairly recent research domain; it was introduced by Chow, Eisen, Johnson and van Oorschot [23] in 2002. Its existence emanates from the ever increasing demand to deploy strong cryptographic algorithms within software applications that are executed in an untrusted, possibly hostile, environment. In such an environment, referred to as the *white-box environment*, it is assumed that the attacker is in possession of the hardware/software of the cryptographic primitive, and furthermore has full access to its software implementation and full control over its execution platform. The powerful capabilities of an attacker in the white-box environment, referred to as a *white-box attacker*, are encapsulated by the *white-box attack context* [23] (see Def. 8 on p. 30). Although the white-box attack context is the worst case attack model with respect to cryptographic software (here, the term 'worst case' is from the perspective of the designer of the cryptographic software), the use case discussed in Chapter 1 illustrates that it is at the same time also a realistic attack model.

White-box cryptography aims at protecting the software implementation of cryptographic primitives when executed in a white-box environment. The objective is to provide a sufficient level of robustness against a white-box attacker. What is meant by this robustness against a white-box attacker? When is a software implementation of a cryptographic primitive considered to be secure within the white-box attack context? The following statement should provide a preliminary answer.

Definition 14 (White-box cryptography)**.** *White-box cryptography is a technique that aims at transforming a cryptographic primitive into a functionally equivalent software implementation in such a way that the software implementation behaves as a "virtual black box", i.e., a white-box attacker who has full access to the software implementation as well as full control over its execution environment has no additional advantage over a black-box attacker restricted to oracle access to the implementation in order to achieve his goal (i.e., total break or global deduction). In the foregoing, both adversaries are assumed to be computationally bounded.*

Note that Def. 14 is similar to the one given by Wyseur in his doctoral thesis [103, p. 36, Def. 3]. Recall from Sect. 2.4 that the assessment of the security of (the implementation of) a cryptographic primitive is based on a security notion, which comprises the following two factors: (i) the attack model and (ii) the attacker's goal. Observe that both factors are present in Def. 14. First, a *white-box* and *black-box attacker* refer to an attacker in the white-box and black-box model, respectively. Recall from Sect. 2.4.6 that the number of queries is always bounded in the black-box model, but not in the white-box model. Second, the attacker tries to achieve his goal, where typically two different goals (out of the hierarchical classification given in Sect. 2.4.4) play a crucial role in the field of white-box cryptography, i.e., *total break (i.e., key recovery)* and *global deduction*. Section 3.4 elaborates in detail on the achievement of both goals within the white-box environment.

With regard to Def. 14, it is interesting to make the following two observations.

1) The white-box model is forced into the black-box model. As mentioned in the introduction of the grey-box attack model in Sect. 2.4.5 on p. 28, the traditional black-box model assumes that any implementation of a cryptographic primitive behaves as an *ideal black box* (referred to as a *virtual black box* in Def. 14). Hence white-box cryptography strives to ensure that the white-box model does not provide any additional advantage over the black-box model.

2) White-box cryptography is a 'special' obfuscation technique. Note that Def. 14 closely resembles the informal definition of an obfuscator of programs introduced by Barak et al. [1]:

Definition 15 (Program obfuscation [1]). *Program obfuscation is a technique that aims at transforming a program P into a functionally equivalent obfuscated program $\mathcal{O}(P)$ behaving as a virtual black box (called the virtual black-box property (VBBP)), i.e., anything that can be efficiently learned from $\mathcal{O}(P)$ could also have been efficiently learned when given only oracle access to P.*

Under this resemblance, white-box cryptography can be considered as a 'special' kind of obfuscation technique. The emphasis should definitely be placed on 'special' as the programs that should be obfuscated into virtual black boxes are cryptographic primitives whose specifications are often publicly known with the exception of the secret key information. Therefore, the primary goal of "white-box obfuscation" is not hiding the functionality of the program, but keeping the key information secret. As pointed out by Yamauchi et al. [108], one should be very careful when applying conventional obfuscation techniques in order to hide a secret (i.e., the cryptographic key) in a public known algorithm.

To build further on the observation that white-box cryptography can be considered as a special kind of obfuscation technique, a quote by Chow et al. [23] is given in the following:

"When the attacker has internal information about a cryptographic implementation, choice of implementation is the sole remaining line of defense. Choice of implementation is precisely what is pursued in white-box cryptography." [23]

The software implementations obtained as a result of the application of white-box cryptography on cryptographic algorithms are referred to as a *white-box implementations*. Up to early 2014, in the academic literature, the focus of white-box cryptography is mainly on the design of white-box implementations of block ciphers. The white-box implementations covered in this chapter are solely based on lookup tables (see Def. 3 on p. 16). The storage requirement of lookup tables (see Property 1 on p. 16) is of great significance with respect to the practicality of white-box implementations. As a straightforward example, the ideal white-box implementation is discussed below.

3.1.1 Ideal White-Box Implementation

As indicated in [23] by Chow et al., the ideal but at the same time also the most impractical white-box implementation is to represent a key-instantiated n_b-bit block cipher E by means of a single lookup table mapping $\mathcal{L}_E$ n_b bits to n_b bits. In particular, $\mathcal{L}_E$ represents the entire *code book* of E, i.e., the table stores the ciphertexts for all 2^{n_b} possible plaintexts encrypted using the same fixed secret key. Interestingly, it is also this lookup table that is used in the generic code book black-box attack (see Sect. 2.5.1, p. 34).

The reason why this implementation achieves *perfect white-box security* (with respect to key extraction), which corresponds to achieving black-box security within the white-box environment, is straightforward: the lookup table $\mathcal{L}_E$ behaves as an *ideal black box*, i.e., it maps plaintexts directly to ciphertexts. However, the storage requirement of such an ideal lookup table implementation is given by $2^{n_b} \cdot n_b$ bits and hence becomes impractical/unrealistic for common values of the block size n_b. As an example, a lookup table representing the codebook of a key-instantiated AES-128 cipher would require $4.95 \cdot 10^{27}$ TB of storage space! It should be mentioned that although the ideal white-box implementation prevents key-extraction, it allows global deduction, i.e., the implementation itself is functionally equivalent to the key-instantiated block cipher. However, due to the impractical aspect of the implementation, this becomes irrelevant.

The following section elaborates in detail on the white-box techniques proposed by Chow, Eisen, Johnson and van Oorschot [23, 24] in order to obtain *practical* white-box implementations of block ciphers.

3.2 Initial Practical White-Box Techniques

In [23, 24], Chow et al. present generic techniques that can be used to design practical white-box implementations of a symmetric cryptographic algorithm (in particular a block cipher). The presented techniques can be applied to both Feistel and SPN block ciphers. The main objective is to resist key extraction in a white-box environment, however, as is discussed in Sect. 3.4, some applications demand that additional 'stronger' security requirements need to be satisfied.

In general, the generic white-box techniques of Chow et al. comprise the following two phases, which can be applied to both the encryption as well as the decryption routine of a block cipher:

Phase 1 writes the block cipher, instantiated with a fixed key, as a series of *practical* (i.e., small storage requirement) lookup tables.

Phase 2 applies secret invertible white-box encodings (both $\mathbb{F}_2$-affine and non-affine functions) to the input and output of all tables. These white-box encodings are pairwise annihilating between successive lookup tables in order to preserve the overall functionality of the block cipher.

The descriptions given below provide detailed information about each phase. In the following, it is assumed that the entire specification of the block cipher is publicly known with the exception of the embedded secret key.

Phase 1: Translating the description of the block cipher into a series of lookup tables

The first phase translates the description of a key-instantiated block cipher into a functionally equivalent network of lookup tables by merging several steps of the round function. As mentioned in the description of the design of block ciphers (see Sect. 2.2), the round function of a block cipher typically consists of the following three operations: (A) a key addition operation, (B) a 'confusion' operation consisting of the parallel execution of non-linear S-boxes, and (C) a 'diffusion' operation consisting of wide linear/affine mappings. Below, the following two techniques [23, 24] are described: (i) *partial evaluation* merges operations A and B, and (ii) *matrix partitioning* handles the lookup table representation of operation C.

Partial evaluation. Assume that the S-box operation directly follows the key addition, which is common in the design of many widespread block ciphers such as DES and AES. Further, let S denote a non-linear m-to-n bit S-box and let $k \in \mathbb{F}_2^m$. As the embedded secret key is fixed, the key addition can be treated as a constant operation and hence can be pre-evaluated, i.e., the function $\oplus_k(x) = x \oplus k$ can be evaluated for all values of x. Next, both the addition by k and the S-box are merged into the key-dependent mapping

$$T : \mathbb{F}_2^m \to \mathbb{F}_2^n : x \mapsto T(x) = S(x \oplus k) \ . \tag{3.1}$$

The mapping T is represented by a key-dependent lookup table, referred to as a T-box, mapping m bits to n bits.

Matrix partitioning. For *wide* linear/affine mappings, where 'wide' refers to an input size of the mapping larger than 16 bits, it quickly becomes impractical (see Property 1) to represent the mapping by a single lookup table. Therefore, the matrix-vector multiplication is decomposed into an exclusive-OR network of smaller lookup tables by partitioning the matrix and input/output vectors accordingly. This is illustrated in the following, where an affine mapping is defined as $y = L \cdot x \oplus c$ with $y, c \in \mathbb{F}_2^{an}$, $L \in \mathbb{F}_2^{an \times bm}$ and $x \in \mathbb{F}_2^{bm}$. Based on the partitioning

$$\left[\begin{array}{c} y_1 \\ \hdashline \vdots \\ \hdashline y_a \end{array}\right] = \left[\begin{array}{c:c:c} L_{1,1} & \cdots & L_{1,b} \\ \hdashline \vdots & \ddots & \vdots \\ \hdashline L_{a,1} & \cdots & L_{a,b} \end{array}\right] \cdot \left[\begin{array}{c} x_1 \\ \hdashline \vdots \\ \hdashline x_b \end{array}\right] \oplus \left[\begin{array}{c} c_1 \\ \hdashline \vdots \\ \hdashline c_a \end{array}\right] \ ,$$

where $y_i, c_i \in \mathbb{F}_2^n$, $L_{i,j} \in \mathbb{F}_2^{n \times m}$ and $x_j \in \mathbb{F}_2^m$ for $1 \leq i \leq a$ and $1 \leq j \leq b$, the partitioned output $y = (y_1, y_2, \ldots, y_a)$ can be calculated by

$$y_i = \left(\bigoplus_{j=1}^{b-1} \underbrace{L_{i,j} \cdot x_j}_{\mathcal{L}_{i,j}} \right) \oplus \underbrace{L_{i,b} \cdot x_b \oplus c_i}_{\mathcal{L}_{i,b}} \qquad \text{for } i = 1, 2, \ldots, a \ , \tag{3.2}$$

where each exclusive-OR operates on n bits. Now, by representing the addends of the summation (3.2) by lookup tables $\mathcal{L}_{i,j}$ ($1 \leq i \leq a$ and $1 \leq j \leq b-1$) and $\mathcal{L}_{i,b}$ ($1 \leq i \leq a$) mapping m bits to n bits, the expression $y = L \cdot x \oplus c$ is represented by an exclusive-OR network of $a \cdot b$ smaller lookup tables; each table has a storage requirement of $2^m \cdot n$ bits. As we discuss in Phase 2 below, at some point the XOR operations need to be represented by lookup tables as well. As these operations take two n-bit values as input in order to produce one n-bit value as output, the input size of the lookup table is given by $2n$ bits. Because of Property 1, this poses a restriction on n. Typically, if n is a multiple of 4, i.e. $n = n' \cdot 4$, the XOR operation on n bits is replaced by the parallel execution of n' XOR operations on 4 bits (i.e., on nibbles), where each nibble

XOR operation is represented by a lookup table $\mathcal{L}_{\oplus}$ mapping 8 bits to 4 bits, defined as follows: $\mathcal{L}_{\oplus}(x, y) = x \oplus y$ with $x, y \in \mathbb{F}_2^4$.

By applying both techniques described above, one can translate the description of any Feistel or SPN block cipher into a network of interconnected lookup tables. Note that this phase is not generic, i.e., the above techniques should be considered as an indication only and, depending on the description of the block cipher, variations may occur in order to meet additional requirements.

Phase 2: Applying secret invertible white-box encodings

The second phase concerns the injection of randomness into the lookup-table implementation obtained in Phase 1. This phase is necessary as the implementation obtained in Phase 1 provides no protection with respect to key extraction. As an example, given access to a T-box as defined in (3.1), the attacker can extract the embedded key by calculating $k = S^{-1}\big(T(0)\big)$. In order to prevent an attacker from extracting any key information out of the key-dependent lookup tables, secret invertible white-box encodings are applied to the input and output of all lookup tables. This brings us to the definition of encoded lookup tables.

Definition 16 (Encoded lookup table [24, Def. 1]). *Let $\mathcal{L}$ denote a lookup table mapping m bits to n bits, and let f and g denote random bijective mappings on the vector space $\mathbb{F}_2^m$ and $\mathbb{F}_2^n$, respectively. Then $\mathcal{L}'$ is called the encoded version of $\mathcal{L}$ if it is is defined as $\mathcal{L}' = g \circ \mathcal{L} \circ f^{-1}$, where f^{-1} is called the input encoding and g is called the output encoding.*

In order to maintain the overall functionality of the lookup-table implementation representing the block cipher, the output and input encodings of successive lookup tables should be pairwise annihilating. This way, an input encoding can also be referred to as an input *decoding* as it decodes the encoded output of the preceding lookup table. This is referred to as *networked encoding* in [24, 23].

Definition 17 (Networked encoding [24, Def. 3]). *A networked encoding for computing $\mathcal{L}_Y \circ \mathcal{L}_X$ (where $\mathcal{L}_X$ and $\mathcal{L}_Y$ denote the lookup tables representing the transformations X and Y, respectively) in an encoded form is given by*

$$\mathcal{L}'_Y \circ \mathcal{L}'_X = (h \circ \mathcal{L}_Y \circ g^{-1}) \circ (g \circ \mathcal{L}_X \circ f^{-1}) = h \circ (\mathcal{L}_Y \circ \mathcal{L}_X) \circ f^{-1} \ ,$$

where $\mathcal{L}'_X$ and $\mathcal{L}'_Y$ denote the encoded versions of $\mathcal{L}_X$ and $\mathcal{L}_Y$, respectively. Note that the output encoding g of $\mathcal{L}'_X$ and the input encoding g^{-1} of $\mathcal{L}'_Y$ are pairwise annihilating.

Due to Property 1 and the pairwise annihilating property of the white-box encodings between the output and input of successive lookup tables, restrictions are posed on the choice of the encodings f and g in Def. 16, i.e., typically they cannot be chosen as random permutations on $\mathbb{F}_2^m$ and $\mathbb{F}_2^n$, respectively. Each invertible white-box encoding should be decomposed into its affine and non-affine component as different rules apply to both. The affine component achieves *diffusion* in the intermediate values of the white-box implementation, whereas the non-affine component achieves *confusion*.

Affine component. As discussed in Phase 1 above, the *matrix partitioning* technique ensures that 'wide' affine mappings (and their inverses) can be represented by a XOR network of small lookup tables. Naturally, this also applies to the affine component of the white-box encodings. As a result, there are no restrictions on the size of the vector space over $\mathbb{F}_2$ upon which the affine mappings operate. Therefore, the affine component of f or g (denoted by F or G, respectively) may be any bijective affine mapping on $\mathbb{F}_2^m$ or $\mathbb{F}_2^n$, respectively.

In order to achieve good diffusion, Chow et al. [24, 23] consider a specific class of bijective affine mappings, i.e., the class of *mixing bijections*; a mixing bijection is a bijective affine mapping that attempts to maximize the dependency of each output bit on all input bits (also referred to as the *avalanche effect*). Therefore, Chow et al. recommend to select $n \times n$ non-singular matrices over $\mathbb{F}_2$ with 4×4 submatrices of full rank as the $\mathbb{F}_2$-linear part of n-bit mixing bijections (under the assumption that n is a multiple of 4). The dimension of these submatrices are in line with the typical dimension of the non-affine components of the white-box encodings (see below) in order to maximize diffusion. Ways how to construct such non-singular matrices are proposed by Xiao and Zhou [106] and by Muir [78].

Special attention should be given when encoding the output of the lookup tables involved in a XOR network, i.e., the affine component of the white-box encodings should be chosen carefully. To illustrate this, consider the XOR network of lookup tables given by $y = \bigoplus_{j=1}^{b} \mathcal{L}_j(x_j)$ (see (3.2) as an example): by encoding the output of each table $\mathcal{L}_j$ by the bijective affine mapping $G_j(x) = L(x) \oplus c_j$ for $1 \le j \le b$, respectively, with $c = \bigoplus_{j=1}^{b} c_j$, one gets

$$G(y) = \bigoplus_{j=1}^{b} G_j\big(\mathcal{L}_j(x_j)\big) \qquad \text{with } G(y) = L(y) \oplus c \ . \tag{3.3}$$

Hence by carefully selecting the affine encodings G_j $(1 \le j \le b)$, the affine encoding G is carried through the XOR operations as it were. Furthermore, each XOR operation can be executed on encoded date, i.e., no representation by lookup tables is necessary.

Non-affine component. In contrast to affine mappings, there exists no technique to represent 'wide' non-affine permutations (and their inverses) by a network of small lookup tables. As a result, the size of the vector space over $\mathbb{F}_2$ upon which the non-affine permutations operate depends on the input size of the subsequent lookup table(s) or on the output size of the preceding lookup table(s). Typically, as discussed below, the non-affine component of f or g is a concatenation of random non-affine permutations on $\mathbb{F}_2^4$ (if m and n are multiples of 4).

As mentioned above, if the white-box encodings consist solely of an affine component, the XOR operations can be executed on encoded data by carefully selecting the affine encodings (see (3.3)). However, once the white-box encodings comprise a non-affine component as well, the encodings can no longer be carried through the XOR operation, i.e., the non-affine component needs to be annihilated before performing the XOR operation. Hence, each XOR operation needs to be represented by means of encoded lookup tables. As discussed in Phase 1, a XOR operation on n bits (assuming that $n = n' \cdot 4$) is replaced by n' parallel nibble XOR operations (each represented by a lookup table $\mathcal{L}_\oplus$ mapping 8 bits to 4 bits). As the 8-bit input of $\mathcal{L}_\oplus$ is a concatenation of two different 4-bit inputs, the encoded version of $\mathcal{L}_\oplus$ is defined as $\mathcal{L}'_\oplus = g \circ \mathcal{L}_\oplus \circ (f_1^{-1}, f_2^{-1})$ where g, f_1 and f_2 are random non-affine permutations on $\mathbb{F}_2^4$. This lookup-table representation of XOR operations typically causes a bottleneck on the composition of the non-affine component of the white-box encodings, as explained in the following.

Summary: affine and non-affine components of encodings. In table-based white-box implementations, it is common that $\mathcal{L}'_\oplus$ tables precede and succeed an encoded table $\mathcal{L}' = g \circ \mathcal{L} \circ f^{-1}$ since XOR operations occur frequently due to the application of *matrix partitioning* on 'wide' linear/affine mappings (which are either part of the block cipher's specification or the affine component of white-box encodings). In that case, the composition of the white-box encodings f and g of $\mathcal{L}'$ is given by

$$\begin{array}{rclcl@{\qquad}lrcl}
f &=& \big(f_1, f_2, \ldots, f_{m'}\big) &\circ& F & \text{with} & m &=& m' \cdot 4 \quad ,\\
g &=& \big(g_1, g_2, \ldots, g_{n'}\big) &\circ& G & \text{with} & n &=& n' \cdot 4 \quad ,
\end{array}$$

where f_i $(1 \le i \le m')$ and g_i $(1 \le i \le n')$ are random non-affine permutations on $\mathbb{F}_2^4$, and where F and G denote random mixing bijections on $\mathbb{F}_2^m$ and $\mathbb{F}_2^n$, respectively. Observe that the affine component of the white-box encodings is applied to the input and output of each lookup table before the non-affine component. More generally, depending on the lookup tables preceding and succeeding $\mathcal{L}'$, the non-affine components f_i and g_i may be random non-affine permutations with varying dimensions. However, as the storage requirement of a lookup table with $m > 16$ quickly becomes impractical, the dimension of the non-affine permutations is most likely to be upper bounded by 16. The above is captured by *concatenated encoding* and *I/O-blocked encoding* in [24, 23].

As a final remark, it is interesting to note that the application of non-affine encodings to all lookup tables results in the fact that each linear/affine mapping occurring in the description of the block cipher becomes non-linear/non-affine in the white-box implementation. This is referred to as *de-linearization.*

External encodings. It is common practice in white-box cryptography to implement an encoded version E'_k of a key-instantiated block cipher E_k. In such an encoded version, invertible white-box encodings are applied to the boundaries of the block cipher, i.e., to the plaintext and ciphertext, such that

$$E'_k = \mathtt{OUT} \circ E_k \circ \mathtt{IN}^{-1} \ , \tag{3.4}$$

where the encodings IN and OUT are referred to as *external encodings.* As a consequence, E'_k takes as input an *encoded* plaintext $\mathtt{IN}(P)$ and outputs the *encoded* ciphertext $\mathtt{OUT}(C)$ where $C = E_k(P)$. Ideally, if E_k is an n_b-bit block cipher, the external encodings are random permutations on $\mathbb{F}_2^{n_b}$. However, as such permutations cannot be implemented efficiently by a single lookup table, instead the external encodings are typically selected as random bijective affine mappings (mixing bijections) on $\mathbb{F}_2^{n_b}$. Next, the *matrix partitioning* technique (see Phase 1) allows the external encodings to be implemented as a XOR network of small lookup tables, which eventually is de-linearized by applying (the concatenation of) pairwise annihilating random non-affine permutations on $\mathbb{F}_2^4$ to all lookup tables (due to the encoded nibble XOR tables $\mathcal{L}'_\oplus$). As is the case for all white-box encodings, the external encodings are kept secret in a white-box implementation.

The motivation behind the use of external encodings is two-fold (see Wyseur [103, Sect. 3.2.3]) and can be summarized as follows:

Prevent code lifting: this reflects the original motivation as presented by Chow et al. in [24, 23]. If the white-box implementation is functionally equivalent to the standard decryption routine D_k of a key-instantiated block cipher (i.e., no external encodings are applied), one might question the relevance of extracting the embedded key as the attacker is already in possession of the software (i.e., the white-box implementation) mapping the original ciphertext to the corresponding plaintext. Extracting pieces of code and using it in an isolated manner is referred to as *code lifting.* In order to prevent code lifting, Chow et al. proposed to apply secret invertible external encodings to the boundaries of the block ciphers. These external encodings appear as pairwise annihilating transformations such that they need to be applied to the plaintext and removed from the ciphertext elsewhere outside the white-box implementation (e.g., on a remote server or on the user's playback device). Hence, the white-box implementation becomes useless in an isolated setting outside its containing application.

Another solution to prevent code lifting is the technique called *node locking* (cf. Michiels [70]), which limits the correct execution of the white-box implementation to one user's device. This technique relies on the ability to include an arbitrary bit string (in the form of an arbitrary set of lookup tables) in a white-box implementation in such a way that (i) the implementation's functionality is preserved, and (ii) the attacker is unable to remove or modify the included bit string without affecting the implementation's functionality (see Michiels and Gorissen [73]). The bit string used for node locking can be a unique hardware identifier of the user's device.

Increase the protection of the outer rounds' white-box implementation: if no external encodings are applied, the lookup tables at the boundaries of the white-box implementation are only partially encoded, i.e., the first lookup tables are unencoded at their input and the final lookup tables are unencoded at their output. Therefore, these particular lookup tables form a primary target for cryptanalysis. From another perspective, the white-box attacker is typically able to isolate the white-box implementation of the first and final round and furthermore is able to observe and adaptively choose the plaintext and ciphertext, which makes the white-box implementation of the outer rounds more vulnerable to cryptanalysis. The application of external encodings may provide a solution for the above weaknesses of the white-box implementation.

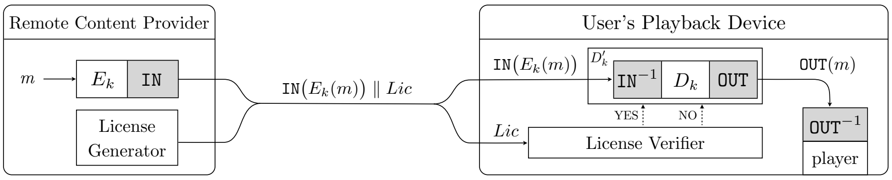


Figure 3.1: Use case of external encodings: a simplified DRM model.

Although the use of external encodings clearly has its advantages (as pointed out above), it also has the following main disadvantage: the white-box implementation is functionally equivalent to the encoded block cipher E'_k instead of the standard block cipher E_k, i.e., the input and output is provided in an *encoded* form. Depending on the application in which the white-box implementation is deployed, the use of external encodings can be justified or not. For example, in the DRM setting (Fig. 3.1) where the white-box implementation is functionally equivalent to the encoded decryption routine $D'_k = \mathtt{OUT} \circ D_k \circ \mathtt{IN}^{-1}$, the external encoding `IN` is applied at the side of the remote content provider, whereas the external encoding `OUT` is annihilated on the user's content player.

3.3 White-Box AES Implementation

Chow et al. apply their generic white-box techniques to DES [68] and AES-128 [69] in order to define an example white-box DES implementation [24] and an example white-box AES-128 implementation [23]. The doctoral thesis by Wyseur [103, Chapter 3] provides a comprehensive overview of the design of Chow et al.'s white-box DES implementation [24] and the cryptanalytic results on this implementation [51, 64, 105, 47]. This section elaborates on Chow et al.'s white-box AES-128 implementation [23]. The cryptanalytic techniques on the white-box AES implementation are presented in Sect. 3.5.

3.3.1 Lookup-Table Suitable Description of AES-128

Recall from Sect. 2.3.1 that there are several equivalent ways to describe AES-128. As pointed out by Muir [78], the alternative description depicted in Fig. 2.1b (p. 21) is particularly suitable for converting AES-128 into a network of lookup tables. The main reason for this is that the steps `AddRoundKey`, `SubBytes` and `MixColumns` are adjacent in the round function which simplifies the merger of these steps into lookup tables.

AES subrounds. In order to support the description of the white-box AES implementations (as the one in this section) and the cryptanalytic techniques on the implementations (see Sect. 3.5 and Chapter 4), the mappings in the following definition are introduced for rounds $1 \leq r \leq 9$. These mappings take advantage of the fact that the steps `AddRoundKey`, `SubBytes` and `MixColumns` are adjacent, and of the fact that the `MixColumns` step comprises the parallel application of four instances of the `MixColumns` matrix operation. In the following text, $\oplus$ and $\otimes$ denote the addition and multiplication operations in the AES polynomial representation of $\mathbb{F}_{256}$, respectively.

Definition 18 (AES subround). *Let $x_i, y_i \in \mathbb{F}_{256}$ for $0 \leq i \leq 3$ be represented using the AES polynomial representation. The mapping $\mathit{AES}^{(r,j)} : \mathbb{F}_{256}^4 \rightarrow \mathbb{F}_{256}^4$ for $1 \leq r \leq 9$ and $0 \leq j \leq 3$, called an AES subround, is defined by*

$(y_0, y_1, y_2, y_3) = \mathit{AES}^{(r,j)}(x_0, x_1, x_2, x_3)$ *with*

$$y_i = mc_{i,0} \otimes S\big(x_0 \oplus \hat{k}_0^{(r,j)}\big) \oplus mc_{i,1} \otimes S\big(x_1 \oplus \hat{k}_1^{(r,j)}\big) \oplus$$
$$mc_{i,2} \otimes S\big(x_2 \oplus \hat{k}_2^{(r,j)}\big) \oplus mc_{i,3} \otimes S\big(x_3 \oplus \hat{k}_3^{(r,j)}\big) \ ,$$

for $0 \le i \le 3$. *Recall from Sect. 2.3.1 that* $mc_{i,j}$ *(*$0 \le i, j \le 3$*) denote the* `MixColumns` *coefficients.*

Observe that an AES subround consists of the key additions, the S-box operations and the `MixColumns` operation in an AES round that are associated with a single `MixColumns` matrix operation, and that each AES round r with $1 \le r \le 9$ comprises four AES subrounds in parallel. The subrounds are indexed by j in Def. 18, and it is assumed that the four subrounds in a round are numbered left to right. The bytes $\hat{k}_i^{(r,j)}$ for $0 \le i, j \le 3$ are the 16 bytes of the AES round key of round r.

Based on these *AES subround* mappings, the alternative description of AES-128 (Fig. 2.1b) for rounds $1 \le r \le 9$ is now given as follows:

```
for $r = 1$ to 9 do
    state ← ShiftRows(state)
    for $j = 0$ to 3 do
        $[\text{state}_{i,j}]_{0\le i\le 3} \leftarrow \texttt{AES}^{(r,j)}\big([\text{state}_{i,j}]_{0\le i\le 3}\big)$
    end for
end for
```

3.3.2 White-Box AES-128 Implementation

In 2013, to celebrate the tenth anniversary of the initial white-box cryptography papers by Chow et al., Muir [78] presented a clear in-depth tutorial on how Chow et al.'s white-box AES implementation is constructed, with more details than in the original paper [23]. The tutorial by Muir is used as a guideline to describe white-box AES-128 in the following. Now, recall from Sect. 3.2 that the process of generating a white-box implementation comprises two phases; each phase is described in detail below.

Phase 1: Translating AES-128 into a series of lookup tables

First, by means of *partial evaluation*, the `AddRoundKey` and `SubBytes` operations of the round function are composed, resulting in 16 8-bit bijective key-dependent lookup tables for each round. In the following, such a table is referred to as a *T-box.* By adopting the notations of the round keys introduced in the alternative description of AES-128, these T-boxes are defined as

$$\begin{array}{lcll} T_i^{(r,j)}(x) & = & S(x \oplus \hat{k}_i^{(r,j)}) & \text{for } 0 \leq i,j \leq 3 \text{ and } 1 \leq r \leq 9 \ , \\ T_i^{(10,j)}(x) & = & S(x \oplus \hat{k}_i^{(10,j)}) \oplus k_i^{(11,j)} & \text{for } 0 \leq i,j \leq 3 \ . \end{array}$$

Observe that the 16 T-boxes of the final round incorporate the bytes of two round keys, i.e., the final round key $\hat{k}^{(10)}$ and the post-whitening key $k^{(11)}$.

Second, by means of *matrix partitioning*, the 4×4 matrix MC over $\mathbb{F}_{256}$ representing the MixColumns operation is split into four 4×1 submatrices over $\mathbb{F}_{256}$: $\texttt{MC}_i$ is defined as column i of MC for $i = 0, 1, 2, 3$. Using this notation, the MixColumns matrix-vector multiplication is decomposed into a XOR of four 32-bit values, given by

$$\begin{pmatrix} \text{state}_{0,j} \\ \text{state}_{1,j} \\ \text{state}_{2,j} \\ \text{state}_{3,j} \end{pmatrix} \leftarrow \bigoplus_{i=0}^{3} \texttt{MC}_i \cdot \text{state}_{i,j} \qquad \text{for } j = 0, 1, 2, 3 \ . \tag{3.5}$$

For rounds $1 \leq r \leq 9$, the T-boxes and MixColumns operation are then merged together by constructing 16 $\texttt{TMC}_i^{(r,j)}$ $(0 \leq i, j \leq 3)$ lookup tables per round, each table mapping 8 bits to 32 bits and composed as

$$\texttt{TMC}_i^{(r,j)} = \texttt{MC}_i \circ T_i^{(r,j)} \qquad \text{for } 0 \leq i,j \leq 3 \text{ and } 1 \leq r \leq 9 \ .$$

Observe that these tables merge the adjacent AddRoundKey, SubBytes and MixColumns steps of the AES round function for $1 \leq r \leq 9$. For the final round, the AddRoundKey and SubBytes steps are already merged by means of the 16 key-dependent T-boxes $T_i^{(10,j)}$ $(0 \leq i, j \leq 3)$. Concerning the ShiftRows operation at the beginning of each round function, it can be incorporated into the data-flow of the implementation (i.e., providing shifted inputs to the tables) as it is only a permutation on the indices of the bytes of the AES state.

At this point, almost the entire alternative description of AES-128 is converted into a network of lookup tables, with the exception of the 32-bit XOR operations which are due to the matrix-vector multiplication decomposition of MC (see (3.5)) for rounds $1 \leq r \leq 9$. To illustrate this, observe that each AES subround $\texttt{AES}^{(r,j)}$ $(0 \leq j \leq 3$ and $1 \leq r \leq 9)$ can now be computed as

$$(y_0, y_1, y_2, y_3) = \texttt{AES}^{(r,j)}(x_0, x_1, x_2, x_3) = \bigoplus_{i=0}^{3} \texttt{TMC}_i^{(r,j)}(x_i) \ .$$

Now, recall from Sect. 3.2 (Phase 1) that each 32-bit XOR operation can be represented by 8 nibble XOR tables $\mathcal{L}_{\oplus}$ in parallel. As a result, AES-128 can be completely translated into a series of lookup tables; the resulting lookup-table implementation of AES-128 is described in Fig. 3.2a. Figure 3.2b depicts the implementation of an AES subround for $0 \leq j \leq 3$ and $1 \leq r \leq 9$ comprising solely lookup tables.

Note that the standard software AES-128 implementation described in Sect. 2.3.2 (referred to as implementation A) relates to the lookup-table implementation of AES-128 as depicted in Fig. 3.2 (referred to as implementation B). While implementation A only merges the `SubBytes` and `MixColumns` operations by means of *matrix partitioning*, implementation B additionally merges the `AddRoundKey` operation (with fixed key) as well by means of *partial evaluation*. This results in the following lookup tables for both implementations:

$$\begin{array}{lll} \text{Implementation A:} & \mathtt{SMC}_i = \mathtt{MC}_i \circ S & (0 \leq i \leq 3) \\ \text{Implementation B:} & \mathtt{TMC}_i^{(r,j)} = \mathtt{MC}_i \circ T_i^{(r,j)} & (0 \leq i,j \leq 3 \text{ and } 1 \leq r \leq 9) \\ & T_i^{(10,j)} & (0 \leq i,j \leq 3) \end{array}$$

Observe that by embedding the fixed round key bytes into the lookup tables, the number of lookup tables increases from 4 8-to-32 bit tables to 144 8-to-32 bit tables and 16 8-to-8 bit tables (disregarding the nibble XOR tables $\mathcal{L}_{\oplus}$).

Phase 2: Applying secret invertible white-box encodings

The lookup-table implementation of AES-128 obtained in Phase 1 provides no protection with respect to key extraction in a white-box environment. In such an environment, the attacker has access to the 16 $\mathtt{TMC}_i^{(1,j)}$ $(0 \leq i,j \leq 3)$ tables of the first round, defined as

$$\mathtt{TMC}_i^{(1,j)} = \mathtt{MC}_i \circ T_i^{(1,j)} = \mathtt{MC}_i \circ S \circ \hat{k}_i^{(1,j)} \qquad \text{for } 0 \leq i,j \leq 3 \ .$$

By exploiting the fact that $\mathtt{MC}$ is a $\mathbb{F}_2$-linear operation and that $S(\mathtt{52}) = \mathtt{00}$, the 8-bit value $x_{i,j}$ for which $\mathtt{TMC}_i^{(1,j)}(x_{i,j}) = 0$ is given by $x_{i,j} = \hat{k}_i^{(1,j)} \oplus \mathtt{52}$. This enables the attacker to compute

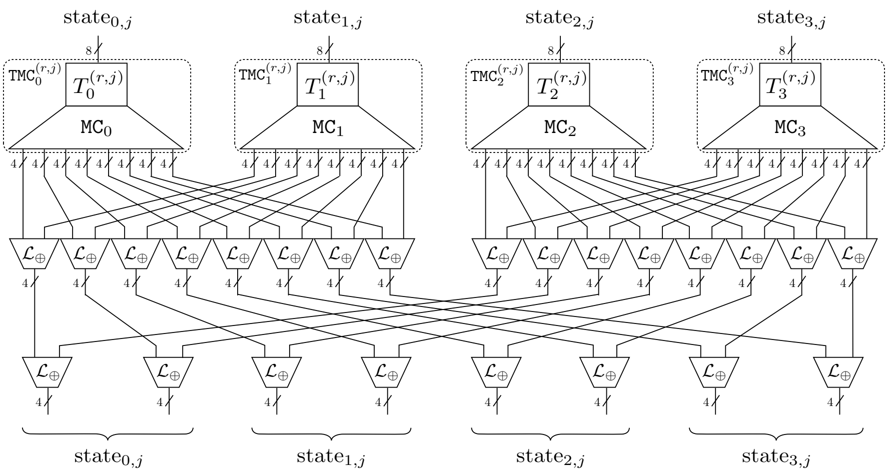
$$\hat{k}_i^{(1,j)} = x_{i,j} \oplus \mathtt{52} \qquad \text{for } 0 \leq i,j \leq 3 \ .$$

Recall that for AES-128, the first round key equals the actual 128-bit AES key, hence the attacker can easily extract the AES key.

As pointed out by Muir [78], the key-dependent lookup tables, i.e., the tables $\mathtt{TMC}_i^{(r,j)}$ $(0 \leq i,j \leq 3 \text{ and } 1 \leq r \leq 9)$ and $T_i^{(10,j)}$ $(0 \leq i,j \leq 3)$, can be considered

```
state ← plaintext
for $r = 1$ to 9 do
    state ← ShiftRows(state)
    for $j = 0$ to 3 do
        $[\text{state}_{i,j}]_{0 \le i \le 3} \leftarrow \bigoplus_{i=0}^{3} \mathtt{TMC}_i^{(r,j)}(\text{state}_{i,j})$
                where each $\bigoplus = \mathcal{L}_\oplus \parallel \mathcal{L}_\oplus \parallel \mathcal{L}_\oplus \parallel \mathcal{L}_\oplus \parallel \mathcal{L}_\oplus \parallel \mathcal{L}_\oplus \parallel \mathcal{L}_\oplus \parallel \mathcal{L}_\oplus$
    end for
end for
state ← ShiftRows(state)
for $j = 0$ to 3 do
    for $i = 0$ to 3 do
        $\text{state}_{i,j} \leftarrow T_i^{(10,j)}(\text{state}_{i,j})$
    end for
end for
ciphertext ← state
```

(a) Description of AES-128 as a network of lookup tables.



(b) Lookup-table implementation of an AES subround for $0 \le j \le 3$ and $1 \le r \le 9$.

Figure 3.2: Translating AES-128 into a series of lookup tables.

as miniature block ciphers. Hence, to prevent an attacker from extracting the AES round keys from these key-dependent tables, both $\mathbb{F}_2$-linear (achieving *diffusion*) and non-linear (achieving *confusion*) secret invertible white-box encodings are applied to the input and output of all lookup tables. Note that in Sect. 3.2 (Phase 2), the more general distinction was made between $\mathbb{F}_2$-affine and non-affine encodings; however, since the constant vector of the affine mappings can be included in the non-affine mappings, both distinctions come down to the same. Now, when applying Phase 2 to the lookup-table implementation of AES-128 obtained in Phase 1, the resulting white-box AES-128 implementation consists of five different types of encoded lookup tables [23].

Below, each of the five types of encoded lookup tables is carefully described. It should be noted that this description only discusses how the $\mathbb{F}_2$-linear (external) input and output white-box encodings (i.e., $\mathbb{F}_2$-linear mixing bijections) are applied. Non-linear encodings, typically in the form of single or concatenated random non-linear permutations on $\mathbb{F}_2^4$ (exceptions are highlighted in the description below), need to be applied afterwards in addition to the linear encodings to the input and output of all lookup tables. These non-linear input and output encodings are pairwise annihilating between successive lookup tables; the data-flow of the white-box implementation determines which pairs of non-linear nibble permutations are pairwise annihilating.

In the following, it is assumed that an encoded version of the key-instantiated AES-128 cipher is implemented, i.e., (see (3.4) on p. 64)

$$\mathtt{AES}'_k = \mathtt{OUT} \circ \mathtt{AES}_k \circ \mathtt{IN}^{-1} \ ,$$

where $\mathtt{AES}_k$ denotes the key-instantiated AES-128 cipher and where the external encodings $\mathtt{IN}$ and $\mathtt{OUT}$ are randomly and uniformly selected linear mixing bijections on $\mathbb{F}_2^{128}$. Both external encodings are kept secret in the white-box implementation.

Type II. This type of lookup table concerns rounds $1 \leq r \leq 9$ and encodes the key-dependent tables $\mathtt{TMC}_i^{(r,j)}$ ($0 \leq i,j \leq 3$ and $1 \leq r \leq 9$) to prevent an attacker from extracting the AES round keys from these tables:

The input of each $\mathtt{TMC}_i^{(r,j)}$ table is encoded by (the inverse of) an 8-bit mixing bijection $\left(L_i^{(r,j)}\right)^{-1}$ (represented by a non-singular 8×8 matrix over $\mathbb{F}_2$);

The output of each $\mathtt{TMC}_i^{(r,j)}$ table is encoded by a 32-bit mixing bijection $R^{(r,j)}$ (represented by a non-singular 32×32 matrix over $\mathbb{F}_2$). Observe that $R^{(r,j)}$ lacks the index i as this encoding is required to be the same for all four $\mathtt{TMC}_i^{(r,j)}$ ($0 \leq i \leq 3$) tables associated with each j.

The resulting encoded $\texttt{TMC}_i^{(r,j)}$ tables are denoted by $\mathcal{L}\texttt{-II}_i^{(r,j)}$ ($0 \leq i, j \leq 3$ and $1 \leq r \leq 9$), and map 8 bits to 32 bits. The composition of the $\mathcal{L}\texttt{-II}_i^{(r,j)}$ tables is depicted in Fig. 3.3c.

Type Ib. This type of lookup table concerns the final round and encodes the key-dependent final round T-boxes $T_i^{(10,j)}$ ($0 \leq i, j \leq 3$) to prevent an attacker from extracting the embedded AES round keys:

The input of each T-box $T_i^{(10,j)}$ is encoded by (the inverse of) an 8-bit mixing bijection $\left(L_i^{(10,j)}\right)^{-1}$ (represented by a non-singular 8×8 matrix over $\mathbb{F}_2$);

The output of all T-boxes $T_i^{(10,j)}$ equals the ciphertext, hence the output is encoded by the external output white-box encoding $\texttt{OUT}$. As mentioned above, the external encoding $\texttt{OUT}$ is a 128-bit mixing bijection, represented by a non-singular 128×128 matrix over $\mathbb{F}_2$. By means of *matrix partitioning*, this matrix is split into 16 128×8 submatrices $\texttt{OUT}_l$ ($l = 0, 1, \ldots, 15$) such that the matrix-vector multiplication is decomposed into a XOR of 16 128-bit values. The output of each T-box $T_i^{(10,j)}$ is then encoded by the corresponding submatrix $\texttt{OUT}_{4j+i}$ for $0 \leq i, j \leq 3$.

The resulting encoded $T_i^{(10,j)}$ tables are denoted by $\mathcal{L}\texttt{-Ib}_i^{(10,j)}$ ($0 \leq i, j \leq 3$), and map 8 bits to 128 bits. The composition of the $\mathcal{L}\texttt{-Ib}_i^{(10,j)}$ tables is depicted in Fig. 3.3b.

Type Ia. This type of lookup table annihilates the external input white-box encoding $\texttt{IN}$ (by means of taking its inverse $\texttt{IN}^{-1}$) and introduces the 16 8-bit mixing bijections $L_i^{(1,j)}$ ($0 \leq i, j \leq 3$) to form annihilating pairs with the linear input encodings of the 16 $\mathcal{L}\texttt{-II}_i^{(1,j)}$ ($0 \leq i, j \leq 3$) tables of the first round, while accounting for the $\texttt{ShiftRows}$ operation at the beginning of the first round.

First, by means of *matrix partitioning*, the 128×128 matrix over $\mathbb{F}_2$ representing the 128-bit mixing bijection $\texttt{IN}^{-1}$ is split into 16 128×8 submatrices $\texttt{IN}_l^{-1}$ ($l = 0, 1, \ldots, 15$) such that the matrix-vector multiplication is decomposed into a XOR of 16 128-bit values. Second, the 16 8-bit mixing bijections $L_i^{(1,sr(i,j))}$ ($0 \leq i, j \leq 3$) are concatenated to form the 128-bit mixing bijection

$$L^{(1)} = \Big(L_0^{(1,sr(0,0))}, L_1^{(1,sr(1,0))}, L_2^{(1,sr(2,0))}, L_3^{(1,sr(3,0))}, \\ \ldots, L_0^{(1,sr(0,3))}, L_1^{(1,sr(1,3))}, L_2^{(1,sr(2,3))}, L_3^{(1,sr(3,3))}\Big) \;, \quad (3.6)$$

where the `ShiftRows` function $sr(i, j)$ was introduced in Sect. 2.3.1 (p. 19). Next, $\mathtt{IN}^{-1}$ and $L^{(1)}$ are composed as

$$L^{(1)} \circ \mathtt{IN}_{4j+i}^{-1} \qquad \text{for } 0 \le i, j \le 3 \ . \tag{3.7}$$

Representing each function defined by (3.7) by a lookup table results in 16 $\mathcal{L}\text{-}\mathtt{Ia}_i^{(1,j)}$ $(0 \le i, j \le 3)$ tables, each mapping 8 bits to 128 bits. The composition of the $\mathcal{L}\text{-}\mathtt{Ia}_i^{(1,j)}$ tables is depicted in Fig. 3.3a. Observe that the non-linear input encoding is a random non-linear permutation on $\mathbb{F}_2^8$ instead of a concatenation of two random non-linear permutations on $\mathbb{F}_2^4$. The reason why there is no restriction on the dimension of the non-linear input encoding is the fact that there are no encoded nibble XOR tables (which typically pose the restriction) preceding the $\mathcal{L}\text{-}\mathtt{Ia}_i^{(1,j)}$ tables.

Type III. This type of lookup table handles the compatibility of the applied white-box encodings between consecutive AES rounds r and $r+1$ for $1 \le r \le 9$. This comes down to annihilating the four 32-bit mixing bijections $R^{(r,j)}$ of round r $(0 \le j \le 3)$ by taking their inverses and introducing the 16 8-bit mixing bijections $L_i^{(r+1,j)}$ $(0 \le i, j \le 3)$ to form annihilating pairs with the linear input encodings of the 16 $\mathcal{L}\text{-}\mathtt{II}_i^{(r+1,j)}$ $(0 \le i, j \le 3)$ tables of round $r+1$, while accounting for the `ShiftRows` operation at the beginning of round $r+1$.

For each j with $0 \le j \le 3$, do the following. First, by means of *matrix partitioning*, the 32×32 matrix over $\mathbb{F}_2$ representing the 32-bit mixing bijection $\left(R^{(r,j)}\right)^{-1}$ is split into four 32×8 submatrices $\left(R^{(r,j)}\right)_i^{-1}$ $(i = 0, 1, 2, 3)$ such that the matrix-vector multiplication is decomposed into a XOR of four 32-bit values. Second, the four 8-bit mixing bijections $L_i^{(r+1,sr(i,j))}$ $(0 \le i \le 3)$ are concatenated to form the 32-bit mixing bijection

$$L^{(r+1,j)} = \left(L_0^{(r+1,sr(0,j))}, L_1^{(r+1,sr(1,j))}, L_2^{(r+1,sr(2,j))}, L_3^{(r+1,sr(3,j))}\right) \ . \tag{3.8}$$

Next, $\left(R^{(r,j)}\right)^{-1}$ and $L^{(r+1,j)}$ are composed as

$$L^{(r+1,j)} \circ \left(R^{(r,j)}\right)_i^{-1} \qquad \text{for } 0 \le i, j \le 3 \text{ and } 1 \le r \le 9 \ . \tag{3.9}$$

Representing each function defined by (3.9) by a lookup table results in 16 $\mathcal{L}\text{-}\mathtt{III}_i^{(r,j)}$ $(0 \le i, j \le 3)$ tables per round $(1 \le r \le 9)$, each mapping 8 bits to 32 bits. The composition of the $\mathcal{L}\text{-}\mathtt{III}_i^{(r,j)}$ tables is depicted in Fig. 3.3d.

Table 3.1: The amount of encoded nibble XOR tables appearing in Chow et al.'s white-box AES-128 implementation.

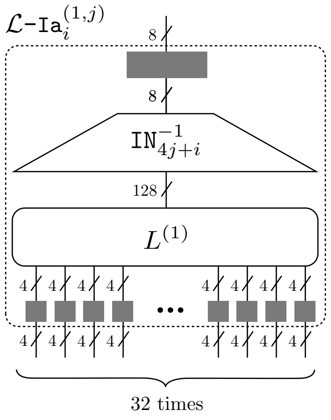
32-bit XOR operations	
$\bigoplus_{i=0}^{3} \mathcal{L}\texttt{-II}_i^{(r,j)}(x_{i,j})$ $(0 \le j \le 3$ and $1 \le r \le 9)$	$3 \cdot 4 \cdot 9 \cdot 8$
$\bigoplus_{i=0}^{3} \mathcal{L}\texttt{-III}_i^{(r,j)}(x_{i,j})$ $(0 \le j \le 3$ and $1 \le r \le 9)$	$3 \cdot 4 \cdot 9 \cdot 8$
128-bit XOR operations	
$\bigoplus_{i,j=0}^{3} \mathcal{L}\texttt{-Ia}_i^{(1,j)}(x_{i,j})$	$15 \cdot 32$
$\bigoplus_{i,j=0}^{3} \mathcal{L}\texttt{-Ib}_i^{(10,j)}(x_{i,j})$	$15 \cdot 32$
Total number of encoded nibble XOR tables	**2688**

Type IV. This type of lookup table handles the XOR operations (both 32-bit and 128-bit) needed due to the decomposition of the matrix-vector multiplication of the `MixColumns` matrix `MC`, the 32-bit mixing bijections $\left(R^{(r,j)}\right)^{-1}$ $(0 \le j \le 3$ and $1 \le r \le 9)$, and the 128-bit mixing bijections `IN` and `OUT`. As pointed out in Sect. 3.2 (Phase 2), each 32-bit or 128-bit XOR operation is represented by 8 or 32 encoded nibble XOR tables $\mathcal{L}'_{\oplus}$ in parallel, respectively. As a result, the Type IV tables, denoted by $\mathcal{L}\texttt{-IV}$, correspond to $\mathcal{L}'_{\oplus}$. The composition of the $\mathcal{L}\texttt{-IV}$ tables, each mapping 8 bits to 4 bits, is depicted in Fig. 3.3e.

The total number of encoded nibble XOR tables appearing in Chow et al.'s white-box AES-128 implementation is summarized in Table 3.1, where it is highlighted how many 32-bit or 128-bit XOR operations need to be performed. Although each encoded nibble XOR table $\mathcal{L}\texttt{-IV}$ internally computes the same result, i.e., the XOR of two different nibble inputs, the application of encodings at their input and output (which are different for each $\mathcal{L}\texttt{-IV}$ table with an overwhelming probability) ensures that all 2688 $\mathcal{L}\texttt{-IV}$ tables are different. However, one still might suggest to implement only a single $\mathcal{L}\texttt{-IV}$ table. This concerns the *re-use of lookup tables*, which is briefly discussed in Chapter 7.

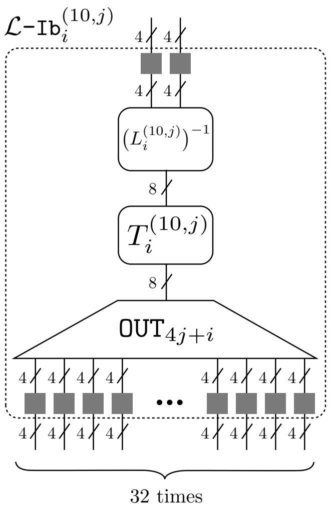
`ShiftRows` operation. As already mentioned in Phase 1, the `ShiftRows` operation at the beginning of each round function is never implemented explicitly, but rather implicitly by including it into the data-flow of the white-box implementation (i.e., providing shifted inputs to the tables) and by accounting for `ShiftRows` when composing encodings together (e.g., (3.6) and (3.8)).

Summary. In order to provide more intuition on how the lookup tables are deployed in the white-box AES-128 implementation, an example white-box implementation of an AES subround for $0 \le j \le 3$ and $1 \le r \le 9$ is depicted in

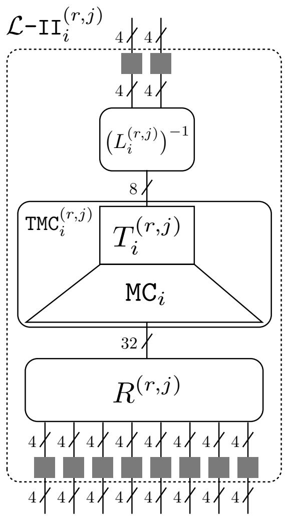


$\mathcal{L}$-$\mathtt{Ia}_i^{(1,j)}$
8
8
$\mathtt{IN}_{4j+i}^{-1}$
128
$L^{(1)}$
4 4 4 4 4 4 4 4
4 4 4 4 4 4 4 4
32 times

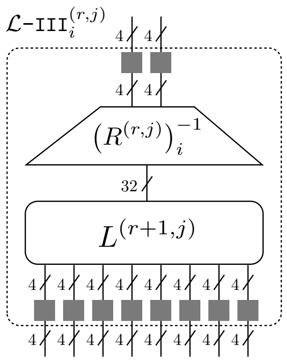
(a) Type Ia: external input encoding $\mathtt{IN}^{-1}$.



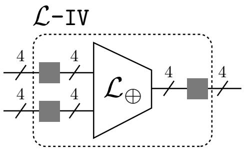
(b) Type Ib: encoded $T_i^{(10,j)}$ tables.



(c) Type II: encoded $\mathtt{TMC}_i^{(r,j)}$ tables.



(d) Type III: compatibility of encodings between consecutive rounds.



(e) Type IV: encoded nibble XOR tables.

Figure 3.3: Five different types of encoded lookup tables of Chow et al.'s white-box AES-128 implementation (the grey boxes indicate the non-linear bijective encodings).

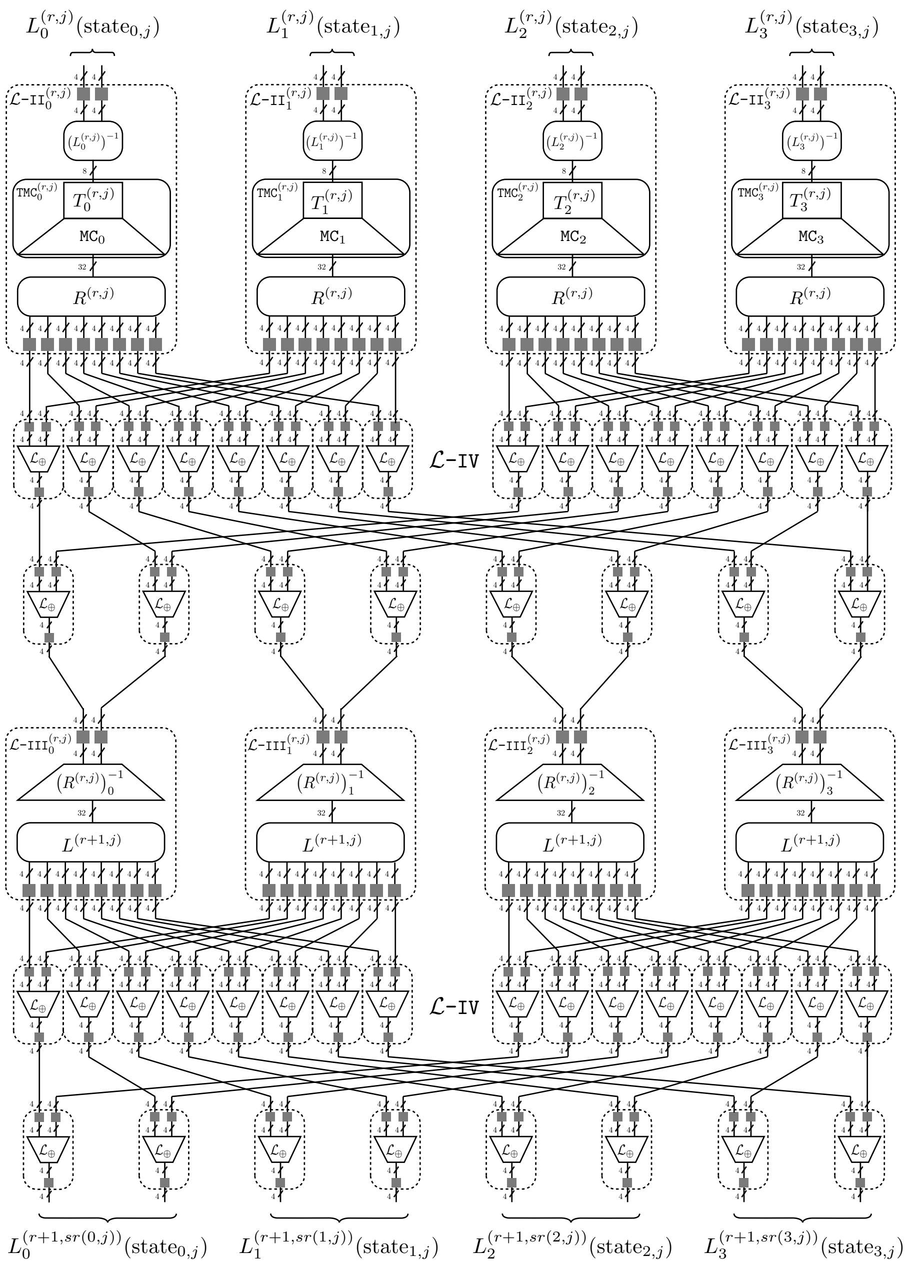

$L_0^{(r,j)}(\text{state}_{0,j})$
$L_1^{(r,j)}(\text{state}_{1,j})$
$L_2^{(r,j)}(\text{state}_{2,j})$
$L_3^{(r,j)}(\text{state}_{3,j})$
$\mathcal{L}$-II$_0^{(r,j)}$
$(L_0^{(r,j)})^{-1}$
TMC$_0^{(r,j)}$
$T_0^{(r,j)}$
MC$_0$
$R^{(r,j)}$
$\mathcal{L}_\oplus$
$\mathcal{L}$-IV
$\mathcal{L}$-III$_0^{(r,j)}$
$(R^{(r,j)})_0^{-1}$
$L^{(r+1,j)}$
$L_0^{(r+1,sr(0,j))}(\text{state}_{0,j})$
$L_1^{(r+1,sr(1,j))}(\text{state}_{1,j})$
$L_2^{(r+1,sr(2,j))}(\text{state}_{2,j})$
$L_3^{(r+1,sr(3,j))}(\text{state}_{3,j})$

Figure 3.4: Chow et al.'s white-box implementation of an AES subround for $0 \leq j \leq 3$ and $1 \leq r \leq 9$ (the grey boxes indicate the non-linear bijective encodings).

Table 3.2: Overall size and performance of Chow et al.'s white-box AES-128 implementation.

Size				**Performance**
Lookup Table			Total Size	# of Table Lookups
#	Type	Size		
144	$\mathcal{L}$-`II`$_i^{(r,j)}$ (8-to-32 bit)	144 kB	752 kB	3008
144	$\mathcal{L}$-`III`$_i^{(r,j)}$ (8-to-32 bit)	144 kB		
16	$\mathcal{L}$-`Ia`$_i^{(1,j)}$ (8-to-128 bit)	64 kB		
16	$\mathcal{L}$-`Ib`$_i^{(10,j)}$ (8-to-128 bit)	64 kB		
2688	$\mathcal{L}$-`IV` (8-to-4 bit)	336 kB		

Fig. 3.4. It is interesting to compare this implementation (which is assumed to provide protection against key-extraction) with the unprotected version depicted in Fig. 3.2b. As one notes that due to the introduction of the 32-bit output mixing bijection $R^{(r,j)}$, the number of tables has been doubled; the second (additional) part ensures that the white-box encodings between consecutive rounds form annihilating pairs.

Table 3.2 gives an overview of the size and performance (expressed in the number of table lookups) of Chow et al.'s white-box AES-128 implementation as specified in [23]. Note that the storage requirement of the encoded nibble XOR tables $\mathcal{L}$-`IV` accounts for $\approx 45\%$ of the total implementation size, while its impact on the overall performance is even more significant, i.e., 2688 out of 3008 ($\approx 89\%$) table lookups. As mentioned later (see Chapter 7), the *re-use of lookup tables* can provide a significant reduction of the implementation size of the XOR operations, however, most likely this introduces security threats. In Chapter 7, a comparison is given with the standard software implementation of AES-128 (see Sect. 2.3.2) as well as with other white-box AES-128 implementation (see the following chapters).

3.3.3 Remark on the Use of Mixing Bijections

The $\mathbb{F}_2$-linear part of the white-box encodings (i.e., the mixing bijections) introduces *diffusion* in the intermediate results of the white-box AES-128 implementation. In the following, *wide* mixing bijections refer to mixing bijections that apply to more than one byte of the AES state simultaneously. As mentioned before, in order to preserve the overall functionality of AES-128, all white-box encodings are pairwise annihilating between successive lookup tables. However, although not explicitly mentioned by Chow et al. [23], instead of annihilating the wide mixing bijections completely, they can be annihilated up to a secret permutation on the indices of the involved bytes. These unknown permutations add confusion to the white-box implementation. In order to maintain the functionality of AES-128, the unknown permutations are accounted for in the data-flow of the white-box implementation. Hence, the additional confusion can be implemented without increasing the size and without decreasing the performance of the white-box implementation. In some sense, it seems natural that the mixing bijections are only annihilated up to some unknown permutations, as otherwise (if they are cancelled completely) their added value would have been questionable.

In the case of Chow et al.'s white-box AES-128 implementation, the relevant wide mixing bijections are (i) the 32-bit mixing bijections $R^{(r,j)}$ ($0 \leq j \leq 3$ and $1 \leq r \leq 9$) and (ii) the 128-bit external input mixing bijection IN. Annihilating the former up to the secret permutations $\Pi^{(r,j)} : (\mathbb{F}_2^8)^4 \to (\mathbb{F}_2^8)^4$ ($0 \leq j \leq 3$ and $1 \leq r \leq 9$) and the latter up to the secret permutation $\Pi^{(\texttt{IN})} : (\mathbb{F}_2^8)^{16} \to (\mathbb{F}_2^8)^{16}$ results in the following confusion included in the white-box implementation: a randomization of the order of the subrounds in an AES round and of the order of the bytes within each subround. The compositions of the Type Ia and Type III tables incorporating these secret permutations are depicted in Fig. 3.5.

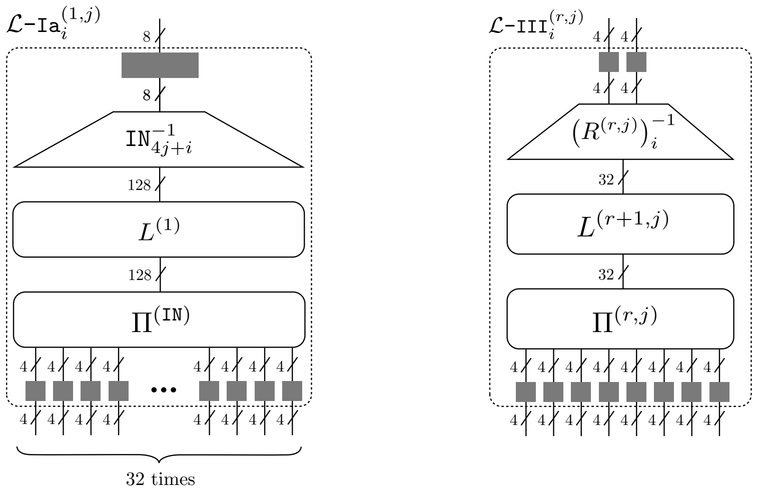

$\mathcal{L}$-$\texttt{Ia}_i^{(1,j)}$
8
8
$\texttt{IN}_{4j+i}^{-1}$
128
$L^{(1)}$
128
$\Pi^{(\texttt{IN})}$
4
32 times
$\mathcal{L}$-$\texttt{III}_i^{(r,j)}$
4
$(R^{(r,j)})_i^{-1}$
32
$L^{(r+1,j)}$
32
$\Pi^{(r,j)}$

Figure 3.5: Generic Type Ia and Type III tables of Chow et al.'s white-box AES-128 implementation (the grey boxes indicate the non-linear bijective encodings).

Table 3.3: Comparison of the overall size and performance of Chow et al.'s white-box implementations of AES-128, AES-192 and AES-256.

AES-k	**Size**	**Performance** (# of table lookups)
AES-128	752 kB	3008
AES-192	864 kB	3456
AES-256	976 kB	3904

3.3.4 Extensions to AES-192 and AES-256

Disregarding the AES key scheduling algorithm, the only differences between AES with different key sizes are the number R of rounds and the number $R+1$ of 128-bit round keys. Since the value of R depends on the key size, $R = 10$, 12 or 14 in the case of AES-128, AES-192 or AES-256, respectively. The AES key scheduling is typically not implemented in the white-box environment (as is also the case for Chow et al.'s white-box AES implementation [23]), i.e., it is assumed that all round keys are directly provided instead of the secret AES key out of which the round keys need to be derived. As a consequence, the techniques presented in Sects. 3.3.1-3.3.3 in order to obtain an example white-box AES-128 implementation can be applied to AES-192 and AES-256 in a straightforward way to obtain example white-box AES-192 and AES-256 implementations, respectively. Table 3.3 lists the storage requirement and performance (expressed in the number of required table lookups) of the white-box implementations (obtained using Chow et al.'s generic white-box techniques) of AES for all three different key size.

3.4 White-Box Security

In order to assess the security of (the implementation of) a block cipher, a specific security notion needs to be taken into account comprising the following two factors: the attacker's model and the cryptanalyst's goal (see Sect. 2.4). This also applies to white-box cryptography that involves the software implementations of key-instantiated block ciphers (for the fixed-key scenario). The *white-box* attack model specifying the capabilities of an attacker in the white-box environment has already been extensively described in Sect. 2.4.5 on p. 29. This section focuses on the objectives of a white-box attacker, that are derived from the classification given in Sect. 2.4.4. As is discussed below, although key-extraction is most often the primary goal, other 'stronger' objectives may be of interest depending on the application in which the white-box implementation is deployed.

3.4.1 White-Box Attacker's Goal

Section 2.4.4 provides a hierarchical classification of the attacker's goals with respect to black-box security (i.e., an attacker complying to the black-box model). However, an attacker complying to the white-box model has more powerful capabilities in addition to a black-box attacker resulting in different interpretations of some existing attacker's goals as well as the introduction of new security requirements for white-box implementations.

With regard to white-box cryptography, two main attacker's goals play a crucial role: *total break* and *global deduction*. Below, the achievement of both goals within the white-box environment is discussed.

Total break. As total break concerns the full recovery of the secret key material, it is crucial to determine what this 'secret key material' includes. Recall that it is common practice in white-box cryptography to implement an encoded version E'_k of an n_b-bit block cipher E_k instantiated with the secret key k, i.e., $E'_k = \texttt{OUT} \circ E_k \circ \texttt{IN}^{-1}$ where the external encodings IN and OUT are bijective mappings on $\mathbb{F}_2^{n_b}$ and are kept secret (see Sect. 3.2 on p. 64). Therefore the resulting white-box implementation is functionally equivalent to E'_k instead of E_k, and takes the encoded plaintext $\texttt{IN}(P)$ as input and outputs the encoded ciphertext $\texttt{OUT}(C)$. With respect to this, next to the secret key k, a *white-box key (WBK)* is defined that comprises both the secret key k and the bijective mappings of the external encodings. Taking the above into account, two different but related definitions of total break with respect to white-box implementations are presented below; these definitions are used throughout this thesis.

Definition 19 (Secret key recovery (KR))**.** *A white-box implementation of a key-instantiated block cipher E_k (or D_k) is called KR-insecure if an attacker extracts the secret key k and furthermore has access to the plaintext P.*

Total break defined as *KR-insecurity* is typically the primary objective of a white-box attacker. The extraction of the secret key k enables the attacker to construct a standard encryption/decryption algorithm instantiated with k. Naturally, if the white-box implementation lacks the application of external encodings, this also allows him to gain access to the plaintext P. However, if the white-box implementation comprises external encodings, the attacker additionally needs access to one of the intermediate states of the block cipher (i.e., the input or output of one of the intermediate rounds) in order to gain access to the plaintext P. Discussion may arise to whether or not it is already sufficient for a white-box attacker to have solely access to the plaintext P without extracting the secret key k. Note that this already closely leans towards global deduction.

The following definition of total break is only meaningful if the white-box implementation includes the application of *bijective* external encodings.

Definition 20 (White-box key recovery (WBKR)). *A white-box implementation of an encoded version of a key-instantiated block cipher E_k (or D_k) is called WBKR-insecure if the attacker extracts the secret key k and the inverse mappings of the applied external encodings.*

Observe that WBKR-insecurity can only be achieved if the applied external encodings are bijective (hence this is a necessary condition on the external encodings). As is shown later, KR-insecurity already allows the attacker to recover the external encodings (if applicable) partially, i.e., only in one way. Hence, if the external encodings are invertible, additional work is required in order to obtain their inverse mappings as well resulting in the WBKR-insecurity. So, naturally, WBKR-insecurity implies KR-insecurity. Consequently, KR-security implies WBKR-security. To summarize:

$$\begin{array}{lcl} \text{KR-security} & \Rightarrow & \text{WBKR-security} \\ \text{KR-insecurity} & \Leftarrow & \text{WBKR-insecurity} \end{array}$$

Note that in the case of white-box implementations, *equivalent keys* (either secret keys or white-box keys) may exist that yield functionally equivalent implementations. Hence, in a broader sense, one should speak about *equivalent secret key recovery* or *equivalent white-box key recovery.* In Chapter 6, it is shown how equivalent keys are extracted from a white-box AES implementation.

Global deduction. Recall from Sect. 2.4.4 that global deduction corresponds to constructing an algorithm that is functionally equivalent to E_k (or D_k) without ever recovering the actual secret key k. This goal becomes particularly interesting in the white-box environment since a white-box attacker is assumed to be in possession of the white-box implementation (without external encodings) that is functionally equivalent to E_k (or D_k). Observe that this relates to the concept of *code-lifting* mentioned earlier. Recall that Chow et al. [23] proposed the application of external encodings in order to preclude code lifting.

Closely related to global deduction is the goal of inverting the white-box implementation: given an implementation functionally equivalent to E_k (or E'_k), construct an implementation functionally equivalent to D_k (or D'_k) without the knowledge of the secret key or the white-box key.

3.4.2 White-Box Security Objectives

The primary objective of white-box cryptography (Def. 14) is to provide protection against total break; this is referred to as *unbreakability* by Delerablée, Lepoint, Paillier and Rivain [39]. In particular, this comes down to achieving KR-security, which is as indicated above a stronger security requirement than WBKR-security. However, Wyseur [103] and Delerablée et al. [39] question the meaningfulness of unbreakability if code-lifting poses a threat.

Next, depending on the application in which a white-box implementation is deployed, there also exist other security objectives (refer to Delerablée et al. [39] for a discussion on these security requirements):

One-wayness provides protection against inverting the white-box implementation. A typical scenario in which one-wayness is required is when a symmetric-key encryption scheme is converted into an asymmetric-key encryption scheme based on white-box cryptography (cf. Joye [52]): the white-box implementation functionally equivalent to E_k (or E'_k) acts as the *public key*, while the secret key k (or the white-box key) acts as the *private key*. Observe that KR-insecurity implies breaking the one-wayness of a white-box implementation without external encodings, and that WBKR-insecurity implies breaking the one-wayness of a white-box implementation with *invertible* external encodings. In those scenarios, one-wayness is a stronger security requirement than unbreakability since a one-way white-box implementation implies KR-security or WBKR-security.

Incompressibility prevents the construction of a functionally equivalent implementation with a significantly smaller memory footprint. The idea behind incompressibility is to discourage attackers to illegally distribute their white-box implementations due to its (large) size.

Traceability relates to the ability to make white-box implementations traceable in order to identify traitors, where a traitor refers to a user illegally distributing his implementation. As pointed out by Michiels [70], relying on the ability to include an arbitrary bit string in a white-box implementation without affecting its functionality and ensuring the integrity of the included string (i.e., resistant against removal or modification), one may include a unique string unambiguously identifying each user.

3.4.3 White-Box Metrics

Security metrics make an effort in quantifying the obtained level of security of a cryptographic primitive. In [24, 23], Chow et al. introduced a few metrics that try to measure the achieved level of white-box security for lookup-table-based white-box implementations of key-instantiated block ciphers: *white-box diversity*, *white-box ambiguity* and *local security*. Below, each of these white-box metrics is discussed with their application to Chow et al.'s white-box AES-128 implementation [23].

White-box diversity and ambiguity. *White-box diversity* (WB-div) counts in how many distinct ways a particular unencoded lookup table can be encoded by randomly and independently chosen permutations (either $\mathbb{F}_2$-linear, $\mathbb{F}_2$-affine or non-linear/non-affine). For key-dependent lookup tables, the variation of the embedded key-material needs to be taken into account. On the other hand, *white-box ambiguity* (WB-amb) captures in how many distinct ways a specific encoded lookup table can be interpreted in terms of composition. Both metrics are closely related by the expression

$$\text{WB-amb}(\mathcal{L}') = \frac{\text{WB-div}(\mathcal{L}')}{\#\text{ of distinct tables of } \mathcal{L}'} \;.$$

These metrics applied above to the level of lookup tables themselves, can also be applied to the level of table-based white-box implementations of a given key-instantiated block cipher, for which both metrics translate into the following: WB-div counts in how many distinct ways a functionally equivalent implementation can be constructed, whereas WB-amb relates to the number of interpretations of distinct candidate keys associated to the same implementation, among which an attacker needs to disambiguate. It should be mentioned that calculating the white-box diversity and ambiguity for entire white-box implementations is a very complex task to complete since relations within the network of encoded lookup tables (such as pairwise annihilating encodings) need to be taken into account.

Local security. In addition to the white-box metrics *diversity* and *ambiguity*, Chow et al. also introduced the term *local security* in [24]. To some extent, local security relates to white-box ambiguity with regard to encoded key-dependent lookup tables.

Definition 21 (Local security [24]). *Let $\mathcal{L}'$ denote an encoded key-dependent lookup table mapping m bits to n bits; $\mathcal{L}'$ is defined as $\mathcal{L}' = g \circ \mathcal{L}_k \circ f$ where k, f and g denote the embedded n_k-bit secret key, the m-bit input encoding (bijective mapping on $\mathbb{F}_2^m$) and the n-bit output encoding (bijective mapping on $\mathbb{F}_2^n$), respectively. Then $\mathcal{L}'$ is called locally secure if, for each possible key value $k' \in \mathbb{F}_2^{n_k}$ with $k' \neq k$, there exists a pair of encodings $(f', g') \neq (f, g)$ such that the table entries of $\mathcal{L}'$ remain invariant, i.e., $\mathcal{L}' = g' \circ \mathcal{L}_{k'} \circ f'$.*

Observe that there exists a close resemblance between (i) the definition of *local security* with respect to lookup tables (see Def. 21), and (ii) the definition of *ideal block ciphers* (see Def. 2). That is, a key-instantiated n_b-bit block cipher is called ideal if it is indistinguishable from a random permutation on n_b bits, i.e., any key value is equally probable. The same reasoning holds for lookup tables, where a key-dependent lookup table is called locally secure if it is indistinguishable from a lookup table instantiated by any other key value.

Example. As pointed out by Chow et al. [23] and Muir [78], the encoded key-dependent Type II tables $\mathcal{L}\text{-}\mathtt{II}_i^{(r,j)}$ ($0 \leq i,j \leq 3$ and $1 \leq r \leq 9$) occurring in Chow et al.'s white-box AES-128 implementation are locally secure: although the non-linear encodings are constructed as concatenated non-linear permutations on $\mathbb{F}_2^4$ such that they are randomly chosen out of all $(2^4!)^2$ possible input encodings and all $(2^4!)^8$ possible output encodings (instead of all possible permutations on $\mathbb{F}_2^8$ and $\mathbb{F}_2^{32}$, respectively), it can be shown that for each possible value of the embedded key byte $\hat{k}_i^{(r,j)}$ (2^8 in total), the encodings can be chosen in such a way that the resulting $\mathcal{L}\text{-}\mathtt{II}_i^{(r,j)}$ table remains invariant. Hence it becomes impossible for an attacker to extract the embedded key byte $\hat{k}_i^{(r,j)}$ by solely observing the $\mathcal{L}\text{-}\mathtt{II}_i^{(r,j)}$ table. The same reasoning holds for the encoded key-dependent Type Ib tables $\mathcal{L}\text{-}\mathtt{Ib}_i^{(10,j)}$ ($0 \leq i,j \leq 3$) as well.

However, although the Type II tables $\mathcal{L}\text{-}\mathtt{II}_i^{(r,j)}$ ($0 \leq i,j \leq 3$ and $1 \leq r \leq 9$) are locally secure with regard to the embedded key byte $\hat{k}_i^{(r,j)}$, other useful information may leak such as the definition of the output encodings if the composition of the tables lacks the application of mixing bijections (see the *S-box frequency attack* in Sect. 3.5.1).

Conclusion. Although the outcome of the various white-box metrics applied to Chow et al.'s white-box AES-128 implementation sounds promising, the cryptanalytic result presented in Sect. 3.5.2 shows the KR-insecurity (and later in Chapter 4 the WBKR-insecurity) of Chow et al.'s implementation up to a very practical level. As a result, the white-box metrics fail in capturing the achieved level of white-box security. It is shown that even though inspecting single isolated locally secure key-dependent lookup tables (*local inspection*) does not yield any key information, inspecting a composition of multiple lookup tables (*global inspection*) may reveal crucial information about the secret white-box encodings, which in turn makes it possible to mount a practical attack in order to extract the secret cryptographic key.

3.5 Cryptanalytic Techniques

3.5.1 Attacks on Weakened Variants

This section describes attacks on weakened variants of Chow et al.'s white-box AES-128 implementation and thereby justifies certain design choices made by Chow et al. [23] in order to improve the security.

S-box frequency attack. This attack highlights the importance of the diffusion effect introduced by the mixing bijections at the input and output of key-dependent lookup tables. In the following, suppose that the key-dependent Type II tables $\mathcal{L}\text{-}\mathtt{II}_i^{(r,j)}$ $(0 \leq i, j \leq 3$ and $1 \leq r \leq 9)$ of Chow et al.'s white-box AES-128 implementation were input and output encoded only by means of concatenated random non-linear permutations on $\mathbb{F}_2^4$ (i.e., they lack the application of mixing bijections). This setting is depicted in Fig. 3.6. As is shown by Chow et al. [23], and subsequently by Muir [78] in a generic context, the output encodings can be fully retrieved by performing a frequency analysis on the AES S-box. Below, a simplified version of the attack is described; for details, refer to [23, 78].

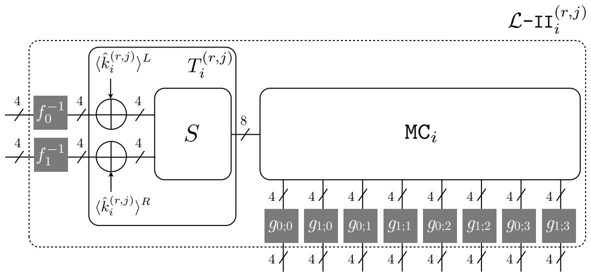

$\mathcal{L}\text{-}\mathtt{II}_i^{(r,j)}$
$\langle \hat{k}_i^{(r,j)} \rangle^L$
$T_i^{(r,j)}$
f_0^{-1}
f_1^{-1}
S
$\mathtt{MC}_i$
$\langle \hat{k}_i^{(r,j)} \rangle^R$
4
8
$g_{0;0}$ $g_{1;0}$ $g_{0;1}$ $g_{1;1}$ $g_{0;2}$ $g_{1;2}$ $g_{0;3}$ $g_{1;3}$

Figure 3.6: Weakened Type II tables of Chow et al.'s white-box AES-128 implementation lacking the input and output mixing bijections and their corresponding diffusion effect.

In the following, it is assumed that there exists no ambiguity about the order of the output nibbles of each $\mathcal{L}\text{-}\mathtt{II}_i^{(r,j)}$ table (Fig. 3.6), i.e., the attacker knows which pair of output nibbles is associated with which coefficient of the embedded `MixColumns` submatrix $\mathtt{MC}_i$. Hence, for each $\mathcal{L}\text{-}\mathtt{II}_i^{(r,j)}$ table, the attacker has access to the four 8-bit bijective mappings

$$\left(g_{0;l}, g_{1;l}\right) \circ \otimes_{mc_l} \circ T_i^{(r,j)} \circ \left(f_0^{-1}, f_1^{-1}\right) \quad \text{for } 0 \leq l \leq 3 \ , \qquad (3.10)$$

where f_0^{-1}, f_1^{-1} denote the two nibble input encodings, and $g_{0;l}, g_{1;l}$ $(0 \leq l \leq 3)$ denote the eight nibble output encodings. The `MixColumns` coefficients mc_l are elements of the set $\{\texttt{01}, \texttt{02}, \texttt{03}\}$. Rewriting (3.10) by including the embedded secret round key byte $\hat{k}_i^{(r,j)}$ into the input encodings results in the mappings

$$\begin{aligned} U_l &= \big(g_{0;l}, g_{1;l}\big) \circ \otimes_{mc_l} \circ S \circ \big(\oplus_{\langle \hat{k}_i^{(r,j)} \rangle^L} \circ f_0^{-1}, \oplus_{\langle \hat{k}_i^{(r,j)} \rangle^R} \circ f_1^{-1} \big) \\ &= \big(g_{0;l}, g_{1;l}\big) \circ S_l \circ \big(f_0'^{-1}, f_1'^{-1}\big) \quad \text{for } 0 \leq l \leq 3 \ , \end{aligned} \tag{3.11}$$

where $\langle x \rangle^L$ and $\langle x \rangle^R$ denote the left and right nibble of $x \in \mathbb{F}_2^8$, respectively.

As U_l and S_l $(0 \leq l \leq 3)$ are bijective mappings on $\mathbb{F}_2^8$, they can each be represented by a 16×16 byte array where the rows and columns are indexed by the left and right nibble of the input bytes, respectively. Next, for each byte entry of the array, a so-called *cell frequency signature* is computed:

Definition 22 (Cell frequency signature [23])**.** *Let L denote an $n \times n$ array of byte entries $L(r, c)$ where $0 \leq r \leq n-1$ (and $0 \leq c \leq n-1$) denote the row (and column) index. The cell frequency signature of an entry $L(r, c)$ is the 64-digit string concatenating the 16-digit nibble-count sequences (i.e., the sequence of the occurrence frequencies of each possible nibble value (0-F)) sorted in descending order of (i) the row r left nibbles, (ii) the row r right nibbles, (iii) the column c left nibbles, and (iv) the column c right nibbles.*

For example, the cell frequency signature of the first entry of the array representing the AES S-box S (i.e., entry $S(0, 0)$) is given by

$$4421111110000000431111111110000032222211100000004222211110000000 \ .$$

Now, the following two observations can be made with regard to each U_l mapping and its associated S_l mapping (see (3.11)) for $0 \leq l \leq 3$:

1. the nibble input encoding $f_0'^{-1}$ (and $f_1'^{-1}$) permutes the order of the rows (and columns) of the array representing S_l;
2. the nibble output encoding $g_{0;l}$ (and $g_{1;l}$) encodes the left (and right) nibble of each byte entry $S_l(r, c)$ with $0 \leq r, c \leq 15$. Observe that although this affects the value of the nibbles, the cell frequency signatures remain invariant (due to the sorting in descending order).

The two observations above imply that all entries of the arrays representing U_l and the associated S_l share the exact same cell frequency signatures, though at different locations in the arrays. Hence, based on finding collisions between cell frequency signatures (abbreviated by `fs`), one can identify matching entries between the arrays representing U_l and S_l, i.e.,

$$\mathtt{fs}\big(U_l(r',c')\big) = \mathtt{fs}\big(S_l(r,c)\big) \text{ where } (r',c') = \big(f_0'^{-1}, f_1'^{-1}\big)(r,c) \ .$$

Further, the relation between matching entries is given by

$$U_l(r',c') = \big(g_{0;l}, g_{1;l}\big)\big(S_l(r,c)\big) \ .$$

As a consequence, by going over all matching entries, the attacker is able to fully retrieve the key-dependent nibble input encodings $f_0'^{-1}, f_1'^{-1}$ and the nibble output encodings $g_{0;l}, g_{1;l}$ $(0 \leq l \leq 3)$. Finally, after the recovery of the nibble output encodings of all $\mathcal{L}\text{-}\mathtt{II}_i^{(r,j)}$ tables $(0 \leq i,j \leq 3)$ of a given round $1 \leq r \leq 8$ and due to the lack of the Type III tables (as they handled the mixing bijections), the attacker is able to retrieve the round key bytes of round $r+1$. For details, refer to Chow et al. [23].

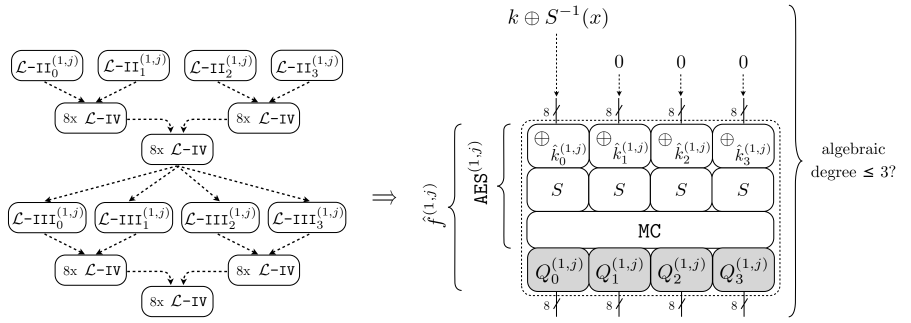
In [78], Muir presents a generic version of the S-box frequency attack in which it is assumed that there exists ambiguity about the order of the output nibbles of each $\mathcal{L}\text{-}\mathtt{II}_i^{(r,j)}$ table, i.e., which output nibbles are associated with which `MixColumns` coefficient and which are left and right output nibbles.

Algebraic degree attack. This attack highlights the importance of the application of external encodings in order to protect the white-box implementation of the outer rounds. In the following, assume a weakened white-box AES-128 implementation that is functionally equivalent to the standard AES-128 cipher (i.e., no external encodings are applied). In this scenario, the input of all Type II tables $\mathcal{L}\text{-}\mathtt{II}_i^{(1,j)}$ $(0 \leq i,j \leq 3)$ of the first round remains unencoded; furthermore, the white-box implementation lacks the Type Ia tables. In the white-box environment, the attacker is able to compose the Type II and Type III tables associated with the first round as depicted in Fig. 3.7 such that he obtains access to the four output encoded AES subrounds $\hat{f}^{(1,j)} : \mathbb{F}_{256}^4 \to (\mathbb{F}_2^8)^4$ $(0 \leq j \leq 3)$ of the first round (disregarding the secret permutations left behind after the annihilation of the mixing bijections), defined by

$$\hat{f}^{(1,j)} = \big(Q_0^{(1,j)}, Q_1^{(1,j)}, Q_2^{(1,j)}, Q_3^{(1,j)}\big) \circ \mathtt{AES}^{(1,j)} \qquad \text{for } 0 \leq j \leq 3 \ ,$$

where each 8-bit bijective output encoding $Q_i^{(1,j)}$ $(0 \leq i,j \leq 3)$ is composed out of an 8-bit mixing bijection and a concatenation of two non-linear permutation on $\mathbb{F}_2^4$.

In [23], Chow et al. showed how to extract the secret round key bytes from each mapping $\hat{f}^{(1,j)}$ $(0 \leq j \leq 3)$ based on the first step of the extended Square



$\mathcal{L}$-II$_0^{(1,j)}$
$\mathcal{L}$-II$_1^{(1,j)}$
$\mathcal{L}$-II$_2^{(1,j)}$
$\mathcal{L}$-II$_3^{(1,j)}$
8x $\mathcal{L}$-IV
$\mathcal{L}$-III$_0^{(1,j)}$
$\mathcal{L}$-III$_1^{(1,j)}$
$\mathcal{L}$-III$_2^{(1,j)}$
$\mathcal{L}$-III$_3^{(1,j)}$
$\Rightarrow$
$k \oplus S^{-1}(x)$
0
8
$\hat{f}^{(1,j)}$
AES$^{(1,j)}$
$\oplus \hat{k}_0^{(1,j)}$
$\oplus \hat{k}_1^{(1,j)}$
$\oplus \hat{k}_2^{(1,j)}$
$\oplus \hat{k}_3^{(1,j)}$
S
MC
$Q_0^{(1,j)}$
$Q_1^{(1,j)}$
$Q_2^{(1,j)}$
$Q_3^{(1,j)}$
algebraic degree ≤ 3?

Figure 3.7: Algebraic degree attack on the first round of a weakened variant of Chow et al.'s white-box AES-128 implementation without external encodings.

attack (see Sect. 2.5.1 on p. 40) with a work factor of 2^{32}. However, the method described below extracts the round key bytes with a lower work factor based on the *algebraic degree* of Boolean functions (to quote Knudsen and Robshaw [58, p. 188]: "the algebraic degree gives a measure of how many input bits might simultaneously have an impact on the value of the output"). The attack discussed below, referred to as the *algebraic degree attack*, is presented by Lepoint et al. [63] as part of a collision-based attack on the white-box AES implementation of Chow et al. (Sect. 3.5.3).

By composition (i.e., a combination of a bijective linear mapping on $\mathbb{F}_2^8$ and a concatenation of two non-linear permutations on $\mathbb{F}_2^4$), each output encoding $Q_i^{(1,j)}$ $(0 \leq i, j \leq 3)$ has an algebraic degree of at most 3. This property can be exploited as follows to extract the key bytes from each mapping $\hat{f}^{(1,j)}$ $(0 \leq j \leq 3)$. To find a round key byte, say $\hat{k}_0^{(1,j)}$, fix the other three input bytes to $\hat{f}^{(1,j)}$ (e.g., to zero), search over all possible 2^8 values of the key byte k and verify if the 8-bit bijective mapping

$$g_k(x) = \hat{f}_0^{(1,j)}\big(k \oplus S^{-1}(x), 0, 0, 0\big)$$

has an algebraic degree of at most 3, where $\hat{f}_0^{(1,j)}$ denotes the first coordinate function of $\hat{f}^{(1,j)}$ (i.e., only consider the first output byte). If $g_k(x)$ has an algebraic degree of at most 3, then $\hat{k}_0^{(1,j)} = k$. Repeat this for all other round key bytes. This methodology is depicted in Fig. 3.7.

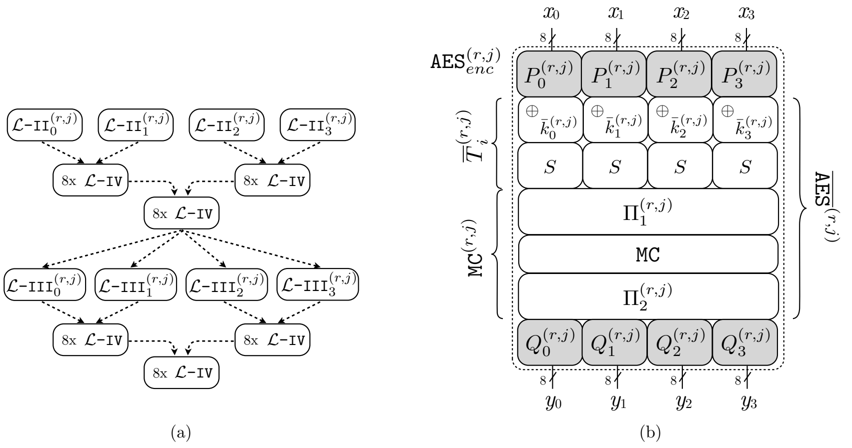
The correctness of the method above relies on the fact that the mapping $S\big(c \oplus S^{-1}(x)\big)$ has an algebraic degree greater than 3 for all non-zero values of c with an overwhelming probability. A method to verify if a bijective mapping on $\mathbb{F}_2^8$ has an algebraic degree of at most 3 is provided by [63, Lemma 2]. The expected work factor to extract all first round key bytes is given by $4 \cdot 4 \cdot 2^7 = 2^{11}$ and is expressed in the number of times one needs to verify the algebraic degree of a bijective mapping on $\mathbb{F}_2^8$.

3.5.2 The BGE Attack

In 2004, two years after the publication of Chow et al.'s white-box AES-128 implementation [23], Billet, Gilbert and Ech-Chatbi [13] presented an algebraic attack (referred to as the BGE attack in the following) on this implementation. The BGE attack successfully extracts the embedded 128-bit AES key with a work factor of 2^{30}, making it a practical attack; the attack can be executed in just a few minutes on a modern PC. As a result, the white-box AES-128 implementation of Chow et al. is *secret key recovery* (KR)-insecure. Observe that this differs from WBKR-insecurity since this additionally requires the extraction of the external encodings by means of the components of their bijective mappings, which is not covered by [13]. However, Chapter 4 presents improvements to the BGE attack as well as an efficient method to extract the external encodings from Chow et al.'s implementation. This result shows that Chow et al.'s white-box AES implementation is WBKR-insecure as well.

Encoded AES Subrounds

In the following text, $P_i^{(r,j)}$ and $Q_i^{(r,j)}$ for $0 \leq i \leq 3$ denote bijective mappings on $\mathbb{F}_2^8$ and are referred to as input and output encodings, respectively. As mentioned before, these encodings are generated randomly and are kept secret in a white-box implementation. In particular, in the case of Chow et al.'s white-box AES implementation [23], the composition of the encodings $P_i^{(r,j)}$ and $Q_i^{(r,j)}$ $(0 \leq i \leq 3)$ is a combination of a bijective linear mapping on $\mathbb{F}_2^8$ (i.e., an 8-bit mixing bijection) and a concatenation of two non-linear permutations on $\mathbb{F}_2^4$. With slight abuse of notation, an input to an AES subround $\mathtt{AES}^{(r,j)}$ (Def. 18) is considered to be an element of $\mathbb{F}_{256}^4$ using the AES polynomial representation in the following definition, and an output of $\mathtt{AES}^{(r,j)}$ is considered to be an element of $(\mathbb{F}_2^8)^4$. Further, in the following definition the permutations $\Pi_i^{(r,j)} : (\mathbb{F}_2^8)^4 \rightarrow (\mathbb{F}_2^8)^4$ $(i = 1, 2)$ for $1 \leq r \leq 9$ and $0 \leq j \leq 3$ permute the order of the input bytes and output bytes of an AES subround, respectively, and $\pi^{(r)} : \{0, 1, 2, 3\} \rightarrow \{0, 1, 2, 3\}$ for $1 \leq r \leq 9$ permutes the order of the four AES subrounds within an AES round. These permutations are randomly chosen and kept secret in a white-box implementation.



$\mathcal{L}$-II$_0^{(r,j)}$
$\mathcal{L}$-II$_1^{(r,j)}$
$\mathcal{L}$-II$_2^{(r,j)}$
$\mathcal{L}$-II$_3^{(r,j)}$
8x $\mathcal{L}$-IV
$\mathcal{L}$-III$_0^{(r,j)}$
$\mathcal{L}$-III$_1^{(r,j)}$
$\mathcal{L}$-III$_2^{(r,j)}$
$\mathcal{L}$-III$_3^{(r,j)}$
(a)
x_0 x_1 x_2 x_3
8
AES$_{enc}^{(r,j)}$
$P_0^{(r,j)}$ $P_1^{(r,j)}$ $P_2^{(r,j)}$ $P_3^{(r,j)}$
$\oplus \bar{k}_0^{(r,j)}$ $\oplus \bar{k}_1^{(r,j)}$ $\oplus \bar{k}_2^{(r,j)}$ $\oplus \bar{k}_3^{(r,j)}$
S
$\overline{T}_i^{(r,j)}$
$\Pi_1^{(r,j)}$
MC
$\Pi_2^{(r,j)}$
MC$^{(r,j)}$
$\overline{\text{AES}}^{(r,j)}$
$Q_0^{(r,j)}$ $Q_1^{(r,j)}$ $Q_2^{(r,j)}$ $Q_3^{(r,j)}$
y_0 y_1 y_2 y_3
(b)

Figure 3.8: Composing Type II, Type III and Type IV tables of Chow et al.'s white-box AES implementation (fig (a)) in order to obtain the encoded AES subrounds $\texttt{AES}_{\text{enc}}^{(r,j)}$ $(1 \leq r \leq 9$ and $0 \leq j \leq 3)$ (fig (b)).

Definition 23 (Encoded AES subround)**.** *The mapping* $\texttt{AES}_{enc}^{(r,j)} : (\mathbb{F}_2^8)^4 \to (\mathbb{F}_2^8)^4$ *for* $1 \leq r \leq 9$ *and* $0 \leq j \leq 3$*, called an encoded AES subround, is defined by*

$$\texttt{AES}_{enc}^{(r,j)} = (Q_0^{(r,j)}, Q_1^{(r,j)}, Q_2^{(r,j)}, Q_3^{(r,j)}) \circ \overline{\texttt{AES}}^{(r,j)} \circ (P_0^{(r,j)}, P_1^{(r,j)}, P_2^{(r,j)}, P_3^{(r,j)}) \ ,$$

where the mapping $\overline{\texttt{AES}}^{(r,j)}$ *is defined by*

$$\Pi_2^{(r,j)} \circ \texttt{AES}^{(r,\pi^{(r)}(j))} \circ \Pi_1^{(r,j)} = \texttt{MC}^{(r,j)} \circ (S,S,S,S) \circ \oplus_{[\bar{k}_i^{(r,j)}]_{0\leq i\leq 3}} \ ,$$

$$\begin{aligned} \textit{with} \quad [\bar{k}_i^{(r,j)}]_{0\leq i\leq 3} &= (\Pi_1^{(r,j)})^{-1}\big([\hat{k}_i^{(r,\pi^{(r)}(j))}]_{0\leq i\leq 3}\big) \\ \textit{and} \quad \texttt{MC}^{(r,j)} &= \Pi_2^{(r,j)} \circ \texttt{MC} \circ \Pi_1^{(r,j)} \ . \end{aligned}$$

Billet et al. [13] showed in their cryptanalysis of Chow et al.'s white-box AES implementation [23] that for rounds $1 \leq r \leq 9$, it is possible to compose Type II and Type III tables as depicted in Fig. 3.4 (and schematically depicted in Fig. 3.8a) such that a white-box attacker has access to the encoded AES subrounds $\texttt{AES}_{\text{enc}}^{(r,j)}$ $(0 \leq j \leq 3)$ of each round r with $1 \leq r \leq 9$ (Def. 23), depicted in Fig. 3.8b.

Recall from Sect. 3.3.3 that, even though the permutations in Def. 23 are not explicitly specified in [23], they are implicitly included in the 32-bit and 128-bit wide mixing bijections. In a practical white-box AES implementation (e.g., Chow et al.'s white-box AES implementation [23]), each wide mixing bijection is annihilated up to a secret permutation on the indices of the involved bytes. This assumption was also made in the BGE attack [13].

As mentioned before, in Chow et al.'s white-box AES implementation, the output encodings $Q_i^{(r-1,j)}$ and input encodings $P_i^{(r,j)}$ for $0 \leq i, j \leq 3$ of successive AES rounds are pairwise annihilating to maintain the functionality of AES. The dataflow of the white-box implementation between successive AES rounds $r-1$ and r (which accounts for the `ShiftRows` step at the beginning for round r as well as for the secret permutations induced by the wide mixing bijections) determines the 16 pairs of output/input encodings which are pairwise annihilating.

BGE Attack

As indicated above, the attacker has access to the encoded AES subrounds $\texttt{AES}_{\text{enc}}^{(r,j)}$ (Fig. 3.8b) for $1 \leq r \leq 9$ and $0 \leq j \leq 3$. In the following, let $(x_0, x_1, x_2, x_3) \in (\mathbb{F}_2^8)^4$ denote the input of $\texttt{AES}_{\text{enc}}^{(r,j)}$; further, let $y_i : (\mathbb{F}_2^8)^4 \to \mathbb{F}_2^8$ for $0 \leq i \leq 3$ denote the coordinate functions of $\texttt{AES}_{\text{enc}}^{(r,j)}$ such that $\texttt{AES}_{\text{enc}}^{(r,j)} = (y_0, y_1, y_2, y_3)$, defined as

$$y_i(x_0, x_1, x_2, x_3) = Q_i^{(r,j)}\Big(\bigoplus_{l=0}^{3} mc_{i,l}^{(r,j)} \otimes \overline{T}_l^{(r,j)}\big(P_l^{(r,j)}(x_l)\big)\Big) \ , \qquad (3.12)$$

for $0 \leq i \leq 3$, where $mc_{i,l}^{(r,j)}$ $(0 \leq i, l \leq 3)$ denote the coefficients of the 'permuted' `MixColumns` matrix $\texttt{MC}^{(r,j)}$ and $\overline{T}_l^{(r,j)} = S \circ \oplus_{\bar{k}_l^{(r,j)}}$ for $0 \leq l \leq 3$.

Next, the BGE attack [13] comprises the following three phases: Phases 1 and 2 retrieve the bytes of the AES round key associated with round r for some r with $2 \leq r \leq 9$, and Phase 3 determines the correct order of the round key bytes and extracts the 128-bit AES key. Each phase is described below. For detailed information about the BGE attack, refer to [13].

The methodology of Phase 1 of the BGE attack is of independent interest (as stated in [13]). In [75], Michiels et al. present a generalization of Phase 1 with regard to white-box implementations of a specific class of key-instantiated block ciphers. This is discussed in Sect. 3.5.4.

Phase 1. The first phase retrieves the encodings $Q_i^{(r,j)}$ $(0 \leq i \leq 3)$ up to an affine part for each encoded AES subround j $(0 \leq j \leq 3)$. Because of the pairwise annihilating property of the encodings between successive rounds, the encodings $P_i^{(r,j)}$ $(0 \leq i, j \leq 3)$ can be retrieved up to an affine part by applying the same technique to the encoded AES subrounds of the previous round.

The removal of the non-affine component of the output encodings $Q_i^{(r,j)}$ $(0 \leq i \leq 3)$ is done as follows. If one fixes the input bytes (x_1, x_2, x_3) of $\texttt{AES}_{\text{enc}}^{(r,j)}$ (Fig 3.8b) to some constant values $(c_1, c_2, c_3) \in (\mathbb{F}_2^8)^3$, then (3.12) becomes

$$y_i(x_0, c_1, c_2, c_3) = Q_i^{(r,j)}\left(mc_{i,0}^{(r,j)} \otimes \overline{T}_0^{(r,j)}\left(P_0^{(r,j)}(x_0)\right) \oplus \beta_i^{c_1,c_2,c_3}\right) , \qquad (3.13)$$

for $0 \leq i \leq 3$, where the constant $\beta_i^{c_1,c_2,c_3} \in \mathbb{F}_2^8$ is dependent on the constant values c_1, c_2, c_3. Now, let $y_i^{c_1,c_2,c_3} : \mathbb{F}_2^8 \rightarrow \mathbb{F}_2^8$ denote the bijective mapping on $\mathbb{F}_2^8$ defined by (3.13) for $0 \leq i \leq 3$. Each mapping $y_i^{c_1,c_2,c_3}$ is entirely determined by a lookup table that is constructed by varying x_0 over $\mathbb{F}_2^8$ and storing the corresponding output values.

Next, varying x_3 over $\mathbb{F}_2^8$ while keeping (x_1, x_2) fixed to (c_1, c_2) results in the fact that $\beta_i^{c_1,c_2,x_3}$ takes all 2^8 values in $\mathbb{F}_2^8$ as well. For each $x_3 \in \mathbb{F}_2^8$, the 8-bit bijective mapping $y_i^{c_1,c_2,x_3}$ is entirely known by means of a lookup table mapping 8 bits to 8 bits. The inverse mapping $(y_i^{c_1,c_2,x_3})^{-1}$ is obtained by inverting the corresponding lookup table. Now, by composing $y_i^{c_1,c_2,c_3}$ and $(y_i^{c_1,c_2,x_3})^{-1}$, one gets the following set of 2^8 bijective mappings on $\mathbb{F}_2^8$ for $0 \leq i \leq 3$:

$$\mathcal{S}_i = \{y_i^{c_1,c_2,c_3} \circ (y_i^{c_1,c_2,x_3})^{-1}\} = \left\{Q_i^{(r,j)} \circ \oplus_\beta \circ \left(Q_i^{(r,j)}\right)^{-1}\right\} , \qquad (3.14)$$

where $\beta = \beta_i^{c_1,c_2,c_3} \oplus \beta_i^{c_1,c_2,x_3}$ takes all 2^8 values in $\mathbb{F}_2^8$ by varying x_3 over $\mathbb{F}_2^8$. Each of the 2^8 bijective mappings (one for each possible β value) within each set $\mathcal{S}_i$ for $0 \leq i \leq 3$ is completely determined by a lookup table.

Given each set $\mathcal{S}_i$ $(0 \leq i \leq 3)$ as defined in (3.14), Billet et al. (see [13, Theorem 1]) are able to compute the non-affine component of the white-box output encodings $Q_i^{(r,j)}$ $(0 \leq i \leq 3)$, denoted by $\widetilde{Q}_i^{(r,j)}$, such that there exists an unknown affine mapping $\widehat{Q}_i^{(r,j)}$ so that $\widetilde{Q}_i^{(r,j)} = Q_i^{(r,j)} \circ \widehat{Q}_i^{(r,j)}$. As a result, all encodings $Q_i^{(r,j)}$ $(0 \leq i \leq 3)$ in Fig 3.8b can be made affine (still unknown) by composing them with the corresponding $\widetilde{Q}_i^{(r,j)}$ as follows:

$$\left(\widetilde{Q}_i^{(r,j)}\right)^{-1} \circ Q_i^{(r,j)} = \left(\widehat{Q}_i^{(r,j)}\right)^{-1} \circ \left(Q_i^{(r,j)}\right)^{-1} \circ Q_i^{(r,j)} = \left(\widehat{Q}_i^{(r,j)}\right)^{-1} .$$

Phase 2. The second phase assumes that all encodings of an encoded AES round are affine mappings (as the other parts have been retrieved in Phase 1). Phase 2 first retrieves the affine encodings $Q_i^{(r,j)}$ $(0 \leq i \leq 3)$ for each encoded AES subround $\texttt{AES}_{\text{enc}}^{(r,j)}$ $(0 \leq j \leq 3)$. During this process, the key-dependent affine mappings $\widetilde{P}_i^{(r,j)}(x) = P_i^{(r,j)}(x) \oplus \bar{k}_i^{(r,j)}$ $(0 \le i, j \le 3)$ are obtained as well. As in Phase 1, the affine encodings $P_i^{(r,j)}$ $(0 \le i, j \le 3)$ are retrieved by applying the same technique to the encoded AES subrounds of the previous round. This enables the attacker to compute the round key bytes $\bar{k}_i^{(r,j)} = \widetilde{P}_i^{(r,j)}(0) \oplus P_i^{(r,j)}(0)$ for $0 \le i, j \le 3$.

Phase 3. After Phases 1 and 2, the values of the round key bytes $\bar{k}_i^{(r,j)}$ $(0 \le i, j \le 3)$ of round r with $2 \le r \le 8$ are known. However, the order of the round key bytes associated with each subround and the order of the four subrounds are still unknown. Therefore, Billet et al. [13] proposed a third phase that (i) retrieves the round key bytes of round $r+1$ as well by the application of Phases 1 and 2, and (ii) uses the fact that the round key bytes of rounds r and $r+1$ are related to each other via both the data-flow of the white-box implementation and the AES key scheduling algorithm to determine the correct order of the round key bytes. Note that Billet et al. [13] does not provide an explicit description of such a method. However, in [36], we presented an efficient method for Phase 3 of the BGE attack, which is described in Chapter 4. Finally, assuming that the AES variant with a 128-bit key is used, the attacker can use the property of the AES key scheduling algorithm that the AES key can be computed if one of the round keys is known.

Work factor of the BGE attack. In [13], Billet et al. claim that the work factor associated with the three phases of the BGE attack is around 2^{30}. Below, the work factor of each phase is listed; a detailed explanation along with improvements is presented in Chapter 4.

Phase 1	$3 \cdot 4 \cdot 4 \cdot 2^{24} < 2^{30}$
Phase 2	$3 \cdot 4 \cdot 4 \cdot 2^{15} \cdot 2^{8} < 2^{29}$
Phase 3	no work factor given
Total work factor	2^{30}

Apart from extracting the secret AES key k, the BGE attack also enables an attacker to gain access to the plaintext P as explained in following. After Phases 2 and 3, the input encodings $P_i^{(r,j)}$ $(0 \le i, j \le 3)$ of the encoded round r with $2 \le r \le 8$ as well as their correct order are fully retrieved, which allows the attacker to have access the unencoded 16-byte input of the AES round r, denoted by $\textsc{state}^{(r)}$. Combined with the ability to construct a standard AES decryption algorithm instantiated with the extracted key k, the attacker can partially decrypt $\textsc{state}^{(r)}$ to obtain the plaintext P. This is explained into more detail in Chapter 4. As a work factor of 2^{30} clearly shows the practicality of the BGE attack, the white-box AES implementation of Chow et al. is KR-insecure.

Vulnerabilities Exploited by the BGE Attack

In order for the BGE attack to be successful, certain vulnerabilities of Chow et al.'s white-box AES implementation have been exploited. These vulnerabilities are weaknesses of the white-box AES design that are available to an attacker in the white-box setting, and which the attacker can use to his advantage to mount an efficient white-box attack. Three main vulnerabilities can be identified which relate to certain properties of the white-box implementation and to the internal structure of AES. The identification of these vulnerabilities is used as a starting point in generalizing the BGE attack (see Sect. 3.5.4).

Vulnerability 1. *In Chow et al.'s white-box AES implementation [23], for each byte $state_{i,j}$ $(0 \leq i, j \leq 3)$ of the input state of a standard AES round r $(1 \leq r \leq 10)$, the attacker has access to an encoded version $g_i^{(r,j)}\big(state_{i,j}\big)$ of $state_{i,j}$. Each encoding $g_i^{(r,j)}$ is a secret fixed bijective mapping on $\mathbb{F}_2^8$, composed of a mixing bijection on $\mathbb{F}_2^8$ together with two concatenated non-linear permutations on $\mathbb{F}_2^4$.*

Vulnerability 1 states that the attacker is able to observe the 16 input and output bytes of AES rounds $1 \leq r \leq 9$ in a fixed encoded form.

Vulnerability 2. *In Chow et al.'s white-box AES implementation [23], for each encoded AES subround $\texttt{AES}_{enc}^{(r,j)}$ $(1 \leq r \leq 9$ and $0 \leq j \leq 3)$, the attacker has access to the following four bijective mappings on $\mathbb{F}_2^8$ for $0 \leq i \leq 3$ (hence 16 mappings in total), where $c_i \in \mathbb{F}_2^8$ $(0 \leq i \leq 3)$ denote constant values:*

$$\begin{array}{rcl} y_i^{c_1,c_2,c_3} : \mathbb{F}_2^8 \to \mathbb{F}_2^8 : x_0 & \mapsto & y_i(x_0, c_1, c_2, c_3)\ , \\ y_i^{c_0,c_2,c_3} : \mathbb{F}_2^8 \to \mathbb{F}_2^8 : x_1 & \mapsto & y_i(c_0, x_1, c_2, c_3)\ , \\ y_i^{c_0,c_1,c_3} : \mathbb{F}_2^8 \to \mathbb{F}_2^8 : x_2 & \mapsto & y_i(c_0, c_1, x_2, c_3)\ , \\ y_i^{c_0,c_1,c_2} : \mathbb{F}_2^8 \to \mathbb{F}_2^8 : x_3 & \mapsto & y_i(c_0, c_1, c_2, x_3)\ . \end{array}$$

Recall that $(x_0, x_1, x_2, x_3) \in (\mathbb{F}_2^8)^4$ denotes the input of $\texttt{AES}_{enc}^{(r,j)}$ and that y_i $(0 \leq i \leq 3)$ denote the coordinate functions of $\texttt{AES}_{enc}^{(r,j)}$. Each bijective mapping on $\mathbb{F}_2^8$ is determined by means of a lookup table.

Vulnerability 2 is the result of combining Vulnerability 1 with the inherent byte-oriented structure of each AES subround $\texttt{AES}^{(r,j)}$ (Def. 18), i.e., the fact that `AddRoundKey` and `SubBytes` are bijective byte operations and `MixColumns` is represented by a 4×4 circulant MDS (Maximum Distance Separable) matrix over $\mathbb{F}_{256}$. The BGE attack mainly exploits Vulnerability 2; e.g., it enables the attacker to successfully build the sets (3.14), which in fact only requires two out of four bijective mappings for each i $(0 \leq i \leq 3)$.

Table 3.4: Comparison of the estimated overall work factor of the BGE attack on Chow et al.'s white-box implementations of AES-128, AES-192 and AES-256.

AES-k	**Work factor of the BGE attack**
AES-128	2^{30}
AES-192	2^{30}
AES-256	2^{31}

Vulnerability 3. *The specification of AES is publicly known up to the secret key; this includes the AES S-box and the* `MixColumns` *coefficients.*

BGE Attack on White-Box AES-192 and AES-256 Implementations?

In the following, it is assumed that the white-box AES-192 and AES-256 implementations are obtained in the same way as Chow et al.'s white-box AES-128 implementation [23] (see Sect. 3.3.4). As a consequence, the attacker has access to the encoded AES subrounds $\texttt{AES}_{\text{enc}}^{(r,j)}$ for $0 \leq i \leq 3$ and $1 \leq r < R$, where $R = 12$ for AES-192 and $R = 14$ for AES-256. Hence Phases 1 and 2 of the BGE attack can be applied to retrieve the round key bytes associated with round r for some r with $2 \leq r < R$. As described in Phase 3 of the BGE attack, due to the ambiguity about the order of the round key bytes, the round key bytes of consecutive rounds need to be retrieved which are related to each other via both the data-flow of the white-box implementation and the AES key scheduling algorithm to determine the correct of the round key bytes. Now, the AES key schedule [69, 31] is an iterative algorithm and involves n_K successive 128-bit round keys at each iteration, where $n_K = 2$ for AES-128, $n_K = 3$ for AES-192 and $n_K = 4$ for AES-256. As a result, Phase 3 of the BGE attack requires that the round key bytes of three or four consecutive rounds are retrieved in the case of AES-192 or AES-256, respectively. The estimated overall work factors of the BGE attack are listed in Table. 3.4. Observe that the BGE attack has similar overall work factors for the three variants of AES.

3.5.3 An Attack Exploiting Internal Collisions

Lepoint and Rivain [62, 63] proposed a new attack on the white-box AES implementation of Chow et al. by exploiting collisions in the output bytes of the four encoded AES subrounds of the first round. This attack, referred to as the *collision attack* in the following, extracts the embedded AES key with a work factor of 2^{22}. However, unlike the BGE attack, it is assumed that the attacker knows the order of the bytes of the intermediate AES results in the white-box implementation. Consequently, the collision attack assumes a composition of the encoded AES subrounds lacking the secret byte permutations, given by the following definition.

Definition 24 (Encoded AES subround without byte permutations)**.** *The mapping $f^{(r,j)} : (\mathbb{F}_2^8)^4 \to (\mathbb{F}_2^8)^4$ for $1 \le r \le 9$ and $0 \le j \le 3$, called an encoded AES subround without byte permutations, is defined by*

$$f^{(r,j)} = (Q_0^{(r,j)}, Q_1^{(r,j)}, Q_2^{(r,j)}, Q_3^{(r,j)}) \circ \mathit{AES}^{(r,j)} \circ (P_0^{(r,j)}, P_1^{(r,j)}, P_2^{(r,j)}, P_3^{(r,j)}) \ ,$$

As mentioned above, the collision attack considers the four encoded AES subrounds of the first round, given by $f^{(1,j)}$ for $0 \le j \le 3$. For each $f^{(1,j)}$, let $f_i^{(1,j)}$ denote the i-th coordinate function of $f^{(1,j)}$ (i.e., only consider the i-th output byte) for $i = 0, 1, 2, 3$ such that $f^{(1,j)} = \big(f_0^{(1,j)}, f_1^{(1,j)}, f_2^{(1,j)}, f_3^{(1,j)}\big)$. Further, in [63], the 8-bit bijective mappings $S_i^{(1,j)}$ defined as $S_i^{(1,j)}(x) = S\big(\hat{k}_i^{(1,j)} \oplus P_i^{(1,j)}(x)\big)$ $(0 \le i, j \le 3)$ are introduced such that

$$f^{(1,j)} = \big(Q_0^{(1,j)}, Q_1^{(1,j)}, Q_2^{(1,j)}, Q_3^{(1,j)}\big) \circ \mathtt{MC} \circ \big(S_0^{(1,j)}, S_1^{(1,j)}, S_2^{(1,j)}, S_3^{(1,j)}\big) \ ,$$

for $0 \le j \le 3$. The collision attack as described in [62, 63] comprises the following two phases:

Phase 1. The first phase retrieves the functions $S_i^{(1,j)}$ $(0 \le i, j \le 3)$ by exploiting collisions in the output bytes of $f^{(1,j)}$ $(0 \le j \le 3)$, i.e., collisions in the output of the coordinate functions $f_i^{(1,j)}$ $(0 \le i, j \le 3)$. Due to the fact that the secret output encodings $Q_i^{(1,j)}$ $(0 \le i, j \le 3)$ are fixed bijective mappings on $\mathbb{F}_2^8$, a collision in the encoded output byte $Q_i^{(1,j)}(x)$ corresponds with a collision in the unencoded output byte x as well. For example, by looking for a collision of the form $f_0^{(1,j)}(\alpha, 0, 0, 0) = f_0^{(1,j)}(0, \beta, 0, 0)$, one can rewrite the equation as

$$Q_0^{(1,j)}\big(\mathsf{02} \otimes S_0^{(1,j)}(\alpha) \oplus \mathsf{03} \otimes S_1^{(1,j)}(0) \oplus c\big) = \\ Q_0^{(1,j)}\big(\mathsf{02} \otimes S_0^{(1,j)}(0) \oplus \mathsf{03} \otimes S_1^{(1,j)}(\beta) \oplus c\big) \ ,$$

where $c = S_2^{(1,j)}(0) \oplus S_3^{(1,j)}(0)$, that eventually yields

$$\mathsf{02} \otimes S_0^{(1,j)}(\alpha) \oplus \mathsf{03} \otimes S_1^{(1,j)}(0) = \mathsf{02} \otimes S_0^{(1,j)}(0) \oplus \mathsf{03} \otimes S_1^{(1,j)}(\beta) \ . \qquad (3.15)$$

Due to the bijectiveness of the $S_i^{(1,j)}$ functions, there exists a total of 256 pairs (α, β) satisfying (3.15), among which the trivial solution $(\alpha, \beta) = (0, 0)$.

In [63, Sect. 4.1], it is discussed how to extract the functions $S_0^{(1,j)}$ and $S_1^{(1,j)}$ by exploitng collisions of the form

$$f_i^{(1,j)}(\alpha,0,0,0) = f_i^{(1,j)}(0,\beta,0,0)$$

for $i = 0,1,2,3$. With the knowledge of $S_0^{(1,j)}$, the functions $S_2^{(1,j)}$ and $S_3^{(1,j)}$ can then be found similarly by exploiting collision of the form

$$f_i^{(1,j)}(\alpha,0,0,0) = f_i^{(1,j)}(0,0,\beta,0) \quad \text{and} \quad f_i^{(1,j)}(\alpha,0,0,0) = f_i^{(1,j)}(0,0,0,\beta)$$

for $i = 0,1,2,3$, respectively. By repeating this process for all four subrounds, i.e., for $j = 0,1,2,3$, all functions $S_i^{(1,j)}$ $(0 \le i,j \le 3)$ can be extracted.

Phase 2. The second phase extracts the round key bytes $\hat{k}_i^{(2,j)}$ $(0 \le i,j \le 3)$ of the second round. Observe that due to the fact that it is assumed that the attacker knows the order of the bytes of the intermediate AES results in the white-box implementation, it suffices to extract only a single round key since there exists no ambiguity about the order of the round key bytes. And, as mentioned before, the secret key of AES-128 can be computed from the knowledge of a single round key.

After Phase 1, all $S_i^{(1,j)}$ $(0 \le i,j \le 3)$ functions are retrieved. This allows the attacker to obtain all secret output encodings $Q_i^{(1,j)}$ $(0 \le i,j \le 3)$ of the first round by computing

$$\left(Q_0^{(1,j)}, Q_1^{(1,j)}, Q_2^{(1,j)}, Q_3^{(1,j)}\right) = f^{(1,j)} \circ \left(S_0^{(1,j)}, S_1^{(1,j)}, S_2^{(1,j)}, S_3^{(1,j)}\right)^{-1} \circ \mathtt{MC}^{-1} \ ,$$

for $0 \le j \le 3$. Further, since the output encodings $Q_i^{(1,j)}$ of the first round and the input encodings $P_i^{(2,j)}$ of the second round for $0 \le i,j \le 3$ are pairwise annihilating, all input encodings $P_i^{(2,j)}$ $(0 \le i \le 3)$ of the encoded AES subrounds $f^{(2,j)}$ $(0 \le j \le 3)$ can be removed. The resulting mappings $\hat{f}^{(2,j)} : (\mathbb{F}_2^8)^4 \to (\mathbb{F}_2^8)^4$ are given by

$$\hat{f}^{(2,j)} = \left(Q_0^{(2,j)}, Q_1^{(2,j)}, Q_2^{(2,j)}, Q_3^{(2,j)}\right) \circ \mathtt{AES}^{(2,j)} \ ,$$

for $0 \le j \le 3$. Next, in order to extract the round key bytes $\hat{k}_i^{(2,j)}$ $(0 \le i \le 3)$ from each mapping $\hat{f}^{(2,j)}$ $(0 \le j \le 3)$, the same technique is used as in the *algebraic degree attack* (Sect. 3.5.1, p. 87), i.e., by exploiting the fact that each output encoding $Q_i^{(2,j)}$ has an algebraic degree of at most 4. E.g., in order to extract the round key byte $\hat{k}_0^{(2,j)}$, verify whether the 8-bit bijective mapping

$$g_k(x) = \hat{f}_0^{(2,j)}\left(k \oplus S^{-1}(x), 0, 0, 0\right)$$

has an algebraic degree of at most 4 for each possible guess $k \in \mathbb{F}_2^8$, where $\hat{f}_0^{(2,j)}$ denotes the first coordinate function of $\hat{f}^{(2,j)}$. For details, refer to [63, Sect. 4.2].

Work factor of the collision attack. In [63, Sect. 4.3], Lepoint and Rivain argue that the work factor of the collision attack is dominated by the work factor of Phase 1 and is given by 2^{22}. In [62], Lepoint and Rivain claim to have extended their attack to the generic case, i.e., assuming that the order of the bytes of the intermediate AES results in the white-box implementation are secretly randomized. For details about the (generic) collision attack, refer to [62, 63].

3.5.4 Generic White-Box Attack of Michiels et al.

The BGE attack [13] was specifically designed for cryptanalyzing the white-box AES implementation of Chow et al. [23]. Throughout the attack, specific properties of AES are exploited such as the AES S-box and the `MixColumns` coefficients (see Vulnerability 3), hence the attack cannot be extended trivially to other SPN block ciphers. However, as already pointed out by Billet et al. [13], the technique of Phase 1 of the BGE attack (i.e., the removal of the non-affine component of the secret white-box encodings) is of independent interest and may be applied to lookup-table-based white-box implementations of other SPN block ciphers if some requirements on both the white-box implementation and the cipher itself are met. This has been covered in [75], where Michiels, Gorissen and Hollmann present an algebraic attack on white-box implementations of a generic class of SPN block ciphers. The primary goal of Michiels et al.'s attack is the secret key recovery (KR, and not WBKR), as was also the case for the BGE attack. Below, before describing the actual attack, first the specifications of the generic class of SPN ciphers and the properties of their white-box implementations are discussed that need to be satisfied in order for the attack to succeed.

Generic Class of SPN Block Ciphers

The generic class of SPN ciphers (Def. 5) considered by Michiels et al. [75] is the family of key-alternating Substitution-Linear Transformation (SLT) ciphers as defined by the following definition.

Definition 25 (Substitution-Linear Transformation (SLT) cipher [75, Def. 1]). *A cipher is called an n-bit SLT cipher if it comprises R rounds with $R \geq 1$, where each round $1 \leq r \leq R$ is a bijective mapping on $\mathbb{F}_2^n$ operating on s input words $x_i \in \mathbb{F}_2^m$ $(1 \leq i \leq s)$ in parallel with $n = s \cdot m$, and consists of the following three consecutive operations: (i) a XOR with a round key $k^{(r)} \in \mathbb{F}_2^n$, (ii) the confusion layer comprising s (different) S-boxes $S_i^{(r)}$ $(1 \leq i \leq s)$ in parallel where each S-box $S_i^{(r)}$ is defined as a non-linear bijective mapping on*

$\mathbb{F}_2^m$, (iii) the diffusion layer comprising an $n \times n$ non-singular matrix $M^{(r)}$ over $\mathbb{F}_2$. Apart from the secret round keys, all other components are included in the cipher's specification.

Observe that many common block ciphers satisfy Def. 25, such as AES [69] and Serpent [7] (one of the five AES finalists). Additionally to Def. 25, Michiels et al. state a mild but necessary condition on the diffusion layer, which is captured by the following property.

Property 2 (Double surjective mappings [75, Def. 3])**.** *Let $(z_1, z_2, \ldots, z_s)$ and $(y_1, y_2, \ldots, y_s)$ with $z_i, y_i \in \mathbb{F}_2^m$ $(1 \leq i \leq s)$ and $n = s \cdot m$ denote the n-bit input and output of the $n \times n$ non-singular diffusion matrix $M^{(r)}$ over $\mathbb{F}_2$, respectively. Now, for each output word $y_j \in \mathbb{F}_2^m$ $(1 \leq j \leq s)$, there exists two disjoint subsets U_j, V_j of input words z_i such that the mappings defined by $U_j \mapsto y_j$ and $V_j \mapsto y_j$ through $M^{(r)}$ (while fixing the uninvolved input words to a constant) are both surjective on $\mathbb{F}_2^m$.*

As mentioned by Michiels et al. [75], Property 2 is met by any $n \times n$ MDS matrix over $\mathbb{F}_2$. Furthermore, MDS matrices are preferably selected as diffusion matrices in block cipher designs because of their good diffusion properties. Note that AES satisfies Def. 25 combined with Property 2.

White-box Implementations of SLT Ciphers

Concerning the white-box implementations of the family of SLT ciphers defined by Def. 25 and Property 2, Michiels et al. [75] discuss certain properties that are necessary for the attack to succeed.

Property 3. *In the white-box implementation, for each input word $x_i \in \mathbb{F}_2^m$ $(1 \leq i \leq s)$ of each round $1 \leq r \leq R$, the attacker has access to an encoded version $f_i^{(r)}(x_i)$ of x_i. Each encoding $f_i^{(r)}$ is a fixed random permutation on $\mathbb{F}_2^m$ and kept secret in the white-box implementation.*

Property 4. *For each fixed encoded output word $f_j^{(r+1)}(y_j) \in \mathbb{F}_2^m$ $(1 \leq j \leq s)$ of each round $1 \leq r < R$, there exists two disjoint subsets $\overline{U}_j, \overline{V}_j$ of fixed encoded input words $f_i^{(r)}(x_i)$ such that the mappings defined by $\overline{U}_j \mapsto f_j^{(r+1)}(y_j)$ and $\overline{V}_j \mapsto f_j^{(r+1)}(y_j)$ through the white-box implementation of round r (while fixing the uninvolved encoded input words to a constant) are both surjective on $\mathbb{F}_2^m$. These surjective mappings on $\mathbb{F}_2^m$ can be made bijective by [75, Theorem 1].*

Observe that Property 4 is a result from combining Def. 25 and Properties 2 and 3. Hence, the disjoint subsets $\overline{U}_j, \overline{V}_j$ of Property 4 relate to the disjoint subsets U_j, V_j of Property 2, as is depicted in Fig. 3.9.

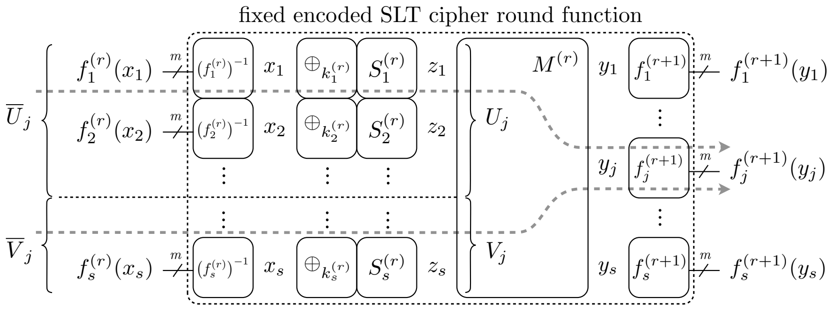


Figure 3.9: Illustration of Michiels et al.'s attack: the existence of double surjective mappings on $\mathbb{F}_2^m$ in the white-box implementation of an SLT cipher.

Note that Properties 2 and 3 correspond to generalizations of Vulnerabilities 1 and 2 (identified in Sect. 3.5.2 with regard to the BGE attack), respectively. Therefore, these properties ensure that a generalized version of Phase 1 of the BGE attack can be mounted against the white-box implementations.

Michiels et al.'s Attack

For the family of SLT ciphers defined by Def. 25 and Property 2, whose lookup-table-based white-box implementations (obtained by for example applying the generic white-box techniques of Chow et al. described in Sect. 3.2) satisfy Property 3 and consequentially also Property 4, Michiels et al. [75] presented an algebraic attack enabling an attacker to extract the n-bit round key $k^{(r)}$ of any round $1 < r < R$, that eventually yields the secret embedded n-bit cryptographic key through the application of the inverse key scheduling algorithm (if applicable). The attack consists of three phases, which are briefly highlighted below. Note that unlike the BGE attack, Michiels et al.'s attack assumes that the order of the m-bit words of the intermediate results in the white-box implementation is known to the attacker (see [75, Property 1]). Additionally, the attack assumes that the cipher's specification is publicly known. For detailed information about Michiels et al.'s attack, refer to [75].

The following three phases need to be applied to any round r with $1 < r < R$ of the white-box implementation, where $\left(f_1^{(r)}(x_1), f_2^{(r)}(x_2), \ldots, f_s^{(r)}(x_s)\right)$ and $\left(f_1^{(r+1)}(y_1), f_2^{(r+1)}(y_2), \ldots, f_s^{(r+1)}(y_s)\right)$ denote the n-bit input and output of the fixed encoded round function, respectively (see Fig. 3.9).

Phase 1. The first phase retrieves the input encodings $\left(f_i^{(r)}\right)^{-1}$ and output encodings $f_i^{(r+1)}$ $(1 \leq i \leq s)$ up to an affine part. This phase is a generalization of Phase 1 of the BGE attack, i.e., the following sets of 2^m bijective mappings on $\mathbb{F}_2^m$ are constructed for $1 \leq i \leq s$:

$$\mathcal{S}_i = \{f_i^{(r+1)} \circ \oplus_\beta \circ \left(f_i^{(r+1)}\right)^{-1} \mid \beta \in \mathbb{F}_2^m\} \ , \tag{3.16}$$

where each of the 2^m bijective mappings within each set $\mathcal{S}_i$ for $1 \leq i \leq s$ is completely determined by a lookup table. Property 4 enables the construction of these sets $\mathcal{S}_i$. Next, given the sets $\mathcal{S}_i$ (3.16), the attacker can obtain $f_i^{(r+1)}$ $(1 \leq i \leq s)$ up to an unknown affine component by applying the following generalized version of [13, Theorem 1].

Theorem 1 (Generalized version of [13, Theorem 1])**.** *Given a set of functions $\mathcal{S} = \{Q \circ \oplus_\beta \circ Q^{-1} \mid \beta \in \mathbb{F}_2^m\}$ given by values, where Q is a permutation of $\mathbb{F}_2^m$ and $\oplus_\beta$ is the translation by β in $\mathbb{F}_2^m$, one can construct a particular solution $\widetilde{Q}$ such that there exists an affine mapping A so that $\widetilde{Q} = Q \circ A$.*

Phase 2. The second phase assumes that all encodings of the encoded round function are affine mappings (as the other parts have been retrieved in Phase 1). Phase 2 transforms the affine encoded round function into a Substitution-Affine Transformation (SAT) cipher (see [75, Def. 4]) round function, defined as a two-layered n-bit bijective mapping $\epsilon^{(r)} \circ Q^{(r)}$ where $\epsilon^{(r)}$ is a bijective affine mapping on $\mathbb{F}_2^n$ and $Q^{(r)}$ comprises s key-dependent m-bit bijective non-linear lookup tables $Q_i^{(r)}$ $(1 \leq i \leq s)$ in parallel. Both layers are separately accessible to the attacker.

Phase 3. The third phase extracts the n-bit round key $k^{(r)}$. By comparing the affine encoded round function with the SAT cipher round function constructed during Phase 2, the only non-affine components are given by the s key-independent m-bit S-boxes $S_i^{(r)}$ (included in the public cipher's specification) in the former and by the s key-dependent m-bit S-boxes $Q_i^{(r)}$ in the latter. This brings us to the fact that there exists an affine equivalence between $Q_i^{(r)}$ and $S_i^{(r)}$ for $1 \leq i \leq s$, i.e., $Q_i^{(r)} = B_i^{(r)} \circ S_i^{(r)} \circ A_i^{(r)}$, where $(A_i^{(r)}, B_i^{(r)})$ is a pair of bijective affine mappings on $\mathbb{F}_2^m$. Furthermore, $A_i^{(r)}$ is dependent on $k_i^{(r)}$, i.e., the i-th m-bit word of the round key $k^{(r)}$. Finally, the algorithm given by [75, Fig. 1] enables the attacker to extract the n-bit round key $k^{(r)}$: the presented algorithm is based on the affine equivalence algorithm (AE) for S-boxes proposed by Biryukov et al. in [14], and on the Linear Equivalence algorithm for matrices (LEPM – [75, Def. 6]).

Work factor of Michiels et al.'s attack. Although the work factor of Phase 1 was originally estimated at $2 \cdot s \cdot 2^{3m}$ by Michiels et al. [75], the improvement by Tolhuizen [101] (discussed in Chapter 4) reduces this cost to only $2 \cdot s \cdot (3+4m) \cdot 2^m$. However, as the work factor of Phase 3 depends on the specification of the chosen SLT cipher (i.e., the S-boxes and diffusion operators), Michiels et al. made no statements with regard to the overall work factor.

Generalizing Even Further

Note that Phase 1 of both the BGE attack as well as Michiels et al.'s attack (i.e., the removal of the non-affine component of the white-box encodings) can be further generalized.

Definition 26 (Double Surjective (DS) function)**.** *Let us define the function $f : U \times V \to \mathbb{F}_2^m : (x_1, x_2) \mapsto y = f(x_1, x_2)$. Then f is called a Double Surjective (DS) function if the following two mappings are surjections on $\mathbb{F}_2^m$ for any constant $(c_1, c_2) \in U \times V$:*

$$\begin{aligned} f^{c_1} &: V \to \mathbb{F}_2^m : x_2 \mapsto y = f(c_1, x_2) \ , \\ f^{c_2} &: U \to \mathbb{F}_2^m : x_1 \mapsto y = f(x_1, c_2) \ . \end{aligned}$$

Theorem 2. *If, for a given white-box implementation, the attacker has access to a fixed encoded DS function f' defined as $f' = b \circ f \circ (a_1^{-1}, a_2^{-1})$ where f is a DS function given by Def. 26, where a_1, a_2 are secret fixed bijective white-box encodings (random permutations) applied to the inputs x_1, x_2, respectively, and where b is a secret fixed bijective white-box encoding (random permutation) applied to the output y, then he is able to remove the non-affine component of the output encoding b.*

Although the above theorem has been applied only at the level of encoded round functions in both the BGE attack [13] as well as Michiels et al.'s attack [75], it can be applied to any lookup table or composition of lookup tables implementing a fixed encoded DS function in the white-box implementation. This has been illustrated by Wyseur [103, Sect. 3.6.2] by applying the above theorem to the encoded nibble XOR tables. Such tables, referred to as the Type IV tables (Fig. 3.3e), occur frequently in Chow et al.'s white-box AES implementation [23].

3.6 Conclusion and Outline of Part II

This chapter elaborated in detail on the design and analysis of practical lookup-table-based white-box AES implementations as specified by Chow et al. in [23].

With respect to the analysis part, this chapter introduced two related definitions of a total break of white-box implementations: secret key recovery (KR - Def. 19) and white-box key recovery (WBKR - Def. 20). These definitions are used throughout this thesis. The practical BGE attack [13] showed the KR-insecurity of Chow et al.'s white-box AES implementation by successfully extracting the embedded secret AES key with a work factor of 2^{30}. Further, generalizations of the BGE attack have been discussed that can be applied to white-box implementations (obtained by the generic white-box techniques of Chow et al.) of key-instantiated block ciphers if some specific (but mild) conditions are met on both the specification of the cipher as well as its white-box implementation. These negative results triggered research on designing new white-box AES implementations offering resistance against all known white-box attacks (i.e., the BGE attack and Michiels et al.'s attack). This led to the following three proposals of new white-box AES implementations.

1. **Bringer et al.'s perturbated white-box AES implementation [20].**
   In 2006, Bringer, Chabanne and Dottax [20] presented a novel white-box technique based on perturbations. The idea behind the novel technique is to hide the algebraic structure of AES by injecting faults in the round function computations that will only be corrected for at the end (i.e., in the final round). Furthermore, the obtained white-box AES implementation is represented by a system of polynomial equations over $\mathbb{F}_{256}$ instead of a network of encoded lookup tables.

2. **The Xiao-Lai white-box AES implementation [107].**
   In 2009, Xiao and Lai [107] proposed a new lookup-table-based white-box AES implementation by applying the generic white-box techniques of Chow et al. in a very specific way: (i) all secret white-box encodings are solely $\mathbb{F}_2$-linear and (ii) the secret white-box encodings operate on at least two bytes simultaneously.

3. **Karroumi's dual white-box AES implementation [53].**
   In 2010, Karroumi [53] proposed a new lookup-table-based white-box AES implementation based on the same idea of Bringer et al., i.e., to hide the algebraic structure of AES. Karroumi's approach in doing so is first to generate a dual AES cipher from a key-instantiated AES cipher and second to apply the white-box techniques as presented by Chow et al. in [23] to the dual AES cipher.

Although all new proposals above were claimed to be resistant against the BGE attack, and hence were claimed to be white-box secure, the new research contributions presented in the following chapters show otherwise. In particular, Chapters 4 and 5 show the WBKR-insecurity of Karroumi's white-box AES implementation and the Xiao-Lai white-box AES implementation, respectively, and Chapter 6 shows the KR-insecurity (WBKR-insecurity is not applicable due to the lack of external encodings) of Bringer et al.'s perturbated white-box AES implementation. Additionally, Chapter 4 presents several improvements to the BGE attack and shows that its overall work factor can be reduced to 2^{22} when all improvements are implemented.

Chapter 4

Revisiting the BGE Attack

In 2004, Billet et al. [13] presented a practical attack called the BGE attack (Sect. 3.5.2) on the white-box AES implementation of Chow et al. [23], efficiently extracting its embedded AES key with a work factor of 2^{30}. As a result, Chow et al.'s implementation was proven to be KR-insecure, which is the primary objective of a white-box attacker. In response to this cryptanalytic result, in 2010, Karroumi [53] presented a new white-box AES implementation that is designed to withstand the BGE attack. In order to do so, Karroumi uses the concept of dual ciphers [3, 2, 88, 14] and the white-box techniques of Chow et al. to design his new white-box AES implementation. In [53], Karroumi argues that the additional secrecy introduced by the dual cipher increases the work factor of the BGE attack to 2^{93}.

In this chapter, we present the following two research contributions, which we presented in [36] and in [63] with Lepoint and Rivain as part of a merged paper:

1. Starting from Tolhuizen's improvement [101] of the first phase of the BGE attack, we present several improvements of the other phases. As a consequence, the work factor of the original BGE attack can be reduced to 2^{22} when Tolhuizen's improvement and all our improvements are combined. Additionally, we propose an efficient method to extract the bijective external encodings from Chow et al.'s white-box AES implementation. This method causes the work factor of the improved BGE attack to slightly increase to 2^{23}. This result, which is covered in Sect. 4.1, shows that the white-box AES implementation of Chow et al. is WBKR-insecure. But recall that the original BGE attack already showed the KR-insecurity of Chow et al.'s implementation with a practical work factor.

2. Second, we show that Karroumi's white-box AES implementation [53] belongs to the class of white-box AES implementations specified by Chow et al. in [23]. As a consequence, Karroumi's implementation is vulnerable to the attack (i.e., the BGE attack and our improved version of this attack) it was designed to resist. This result is covered in Sect. 4.2.

Recall from Sect. 3.5.2 that the original BGE attack was described by means of three phases. This chapter builds on that description. Consequently, the notation used throughout this chapter corresponds to the one introduced in Sect. 3.5.2; for instance, an *encoded AES subround* refers to the mapping defined by Def. 23. For details about the BGE attack, refer to [13]. This chapter assumes throughout and without loss of generality that AES-128 is used.

4.1 Improving the BGE Attack

In 2012, Tolhuizen [101] proposed an improvement of the most time-consuming phase of the BGE attack (i.e., Phase 1), reducing the work factor of this phase to approximately 2^{19}. If the improvement of Tolhuizen is implemented, then the work factor of the BGE attack is dominated by the other phases of the BGE attack, and equals 2^{29}. In this section we present several improvements to the other phases of the BGE attack:

1. A method to reduce the work factor of Phase 2 of the BGE attack;
2. An efficient method to retrieve the round key bytes of round $r + 1$ after the round key bytes of round r are extracted;
3. An efficient method to determine the correct order of the round key bytes, given the round key bytes of two consecutive rounds.

As the work factors of Phases 1 and 2 of the BGE attack are reduced by Tolhuizen's improvement and the first improvement above, respectively, it is now important to have an efficient method for Phase 3 of the BGE attack as well, as otherwise the work factor of this phase could dominate the overall work factor. The second and third improvements above comprise such a method for Phase 3. It will be shown that Tolhuizen's improvement of Phase 1 of the BGE attack and the above improvements of the other phases reduce the work factor of the BGE attack to 2^{22}. In addition to the above improvements, we also present the following method that is not included in the original BGE attack:

4. An efficient method to extract the invertible external encodings.

As will be shown, the work factor of this method causes the work factor of the improved BGE attack to slightly increase to 2^{23}.

The improved BGE attack comprises five (instead of three) phases. Each phase is carefully described in the following.

4.1.1 Phases 1 and 2: Retrieve the round key bytes $\bar{k}_i^{(r,j)}$ $(0 \leq i, j \leq 3)$ associated with round r $(2 \leq r \leq 8)$.

The first two phases are the ones of the BGE attack [13] (Sect. 3.5.2) using Tolhuizen's improvement, and retrieve the round key bytes $\bar{k}_i^{(r,j)}$ for $0 \leq i, j \leq 3$ associated with round r for some r with $2 \leq r \leq 8$.

Work factor of Phase 1. Tolhuizen's improvement [101] reduces the work factor of Phase 1 to around $2 \cdot 4 \cdot 4 \cdot (35 \cdot 2^8) < 2^{19}$. The first three factors (i.e., $2 \cdot 4 \cdot 4$) denote the number of encodings involved in Phase 1, i.e., four encodings for each of the four subrounds for each of the two consecutive rounds. The fourth factor (i.e., $35 \cdot 2^8$) denotes the work factor required to retrieve one encoding up to an unknown affine part using Tolhuizen's method.

Work factor of Phase 2. The expected work factor F of the second phase as described in [13] is given by

$$F \approx 2 \cdot 4 \cdot 4 \cdot 2^{15} \cdot 2^8 = 2^{28} \ ,$$

and is measured in the number of evaluations of mappings on $\mathbb{F}_2^8$. The evaluations are required to determine if a mapping on $\mathbb{F}_2^8$ is affine. The mappings f that need to be tested for being affine are listed in [13, Proposition 3]. Each f is associated with a secret encoding $P_i^{(r,j)}$ $(0 \leq i, j \leq 3)$ of a round r. As Phase 2 needs to be applied to two consecutive rounds, this involves a total of $2 \cdot 4 \cdot 4$ mappings (which corresponds to the first three factors in F). The mappings f are permutations on $\mathbb{F}_2^8$ and have the structure

$$f = S^{-1} \circ Q_{(c,d)}^{-1} \circ Q \circ S \circ \oplus_k \circ P \ , \tag{4.1}$$

where S denotes the AES S-box mapping (viewed as a permutation on $\mathbb{F}_2^8$), k denotes a key byte, P and Q denote bijective affine mappings on $\mathbb{F}_2^8$, and $Q_{(c,d)}^{-1}$ denotes a bijective affine mapping on $\mathbb{F}_2^8$ for each pair $(c,d) \in \mathbb{F}_{256}^2$. Furthermore, $Q_{(c,d)}^{-1} = Q^{-1}$ for one specific pair $(c,d) \in \mathbb{F}_{256}^2$. An affine test is performed for each possible pair $(c,d) \in \mathbb{F}_{256}^2$ until the corresponding mapping f is affine. The expected number of pairs for which the test is performed equals approximately 2^{15}, which is the fourth factor in F. The fifth factor in F, i.e., 2^8, is associated with the test used in [13].

Instead of the test used in [13], which requires 2^n evaluations to determine if $f : \mathbb{F}_2^n \to \mathbb{F}_2^n$ is affine, we use the following algorithm (under the assumption that $n \geq 2$) to reduce the expected number of evaluations. If e_i $(1 \leq i \leq n)$ denotes the i-th unit vector in $\mathbb{F}_2^n$, then the algorithm first verifies if the equation

$$f(e_1 \oplus e_2) = f(0) \oplus f(e_1) \oplus f(e_2) \tag{4.2}$$

holds true. If this equation does not hold true, then the algorithm terminates with "f is not affine". Observe that the algorithm requires four evaluations of f in this case. If (4.2) holds true, then the algorithm applies the method used in [13] to determine if f is affine (with the only difference that f is not re-evaluated for the four input values $0, e_1, e_2$ and $e_1 \oplus e_2$). In this case 2^n evaluations of f are required.

To show the correctness of this algorithm (called the *affine test* in the following), it is sufficient to show that an affine mapping always satisfies (4.2). If f is affine, then $f(x) = A(x) \oplus b$ for some $A \in \mathbb{F}_2^{n\times n}$ and some $b \in \mathbb{F}_2^n$. It follows that $f(0) \oplus f(e_1) \oplus f(e_2) = b \oplus A(e_1) \oplus b \oplus A(e_2) \oplus b = A(e_1 \oplus e_2) \oplus b = f(e_1 \oplus e_2)$.

Lemma 1. *If f is a random permutation on $\mathbb{F}_2^n$ and if $E(n)$ denotes the expected number of evaluations of f required by the algorithm described above, then $E(n) < 5$.*

Proof. Let $p(n)$ denote the probability that (4.2) holds true for a random permutation. To determine $p(n)$, note that $f(0), f(e_1), f(e_2)$ and $f(e_1 \oplus e_2)$ are four distinct elements of $\mathbb{F}_2^n$ if f is a permutation. From this it follows that $f(0) \oplus f(e_1) \oplus f(e_2)$ and $f(e_1 \oplus e_2)$ are both elements of $\mathbb{F}_2^n \setminus \{f(0), f(e_1), f(e_2)\}$. Further, as f is a random permutation, $f(e_1 \oplus e_2)$ is a random element of this set. Hence, $p(n) = 1/(2^n - 3)$ and $E(n) = 4(1-p) + 2^n p = 4 + (2^n - 4)/(2^n - 3) < 5$. □

Under the assumption that f in (4.1) behaves as a random permutation on $\mathbb{F}_2^8$ for every incorrect guess for (c, d), the expected work factor of the affine test is reduced from 2^8 to approximately five evaluations if f is not affine and the work factor is 2^8 if f is affine. This implies that the fifth factor in F is reduced to approximately five. That is, the expected work factor of Phase 2 of the BGE attack is now approximately $2 \cdot 4 \cdot 4 \cdot 2^{15} \cdot 5 \approx 2^{22}$.

4.1.2 Phase 3: Retrieve the round key bytes $\bar{k}_i^{(r+1,j)}$ $(0 \leq i, j \leq 3)$ associated with round $r+1$.

As mentioned in the description of the third phase of the original BGE attack in Sect. 3.5.2, Billet et al. [13] obtain the round key bytes of round $r+1$ by applying Phases 1 and 2 to round $r+1$ as well. Here, we present a more efficient method based on the affine test described above. The method comprises the following three steps for each encoded AES subround $\texttt{AES}_{enc}^{(r+1,j)}$ $(0 \leq j \leq 3)$ associated with round $r+1$ to retrieve the round key bytes $\bar{k}_i^{(r+1,j)}$ $(0 \leq i, j \leq 3)$:

<u>*Step 3.1*</u> applies Phase 1 (using Tolhuizen's improvement [101]) to round $r+1$ in order to retrieve the encodings $Q_i^{(r+1,j)}$ $(0 \leq i \leq 3)$ up to an affine part.

<u>*Step 3.2*</u> first removes the non-affine part of the output encodings as recovered in Step 3.1 from the encoded AES subround. Next, Step 3.2 removes the input encodings $P_i^{(r+1,j)}$ $(0 \leq i \leq 3)$ from the encoded AES subround (observe that the inverses of these input encodings were obtained in Phases 1 and 2). The resulting mapping $f^{(r+1,j)} : (\mathbb{F}_2^8)^4 \rightarrow (\mathbb{F}_2^8)^4$ is given by

$$f^{(r+1,j)} = \big(\widehat{Q}_0^{(r+1,j)}, \widehat{Q}_1^{(r+1,j)}, \widehat{Q}_2^{(r+1,j)}, \widehat{Q}_3^{(r+1,j)}\big) \circ \overline{\texttt{AES}}^{(r+1,j)} \quad ,$$

where $\widehat{Q}_i^{(r+1,j)}$ $(0 \leq i \leq 3)$ are affine output encodings.

<u>*Step 3.3*</u> retrieves the round key bytes $\bar{k}_i^{(r+1,j)}$ $(0 \leq i \leq 3)$. To find a key byte, say $\bar{k}_0^{(r+1,j)}$, fix the other three input bytes to $f^{(r+1,j)}$ (e.g., to zero), search over all possible 2^8 values of the key byte k and verify whether

$$g_k(x) = f^{(r+1,j)}\big(k \oplus S^{-1}(x), 0, 0, 0\big)$$

is affine using the test described above. If $g_k(x)$ is affine, then $\bar{k}_0^{(r+1,j)} = k$. Repeat this for $\bar{k}_i^{(r+1,j)}$ $(i = 1, 2, 3)$. This step is illustrated in Fig. 4.1.

The correctness of Step 3.3 uses the fact that the mapping $S\big(c \oplus S^{-1}(x)\big)$ is non-affine for all non-zero values of c. This has already been proven in [13, proof of Proposition 3].

Work factor of Phase 3. The work factor of Step 3.3 equals $4 \cdot 4 \cdot 2^7 \cdot 5 \approx 2^{13}$, where $4 \cdot 4$ denotes the number of round key bytes, 2^7 denotes the expected number of key values for which the affine test is performed and 5 denotes the expected number of evaluations of the affine test if g_k is not affine. The work factor of Step 3.1 is $4 \cdot 4 \cdot (35 \cdot 2^8) < 2^{18}$, where the first two factors denote the number of output encodings involved in Step 3.1. As a result, the work factor of Phase 3 is dominated by Step 3.1 and is less than 2^{18}.

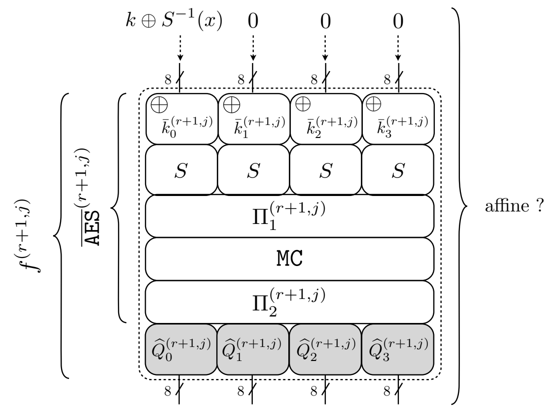


Figure 4.1: Phase 3 of the improved BGE attack: retrieve round key byte $\bar{k}_0^{(r+1,j)}$.

4.1.3 Phase 4: Determine the correct order of the round key bytes and extract the secret AES key.

After Phases 1-3, the values of the round key bytes of two consecutive rounds r and $r+1$ are known. However, for each round, the order of the round key bytes of each subround and the order of the four subrounds are still unknown. Notice that there are still $(4!)^5 \approx 2^{23}$ possibilities for the round key if only the bytes of that round key are considered. In [13], Billet et al. indicated how the correct order can be determined given the "shuffled" round key bytes of rounds r and $r+1$. However, [13] does not contain an explicit description of such a method. As the work factor of the first three phases equals 2^{22}, it is desirable to have a method to determine the correct order of the round key bytes with a work factor that is less than 2^{22}. Below we present such a method, comprising the following three steps:

Step 4.1 retrieves $\mathtt{MC}^{(r,j)}$ associated with each encoded AES subround $\mathtt{AES}_{\mathrm{enc}}^{(r,j)}$ $(0 \le j \le 3)$ of round r. Recall that the encodings $P_i^{(r,j)}$ and $Q_i^{(r,j)}$ $(0 \le i, j \le 3)$ were obtained in Phases 1 and 2. Together with the knowledge of the round key bytes $\bar{k}_i^{(r,j)}$ $(0 \le i, j \le 3)$, compute

$$\mathtt{MC}^{(r,j)} = \left(Q_0^{(r,j)}, Q_1^{(r,j)}, Q_2^{(r,j)}, Q_3^{(r,j)}\right)^{-1} \circ \mathtt{AES}_{\mathrm{enc}}^{(r,j)} \circ \left(P_0^{(r,j)}, P_1^{(r,j)}, P_2^{(r,j)}, P_3^{(r,j)}\right)^{-1} \circ \oplus_{[\bar{k}_i^{(r,j)}]_{0\le i\le 3}} \circ (S,S,S,S)^{-1} \ ,$$

for $j = 0, 1, 2, 3$.

Step 4.2 : for each $\texttt{MC}^{(r,j)}$ $(0 \leq j \leq 3)$, compute permutations $\Pi_1, \Pi_2 : (\mathbb{F}_2^8)^4 \to (\mathbb{F}_2^8)^4$ such that

$$\texttt{MC}^{(r,j)} = \Pi_2 \circ \texttt{MC} \circ \Pi_1 \ . \tag{4.3}$$

Let $(\Pi^{(1)}, \Pi^{(2)})$ denote the pairs of permutations for which $\texttt{MC}$ remains invariant, i.e., $\texttt{MC} = \Pi^{(2)} \circ \texttt{MC} \circ \Pi^{(1)}$. It is easily verified that there are exactly four such pairs. The four permutations $\Pi^{(1)}$ are the four different circular shifts on the indices of a 4-byte vector, and $\Pi^{(2)} = (\Pi^{(1)})^{-1}$ for each of these pairs. This implies that there are also exactly four different pairs of permutations satisfying (4.3), given by

$$\left(\Pi^{(1)} \circ \Pi_1 \ , \ \Pi_2 \circ \Pi^{(2)}\right) \ . \tag{4.4}$$

As a consequence, finding one pair of permutation matrices satisfying (4.3) suffices to find the remaining three as well. Notice that exactly one of these four pairs of permutations equals the pair $(\Pi_1^{(r,j)}, \Pi_2^{(r,j)})$ of the encoded subround (see also Def. 23); in other words, one of these pairs is the correct pair.

After this, the order of the round key bytes associated with each subround is known up to an uncertainty of four possibilities (circular shifts). Observe that the order of the four subrounds is still unknown.

Step 4.3 determines the correct order of the round key bytes. For each of the possible orderings of the four AES subrounds of round r and the round key bytes within these subrounds (as determined in Step 4.2), obtain a candidate for the $(r+1)^{th}$ round key using the following two methods: (i) the AES key scheduling algorithm and (ii) the data-flow of the white-box AES implementation between the encoded subrounds of rounds r and $r+1$. Notice that once an order of the round key bytes of round r is selected, the order of the round key bytes of round $r+1$ can be determined using the corresponding pair of permutations (4.4) of each of the subrounds of round r and the data-flow of the white-box implementation. With overwhelming probability, only one ordering of round key bytes of round r results in the same $(r+1)^{th}$ round key; this ordering corresponds to the correct round key of round r. Finally, use the property of the AES key scheduling algorithm that the AES key can be computed if one of the round keys is known.

Work factor of Phase 4. A naive approach yields an expected work factor of $(4!)^2 \approx 2^9$ for Step 4.2 by searching over all possible pairs of permutations. Step 4.2 reduces the number of possible orderings of the round key bytes from 2^{23} to $4^4 \cdot 4! < 2^{13}$ (where the first and second factor denote the possible orderings of round key bytes within each subround and of the four subrounds, respectively), which equals the work factor of Step 4.3. As a result, the overall work factor of Phase 4 is dominated by the work factor of Step 4.3 and hence is less than 2^{13}.

4.1.4 Phase 5: Extract the external encodings.

After Phases 1-4, the embedded secret AES key is successfully extracted from the white-box AES implementation of Chow et al., showing its KR-insecurity. Below we present an efficient method to retrieve the external encodings as well by extracting their corresponding bijective mappings on $\mathbb{F}_2^{128}$. As a consequence, Chow et al.'s white-box AES implementation becomes WBKR-insecure. This result complements the original BGE attack since [13] does not provide such a method. The presented method has a work factor of less than 2^{22} which is desirable since the work factor of the first four phases equals 2^{22}.

Defining the External Encodings

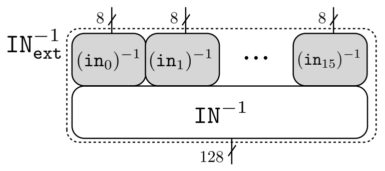
First, the composition of the external encodings should be well-defined. From the compositions of the Type Ia (Fig. 3.3a) and Type Ib (Fig. 3.3b) tables of Chow et al.'s white-box AES implementation follows that

1. the bijective mapping $\mathtt{IN}_{\mathtt{ext}}^{-1} : \mathbb{F}_2^{128} \to \mathbb{F}_2^{128}$, called the *external input encoding*, is composed of the concatenation of the 16 bijective non-linear mappings on $\mathbb{F}_2^{8}$ at the input of the 16 $\mathcal{L}\text{-}\mathtt{Ia}_i^{(1,j)}$ $(0 \leq i, j \leq 3)$ tables followed by the 128-bit linear mixing bijection $\mathtt{IN}^{-1}$;

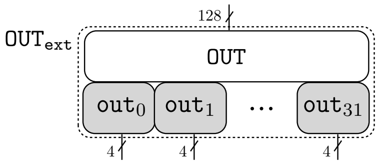
2. the bijective mapping $\mathtt{OUT}_{\mathtt{ext}} : \mathbb{F}_2^{128} \to \mathbb{F}_2^{128}$, called the *external output encoding*, is composed of the 128-bit mixing bijection $\mathtt{OUT}$ followed by the concatenation of the 32 bijective non-linear mappings on $\mathbb{F}_2^{4}$ at the output of the 32 encoded nibble XOR tables at the tail of the XOR lookup-table network; recall that this XOR table network implements the 15 128-bit XOR operations needed due to the matrix partitioning of $\mathtt{OUT}$.

The composition of the external encodings is depicted in Fig. 4.2, where $(\mathtt{in}_i)^{-1}$ $(0 \leq i \leq 15)$ denote the 16 bijective non-linear mappings on $\mathbb{F}_2^{8}$ of $\mathtt{IN}_{\mathtt{ext}}^{-1}$ and $\mathtt{out}_i$ $(0 \leq i \leq 31)$ denote the 32 bijective non-linear mappings on $\mathbb{F}_2^{4}$ of $\mathtt{OUT}_{\mathtt{ext}}$.

Note that in Chapter 3, the external encodings were often referred to as comprising solely the 128-bit linear mixing bijections $\mathtt{IN}^{-1}$ and $\mathtt{OUT}$. However, observe that the white-box implementation of these mixing bijections resulted in the additional applications of the non-linear white-box encodings.



(a) External input encoding $\texttt{IN}^{-1}_{\texttt{ext}}$



(b) External output encoding $\texttt{OUT}_{\texttt{ext}}$

Figure 4.2: The composition of the external input and output encodings of the white-box AES implementation of Chow et al.

Retrieving the External Encodings

After Phases 1-4, the following two observations can be made:

1. After Phase 2, the input encodings $P_i^{(r,j)}$ $(0 \le i, j \le 3)$ of the encoded round r with $2 \le r \le 8$ of the white-box AES implementation are fully retrieved. Further, a round key byte of round r is associated to each $P_i^{(r,j)}$; this one-to-one relation is known to the attacker through the white-box implementation. Since the correct order of the round key bytes of round r is determined in Phase 4, the attacker has access to the unencoded 16-byte AES state at the input of round r, denoted by $\text{STATE}^{(r)}$.

2. After Phase 4, the secret AES key k is retrieved. This enables the attacker to construct a standard AES encryption or decryption algorithm instantiated with k (denoted by $\texttt{AES}_k$ or $\left(\texttt{AES}_k\right)^{-1}$, respectively) mapping unencoded plaintexts P to unencoded ciphertexts C or vice versa.

In the following, let the mapping $\texttt{AES}_k : \mathbb{F}_{256}^{16} \to \mathbb{F}_{256}^{16}$ be defined as $\texttt{AES}_k = \texttt{AES}_k^2 \circ \texttt{AES}_k^1$, where $\texttt{AES}_k^1$ denotes the first $r-1$ rounds of $\texttt{AES}_k$ mapping an unencoded plaintext P to $\text{STATE}^{(r)}$, and where $\texttt{AES}_k^2$ denotes the last $10-(r-1)$ rounds of $\texttt{AES}_k$ mapping $\text{STATE}^{(r)}$ to the unencoded ciphertext C. Consequently, the mapping $\left(\texttt{AES}_k\right)^{-1} : \mathbb{F}_{256}^{16} \to \mathbb{F}_{256}^{16}$ is defined as $\left(\texttt{AES}_k\right)^{-1} = \left(\texttt{AES}_k^1\right)^{-1} \circ \left(\texttt{AES}_k^2\right)^{-1}$.

Now, given the above two observations, the attacker has access to the bijective mappings $\left(\texttt{AES}_k^1\right)^{-1} : \mathbb{F}_{256}^{16} \to \mathbb{F}_{256}^{16}$ and $\left(\texttt{AES}_k^2\right)^{-1} : \mathbb{F}_{256}^{16} \to \mathbb{F}_{256}^{16}$ as defined above, but also to the following two bijective mappings, each indicated in Fig. 4.3:

$$\begin{array}{lclcll}
\texttt{WBAES}_k^1 & : & \mathbb{F}_2^{128} \to \mathbb{F}_{256}^{16} & : & \texttt{WBAES}_k^1 = \texttt{AES}_k^1 \circ \texttt{IN}_{\texttt{ext}}^{-1} & , \\
\texttt{WBAES}_k^2 & : & \mathbb{F}_{256}^{16} \to \mathbb{F}_2^{128} & : & \texttt{WBAES}_k^2 = \texttt{OUT}_{\texttt{ext}} \circ \texttt{AES}_k^2 & .
\end{array}$$

Next, by composing the above mappings in the following specific ways (Fig. 4.3), the attacker has access to the external encodings

$$\mathtt{IN}_{\mathtt{ext}}^{-1} = \left(\mathtt{AES}_k^1\right)^{-1} \circ \mathtt{WBAES}_k^1 \quad \text{and} \quad \mathtt{OUT}_{\mathtt{ext}} = \mathtt{WBAES}_k^2 \circ \left(\mathtt{AES}_k^2\right)^{-1} \; . \qquad (4.5)$$

However, observe from (4.5) that the attacker has *no* access to the inverse mappings of the external encodings since this requires the white-box AES implementation of Chow et al. to be invertible (which is not the case because of the specific composition of the network of encoded lookup tables).

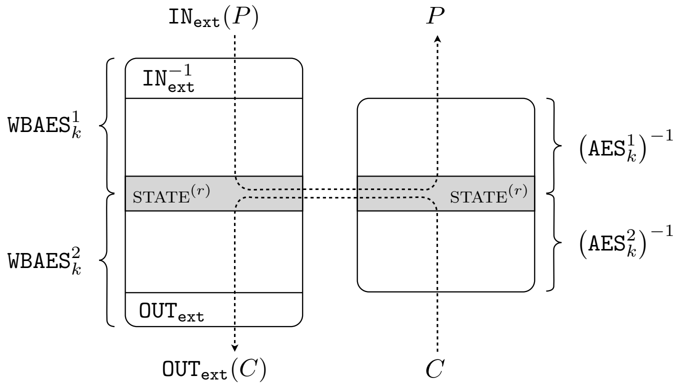

$\mathtt{IN}_{\mathtt{ext}}(P)$
P
$\mathtt{IN}_{\mathtt{ext}}^{-1}$
$\mathtt{WBAES}_k^1$
$\left(\mathtt{AES}_k^1\right)^{-1}$
STATE$^{(r)}$
STATE$^{(r)}$
$\mathtt{WBAES}_k^2$
$\left(\mathtt{AES}_k^2\right)^{-1}$
$\mathtt{OUT}_{\mathtt{ext}}$
$\mathtt{OUT}_{\mathtt{ext}}(C)$
C

Figure 4.3: The way how the attacker gains access to the external encodings in the white-box AES implementation of Chow et al.

So after Phases 1-4, it is clear that the attacker has only partially retrieved the external encodings (i.e., only in one way). In order to fully retrieve the external encodings, he needs to find their bijective components separately (Fig. 4.2). Due to the specific composition of the external encodings, *equivalent* components will be retrieved; *equivalent* in the sense that their composition nevertheless yields the actual external input and output encodings. Below, we present a method how to extract these *equivalent* components. The method exploits the ability to have access to the mappings defined by (4.5).

External input encoding. Recall from Fig. 4.2a that the composition of the external input encoding is given by a two-layered structure $\mathtt{IN}_{\mathtt{ext}}^{-1} = L \circ S$: (i) the S layer consists of 16 lookup tables S_i in parallel representing the 16 8-bit bijective non-linear mappings $(\mathtt{in}_i)^{-1}$ for $0 \leq i \leq 15$, and (ii) the L layer consists of the 128-bit mixing bijection $\mathtt{IN}^{-1}$. The method to retrieve equivalent components originates from the SASAS structural cryptanalysis (cf. Biryukov and Shamir [15]) and is explained below.

First, an equivalent representation of the L layer is obtained by repeating the following for each S-box S_i $(0 \leq i \leq 15)$: (i) vary the input to S_i over all 2^8 possible values while fixing the input to the other S-boxes S_j $(j \neq i)$ to some constant, (ii) store the corresponding 2^8 128-bit output values of $\mathtt{IN}_{\mathtt{ext}}^{-1}$, select one of these values and XOR it to all 2^8 values in order to obtain the 8-dimensional linear subspace generated by all possible outputs from S_i, and (iii) use Gaussian elimination to obtain a basis for this 8-dimensional linear subspace. In total, 16 8-dimensional linear subspaces are obtained that together form the equivalent representation of L, denoted by L'. The 128-bit bijective linear mapping L' can be easily inverted.

Second, an equivalent representation of the remaining S layer is obtained. Each equivalent lookup table S'_i $(0 \leq i \leq 15)$ can be constructed by calculating

$$(y_0, y_1, \ldots, y_{15}) = (L')^{-1}\big(\mathtt{IN}_{\mathtt{ext}}^{-1}(\underbrace{x, x, \ldots, x}_{16 \text{ times}})\big)$$

for all 2^8 possible values of $x \in \mathbb{F}_2^8$, and storing $y_i = S'_i(x)$ for $0 \leq i \leq 15$. Each table S'_i $(0 \leq i \leq 15)$ mapping 8 bits to 8 bits can be easily inverted.

This implies that the 128-bit bijective mapping $\mathtt{IN}_{\mathtt{ext}}^{-1}$ can be fully recovered by means of its invertible equivalent components S'_i $(0 \leq i \leq 15)$ and L' such that

$$\begin{aligned} \mathtt{IN}_{\mathtt{ext}}^{-1} &= L' \circ \big(S'_0, S'_1, \ldots, S'_{15}\big) \quad , \\ \mathtt{IN}_{\mathtt{ext}} &= \big((S'_0)^{-1}, (S'_1)^{-1}, \ldots, (S'_{15})^{-1}\big) \circ (L')^{-1} \quad . \end{aligned}$$

External output encoding. Recall that the 32 4-bit bijective non-linear mappings $\mathtt{out}_i$ $(0 \leq i \leq 31)$ of $\mathtt{OUT}_{\mathtt{ext}}$ (Fig. 4.2b) correspond with the non-linear output encodings of 32 encoded nibble XOR tables. Further, recall from Sect. 3.5.4 (p. 102) that the non-affine component of the output encoding of each encoded nibble XOR table can be retrieved (see also Wyseur [103, Sect. 3.6.2]).

As a result, the attacker is able to compute the non-affine component of each $\mathtt{out}_i$ $(0 \leq i \leq 31)$, denoted by $\widetilde{\mathtt{out}_i}$, such that $\widetilde{\mathtt{out}_i} = \mathtt{out}_i \circ \widehat{\mathtt{out}_i}$ for some unknown bijective affine mapping $\widehat{\mathtt{out}_i}$ (see Theorem 1, p. 101). Each mapping $\widetilde{\mathtt{out}_i}$ $(0 \leq i \leq 31)$ is determined by means of a lookup table mapping 4 bits to 4 bits, and hence can easily be inverted. The remaining part of $\mathtt{OUT}_{\mathtt{ext}}$, given by

$$\begin{aligned} \widehat{\mathtt{OUT}}_{\mathtt{ext}} &= \big((\widetilde{\mathtt{out}}_0)^{-1}, (\widetilde{\mathtt{out}}_1)^{-1}, \ldots, (\widetilde{\mathtt{out}}_{31})^{-1}\big) \circ \mathtt{OUT}_{\mathtt{ext}} \\ &= \big((\widehat{\mathtt{out}}_0)^{-1}, (\widehat{\mathtt{out}}_1)^{-1}, \ldots, (\widehat{\mathtt{out}}_{31})^{-1}\big) \circ \mathtt{OUT} \ , \end{aligned}$$

is a bijective affine mapping on $\mathbb{F}_2^{128}$ that can be easily retrieved and inverted.

This implies that the 128-bit bijective mapping $\mathtt{OUT}_{\mathtt{ext}}$ can be fully recovered by means of its invertible equivalent components $\widetilde{\mathtt{out}}_i$ $(0 \leq i \leq 31)$ and $\widehat{\mathtt{OUT}}_{\mathtt{ext}}$ such that

$$\begin{aligned} \mathtt{OUT}_{\mathtt{ext}} &= \left(\widetilde{\mathtt{out}}_0, \widetilde{\mathtt{out}}_1, \ldots, \widetilde{\mathtt{out}}_{31}\right) \circ \widehat{\mathtt{OUT}}_{\mathtt{ext}} & , \\ \mathtt{OUT}_{\mathtt{ext}}^{-1} &= \widehat{\mathtt{OUT}}_{\mathtt{ext}}^{-1} \circ \left((\widetilde{\mathtt{out}}_0)^{-1}, (\widetilde{\mathtt{out}}_1)^{-1}, \ldots, (\widetilde{\mathtt{out}}_{31})^{-1}\right) & . \end{aligned}$$

Work factor of Phase 5. The work factor required to extract the external input encoding is given by

$$\underbrace{16 \cdot (2^8 + 8^3)}_{\text{retrieving } L'} + \underbrace{2^8 + 128^3}_{\text{retrieving } S_i'} \approx 2^{21} \ ,$$

where the dominant work factor 128^3 refers to the expected work factor required to invert L' using Guass-Jordan elimination. The work factor required to extract the external output encoding is given by

$$\underbrace{32 \cdot 19 \cdot 2^4}_{\text{retrieving } \widetilde{\mathtt{out}}_i} + \underbrace{129}_{\text{retrieving } \widehat{\mathtt{OUT}}_{\mathtt{ext}}} \approx 2^{13} \ ,$$

using Tolhuizen's improvement [101]. As a result, the overall work factor of Phase 5 is dominated by the work factor required to extract the external input encoding and hence is approximately 2^{21}.

4.1.5 Work Factor and Conclusion

The work factor of the improved BGE attack (Phases 1-4) is dominated by the work factor of the second phase and equals 2^{22}.

Phase 1	$2 \cdot 4 \cdot 4 \cdot (35 \cdot 2^8) < 2^{19}$
Phase 2	$2 \cdot 4 \cdot 4 \cdot 2^{15} \cdot 5 \approx 2^{22}$
Phase 3	$4 \cdot 4 \cdot (35 \cdot 2^8) < 2^{18}$
Phase 4	$4^4 \cdot 4! < 2^{13}$
Total work factor	2^{22}

Now, by including the method to extract the bijective mappings defining the external encodings (i.e., Phase 5), the work factor of the improved BGE attack is slightly increased to 2^{23}. As a work factor of 2^{23} clearly indicates the practicality of the attack, the white-box implementation of Chow et al. is WBKR-insecure. Observe that the original BGE attack [13] already indicated the KR-insecurity of Chow et al.'s white-box AES implementation.

Note that the uncertainty in the order of the round key bytes results in the need to retrieve key bytes of two consecutive rounds. This affects the work factor of the original BGE attack. In the improved BGE attack this is no longer the case, as the work factors of the phases that determine the correct order (i.e. Phases 3 and 4) are negligible compared to the work factor of Phase 2. A consequence of Tolhuizen's improvement is that the use of non-affine white-box encodings has a negligible impact on the overall work factor of the improved BGE attack.

4.2 Cryptanalysis of Karroumi's White-Box AES Implementation

This section shows that Karroumi's [53] and Chow et al.'s [23] white-box AES implementations are the same so that Karroumi's white-box AES implementation remains vulnerable to the attack it was designed to resist, i.e., the BGE attack [13] of which an improved version is presented in Sect. 4.1. Note that this observation was independently made by us [36], by Lepoint and Rivain [62] and by Klinec [56].

4.2.1 Karroumi's White-Box AES Implementation

Karroumi's method to generate a white-box AES implementation [53] can be divided into the following two phases:

Phase 1 generates a dual AES cipher from a key-instantiated AES cipher.

Phase 2 applies the white-box techniques presented by Chow et al. in [23] (Sect. 3.3) to the dual AES cipher.

Below, aspects of these phases that are relevant to the cryptanalysis presented in Sect. 4.2.2 are described. In the following, the alternative description of AES-128 (Fig. 2.1b) is assumed.

Phase 1: Dual AES cipher

In this section we give a description of the set of dual AES ciphers used by Karroumi in [53]. First, we define a *dual AES subround*. The following notation is used: $m_\alpha : \mathbb{F}_{256} \to \mathbb{F}_{256}$ with $\alpha \in \mathbb{F}_{256}^*$ is defined by $m_\alpha(x) = \alpha \otimes x$, and $f_t : \mathbb{F}_{256} \to \mathbb{F}_{256}$ defined by $f_t(x) = x^{2^t}$ for $0 \leq t \leq 7$ are the automorphisms of $\mathbb{F}_{256}$ over $\mathbb{F}_2$. Further, $R_l : \mathbb{F}_{256} \to \mathbb{F}_{256}$ are the isomorphisms mapping elements in the AES polynomial representation (as specified in FIPS 197 [69]) to field elements in one of the polynomial representations of $\mathbb{F}_{256}$. There are 30 irreducible polynomials of degree 8 over $\mathbb{F}_2$, each one resulting in a unique polynomial representation of $\mathbb{F}_{256}$ (one of these representations being the AES polynomial representation), hence in total there are 30 distinct isomorphisms R_l $(1 \leq l \leq 30)$. The addition and multiplication operations in the polynomial representation associated with R_l are denoted by $\oplus_l$ and $\otimes_l$, respectively ($\oplus_l$ and $\otimes_l$ being equal to $\oplus$ and $\otimes$ for exactly one value of l with $1 \leq l \leq 30$). Finally, the definition of a dual AES subround uses a set of mappings, denoted by $\mathcal{T}$, and defined by

$$\mathcal{T} = \{R_l \circ m_\alpha \circ f_t \mid 1 \leq l \leq 30, \alpha \in \mathbb{F}_{256}^* \text{ and } 0 \leq t \leq 7\} \ .$$

Observe that an element of $\mathcal{T}$ maps elements in the AES polynomial representation to elements in one of the 30 polynomial representations of $\mathbb{F}_{256}$. Note that the set $\mathcal{T}$ with $|\mathcal{T}| = 61\,200$ was identified by Biryukov, De Cannière, Braeken and Preneel in [14].

Definition 27 (Dual AES subround). *Let $\Delta_{r,j} \in \mathcal{T}$ with $\Delta_{r,j} = R_l \circ m_\alpha \circ f_t$ for some triplet (l, α, t) with $1 \leq l \leq 30, \alpha \in \mathbb{F}_{256}^*$ and $0 \leq t \leq 7$, and let $\delta_{r,j} = R_l \circ f_t$. Further, let $v_i, w_i \in \mathbb{F}_{256}$ for $0 \leq i \leq 3$ be represented using the polynomial representation associated with R_l. The mapping $\mathit{AES}^{(r,j,\Delta_{r,j})} : \mathbb{F}_{256}^4 \to \mathbb{F}_{256}^4$ for $1 \leq r \leq 9$ and $0 \leq j \leq 3$, called a dual AES subround, is defined by $(w_0, w_1, w_2, w_3) = \mathit{AES}^{(r,j,\Delta_{r,j})}(v_0, v_1, v_2, v_3)$ with*

$$\begin{aligned} w_i = {} & \delta_{r,j}(mc_{i,0}) \otimes_l \Delta_{r,j} \circ S \circ \Delta_{r,j}^{-1}\big(v_0 \oplus_l \Delta_{r,j}(\hat{k}_0^{(r,j)})\big) \\ & \oplus_l \delta_{r,j}(mc_{i,1}) \otimes_l \Delta_{r,j} \circ S \circ \Delta_{r,j}^{-1}\big(v_1 \oplus_l \Delta_{r,j}(\hat{k}_1^{(r,j)})\big) \\ & \oplus_l \delta_{r,j}(mc_{i,2}) \otimes_l \Delta_{r,j} \circ S \circ \Delta_{r,j}^{-1}\big(v_2 \oplus_l \Delta_{r,j}(\hat{k}_2^{(r,j)})\big) \\ & \oplus_l \delta_{r,j}(mc_{i,3}) \otimes_l \Delta_{r,j} \circ S \circ \Delta_{r,j}^{-1}\big(v_3 \oplus_l \Delta_{r,j}(\hat{k}_3^{(r,j)})\big) \ , \end{aligned}$$

for $0 \leq i \leq 3$.

The following lemma presents a property that is required to show that a dual AES cipher maintains the functionality of AES. As the lemma is also used in the cryptanalysis presented in Sect. 4.2.2, and as a formal proof of this property is omitted in [14] and [53], we include a proof as well.

Lemma 2. *If $\Delta_{r,j} \in \mathcal{T}$, then*

$$\mathit{AES}^{(r,j,\Delta_{r,j})} \circ (\Delta_{r,j}, \Delta_{r,j}, \Delta_{r,j}, \Delta_{r,j}) = (\Delta_{r,j}, \Delta_{r,j}, \Delta_{r,j}, \Delta_{r,j}) \circ \mathit{AES}^{(r,j)} \ ,$$

for $1 \leq r \leq 9$ and $0 \leq j \leq 3$.

Proof. Let x_i for $0 \leq i \leq 3$ be elements of $\mathbb{F}_{256}$ using the AES polynomial representation, let w_i for $0 \leq i \leq 3$ be elements of $\mathbb{F}_{256}$ using the polynomial representation associated with R_l (assuming that $\Delta_{r,j} = R_l \circ m_\alpha \circ f_t$), and let

$$(w_0, w_1, w_2, w_3) = \texttt{AES}^{(r,j,\Delta_{r,j})} \circ (\Delta_{r,j}, \Delta_{r,j}, \Delta_{r,j}, \Delta_{r,j})(x_0, x_1, x_2, x_3) \ .$$

Substituting $v_i = \Delta_{r,j}(x_i)$ for $0 \leq i \leq 3$ in the equation in Def. 27 yields

$$w_i = \bigoplus_{z=0}^{3}{}_l\, \delta_{r,j}(mc_{i,z}) \otimes_l \Delta_{r,j} \circ S \circ \Delta_{r,j}^{-1}\big(\Delta_{r,j}(x_z) \oplus_l \Delta_{r,j}(\hat{k}_z^{(r,j)})\big) \ ,$$

for $0 \leq i \leq 3$. Next, observe that

$$\begin{aligned}
\Delta_{r,j}(a) \oplus_l \Delta_{r,j}(b) &= R_l \circ m_\alpha \circ f_t(a) \oplus_l R_l \circ m_\alpha \circ f_t(b) \\
&= R_l(m_\alpha \circ f_t(a) \oplus m_\alpha \circ f_t(b)) \\
&= R_l(m_\alpha(f_t(a) \oplus f_t(b)) \\
&= R_l(m_\alpha(f_t(a \oplus b))) \qquad\qquad = \Delta_{r,j}(a \oplus b)
\end{aligned}$$

for all $a, b \in \mathbb{F}_{256}$; the second equality holds true since R_l is an isomorphism, the third equality holds true as $\alpha(a \oplus b) = \alpha(a) \oplus \alpha(b)$ for all $a, b \in \mathbb{F}_{256}$ and all $\alpha \in \mathbb{F}_{256}^*$, and the fourth equality holds true since f_t is an automorphism. It follows that

$$w_i = \bigoplus_{z=0}^{3}{}_l\, \delta_{r,j}(mc_{i,z}) \otimes_l \Delta_{r,j} \circ S\big(x_z \oplus \hat{k}_z^{(r,j)}\big) \ ,$$

for $0 \leq i \leq 3$. Next, note that

$$\begin{aligned}
\delta_{r,j}(a) \otimes_l \Delta_{r,j}(b) &= R_l \circ f_t(a) \otimes_l R_l \circ m_\alpha \circ f_t(b) \\
&= R_l(f_t(a) \otimes m_\alpha \circ f_t(b)) \\
&= R_l(m_\alpha(f_t(a \otimes b))) \qquad\qquad = \Delta_{r,j}(a \otimes b)
\end{aligned}$$

for all $a, b \in \mathbb{F}_{256}$; the second equality holds true since R_l is an isomorphism and the third equality uses the fact that $a^{2^t} \otimes \alpha b^{2^t} = \alpha(ab)^{2^t}$ for all $a, b \in \mathbb{F}_{256}$ and all $\alpha \in \mathbb{F}_{256}^*$. It follows that

$$w_i = \bigoplus_{z=0}^{3}{}_l\, \Delta_{r,j}\Big(mc_{i,z} \otimes S\big(x_z \oplus \hat{k}_z^{(r,j)}\big)\Big) \ ,$$

for $0 \leq i \leq 3$. From this, $\Delta_{r,j}(a) \oplus_l \Delta_{r,j}(b) = \Delta_{r,j}(a \oplus b)$ for all $a, b \in \mathbb{F}_{256}$, and the definition of y_i in Def. 18 (AES subround), it follows that $w_i = \Delta_{r,j}(y_i)$ for $0 \leq i \leq 3$. □

The dual AES rounds used by Karroumi [53] are obtained by performing the following two steps for $1 \leq r \leq 9$:

Step 1.1 assigns a randomly chosen $\Delta_{r,j} \in \mathcal{T}$ to each AES subround $\texttt{AES}^{(r,j)}$ $(1 \leq r \leq 9$ and $0 \leq j \leq 3)$. Based on $\Delta_{r,j}$, the corresponding dual AES subround $\texttt{AES}^{(r,j,\Delta_{r,j})}$ is implemented as specified by Def. 27:

- each instance of an AES S-box is replaced by $\Delta_{r,j} \circ S \circ \Delta_{r,j}^{-1}$;
- the round key bytes $\hat{k}_i^{(r,j)}$ and `MixColumns` coefficients $mc_{i,z}$ are replaced by $\Delta_{r,j}\left(\hat{k}_i^{(r,j)}\right)$ and $\delta_{r,j}(mc_{i,z})$ (with $\delta_{r,j}$ defined as in Def. 27), respectively, for $1 \leq r \leq 9$ and $0 \leq i, j, z \leq 3$;
- the operations $\oplus_l$ and $\otimes_l$ depend on the polynomial representation associated with the mapping R_l (assuming that $\Delta_{r,j} = R_l \circ m_\alpha \circ f_t$).

The mappings $\Delta_{r,j}$ and $\delta_{r,j}$ (and as a result the implementation of the dual cipher) are kept secret.

Step 1.2 ensures that the functionality of AES is maintained by including an additional operation (referred to as `ChangeDualState`) between the `ShiftRows` and `AddRoundKey` operations of round r for $1 \leq r \leq 9$. If the inverse `ShiftRows` operation is defined by the mapping $isr(i,j) = (j+i) \bmod 4$ for $0 \leq i,j \leq 3$, then the `ChangeDualState` operation of round r applies the mapping $C_i^{(r,j)} : \mathbb{F}_{256} \to \mathbb{F}_{256}$ to the byte of the state associated with the i-th input byte of $\texttt{AES}^{(r,j,\Delta_{r,j})}$ for $0 \leq i,j \leq 3$, defined by

$$C_i^{(1,j)} = \Delta_{1,j} \quad \text{and} \quad C_i^{(r,j)} = \Delta_{r,j} \circ \Delta_{r-1,isr(i,j)}^{-1} \quad \text{if} \quad 2 \leq r \leq 9 \ .$$

Observe that for $2 \leq r \leq 9$, $C_i^{(r,j)}$ maps elements from $\mathbb{F}_{256}$ using the polynomial representation associated with $\Delta_{r-1,isr(i,j)}$ to elements of $\mathbb{F}_{256}$ using the polynomial representation associated with $\Delta_{r,j}$.

Karroumi presents two different but equivalent methods (from a security point of view) in [53] to perform the `ChangeDualState` operation, and specifies the white-box AES implementation using one of these methods. Here we use the specification as in [53]; the cryptanalysis can easily be adapted if the other method is used.

Phase 2: Apply the white-box techniques of Chow et al.

The following description of Karroumi's white-box AES implementation is equivalent to the description in [53]:

Step 2.1 applies the techniques of Chow et al. to write the dual AES cipher (with a fixed key) obtained in Phase 1 as a series of lookup tables. In particular, the dual AES key addition operations and the dual S-box operations are merged into key-dependent bijective mappings $T_i^{(r,j,\Delta_{r,j})}$ for $0 \leq i, j \leq 3$ and $1 \leq r \leq 9$. These mappings are referred to as *dual T-boxes* and are defined by

$$T_i^{(r,j,\Delta_{r,j})} = \Delta_{r,j} \circ S \circ \Delta_{r,j}^{-1} \circ \oplus_{\Delta_{r,j}(\hat{k}_i^{(r,j)})} \circ C_i^{(r,j)} \ ,$$

where each dual T-box mapping is implemented as a lookup table mapping 8 input bits to 8 output bits. Recall that $C_i^{(r,j)}$ are the mappings defining the `ChangeDualState` operation. Next, write the other part of the dual AES cipher as a series of lookup tables as indicated by Chow et al. in [23] (described in Sect. 3.3.2 – Phase 1). The number and types of tables (including the tables representing the dual T-boxes) and the data-flow between tables are the same as in the lookup table implementation of AES in [23]. The only difference is that the values of the table entries of the dual AES implementation are likely to be different from the values of the corresponding entries in the AES implementation in [23] due to the dual version of the AES operations.

Step 2.2 applies the white-box encoding techniques of Chow et al. in [23] (described in Sect. 3.3.2 – Phase 2) to this lookup table implementation of dual AES. As these white-box encoding techniques do not depend on the values of the table entries, the number and types of white-box tables, and the data-flow of Karroumi's white-box AES implementation are the same as in the white-box AES implementation of Chow et al. in [23].

In [53], Karroumi argues that the secrecy of the mappings $\Delta_{r,j}$ (and $\delta_{r,j}$), randomly selected from the set $\mathcal{T}$ and used to generate the dual cipher, increases the work factor of the BGE attack to 2^{93}.

4.2.2 Cryptanalysis

This section shows that Karroumi's white-box AES implementation [53] is insecure. Recall that Karroumi's white-box AES implementation uses the same number and types of white-box tables, and that the data-flow of the implementation is the same as in Chow et al.'s white-box AES implementation in [23]. As a result, the techniques of Billet et al. can be applied directly to compose lookup tables in Karroumi's implementation to obtain access to the encoded *dual* AES subrounds (instead of the encoded AES subrounds (Def. 23, p. 89) in the case of Chow et al.'s implementation) for rounds $1 \leq r \leq 9$. In the following definition, $A_i^{(r,j)}$ and $B_i^{(r,j)}$ for $0 \leq i \leq 3$ denote bijective mappings (or encodings) on $\mathbb{F}_2^8$, and the permutations $\Pi_1^{(r,j)}, \Pi_2^{(r,j)}$ and $\pi^{(r)}$ are defined as in

Def. 23. Furthermore, $\pi_1^{(r,j)} : \{0,1,2,3\} \to \{0,1,2,3\}$ denotes the permutation on the indices of a 4-byte vector as a result of the application of $\Pi_1^{(r,j)}$. With respect to these permutations, we introduce the following two notations:

$$i' = \left(\pi_1^{(r,j)}\right)^{-1}(i) \quad \text{and} \quad j' = \pi^{(r)}(j) \quad \text{for} \quad 1 \leq r \leq 9 \text{ and } 0 \leq i,j \leq 3 \ .$$

Further, with slight abuse of notation, an output of $A_i^{(r,j)}$ is considered to be an element of $\mathbb{F}_{256}$ using the polynomial representation associated with the mapping R_l as defined by $\Delta_{r-1,isr(i',j')}$, and an output of $\texttt{AES}^{(r,j,\Delta_{r,j})}$ is considered to be an element of $(\mathbb{F}_2^8)^4$.

Definition 28 (Encoded dual AES subround). *The mapping* $\mathit{AES}_{enc}^{(r,j,\Delta_{r,j})} : (\mathbb{F}_2^8)^4 \to (\mathbb{F}_2^8)^4$ *for* $1 \leq r \leq 9$ *and* $0 \leq j \leq 3$*, called an encoded dual AES subround, is defined by*

$$\mathit{AES}_{enc}^{(r,j,\Delta_{r,j})} = (B_0^{(r,j)}, B_1^{(r,j)}, B_2^{(r,j)}, B_3^{(r,j)}) \circ \overline{\mathit{AES}}^{(r,j,\Delta_{r,j})} \circ (A_0^{(r,j)}, A_1^{(r,j)}, A_2^{(r,j)}, A_3^{(r,j)}) \ , \quad (4.6)$$

where the mapping $\overline{\mathit{AES}}^{(r,j,\Delta_{r,j})}$ *is defined by*

$$\Pi_2^{(r,j)} \circ \mathit{AES}^{(r,j',\Delta_{r,j'})} \circ (C_0^{(r,j')}, C_1^{(r,j')}, C_2^{(r,j')}, C_3^{(r,j')}) \circ \Pi_1^{(r,j)} \ . \quad (4.7)$$

The next lemma shows that an encoded dual AES subround can be represented by an encoded AES subround (Def. 23) using the same key bytes:

Lemma 3. *An encoded dual AES subround* $\mathit{AES}_{enc}^{(r,j,\Delta_{r,j})}$ *is an encoded AES subround* $\mathit{AES}_{enc}^{(r,j)}$ *as in Def. 23 with*

$$P_i^{(1,j)} = A_i^{(1,j)} \quad \textit{and} \quad P_i^{(r,j)} = \Delta_{r-1,isr(i',j')}^{-1} \circ A_i^{(r,j)} \quad \textit{if} \quad 2 \leq r \leq 9 \ ,$$

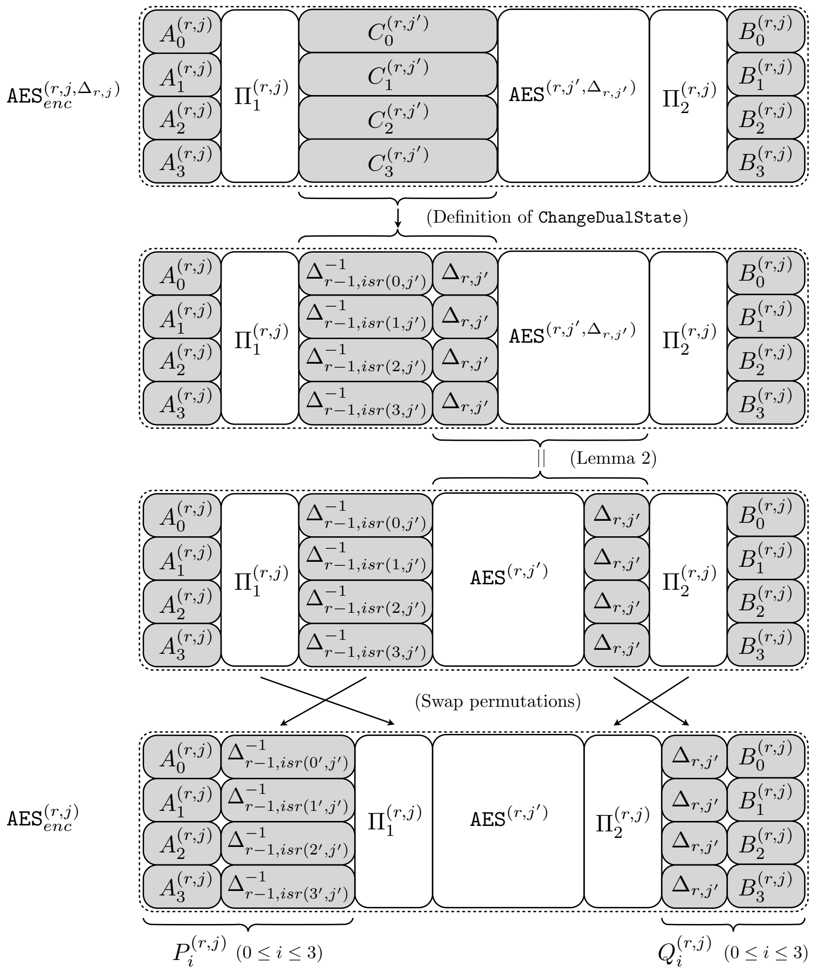
and

$$Q_i^{(r,j)} = B_i^{(r,j)} \circ \Delta_{r,j'} \ ,$$

for $0 \leq i,j \leq 3$ *and* $1 \leq r \leq 9$.

Proof. The proof is given for the case $2 \leq r \leq 9$ (Fig. 4.4); similar reasoning applies to the case $r = 1$. From the definition of the `ChangeDualState` operation (see Step 1.2 of Phase 1 of Karroumi's implementation) it follows that

$$(C_0^{(r,j')}, C_1^{(r,j')}, C_2^{(r,j')}, C_3^{(r,j')}) = (\Delta_{r,j'}, \Delta_{r,j'}, \Delta_{r,j'}, \Delta_{r,j'}) \circ (\Delta_{r-1,isr(0,j')}^{-1}, \Delta_{r-1,isr(1,j')}^{-1}, \Delta_{r-1,isr(2,j')}^{-1}, \Delta_{r-1,isr(3,j')}^{-1}) \quad \text{if } 2 \leq r \leq 9 \ ,$$



$\texttt{AES}_{enc}^{(r,j,\Delta_{r,j})}$
$A_0^{(r,j)}$ $A_1^{(r,j)}$ $A_2^{(r,j)}$ $A_3^{(r,j)}$
$\Pi_1^{(r,j)}$
$C_0^{(r,j')}$ $C_1^{(r,j')}$ $C_2^{(r,j')}$ $C_3^{(r,j')}$
$\texttt{AES}^{(r,j',\Delta_{r,j'})}$
$\Pi_2^{(r,j)}$
$B_0^{(r,j)}$ $B_1^{(r,j)}$ $B_2^{(r,j)}$ $B_3^{(r,j)}$
(Definition of ChangeDualState)
$\Delta_{r-1,isr(0,j')}^{-1}$ $\Delta_{r-1,isr(1,j')}^{-1}$ $\Delta_{r-1,isr(2,j')}^{-1}$ $\Delta_{r-1,isr(3,j')}^{-1}$
$\Delta_{r,j'}$
(Lemma 2)
$\texttt{AES}^{(r,j')}$
(Swap permutations)
$\texttt{AES}_{enc}^{(r,j)}$
$\Delta_{r-1,isr(0',j')}^{-1}$ $\Delta_{r-1,isr(1',j')}^{-1}$ $\Delta_{r-1,isr(2',j')}^{-1}$ $\Delta_{r-1,isr(3',j')}^{-1}$
$P_i^{(r,j)}$ $(0 \leq i \leq 3)$
$Q_i^{(r,j)}$ $(0 \leq i \leq 3)$

Figure 4.4: An encoded dual AES subround is an encoded AES subround.

for $0 \leq j \leq 3$. Substituting the above expression for the `ChangeDualState` operation in (4.7) and applying Lemma 2 gives

$$\overline{\texttt{AES}}^{(r,j,\Delta_{r,j})} = \Pi_2^{(r,j)} \circ (\Delta_{r,j'}, \Delta_{r,j'}, \Delta_{r,j'}, \Delta_{r,j'}) \circ \texttt{AES}^{(r,j')} \circ$$
$$(\Delta^{-1}_{r-1,isr(0,j')}, \Delta^{-1}_{r-1,isr(1,j')}, \Delta^{-1}_{r-1,isr(2,j')}, \Delta^{-1}_{r-1,isr(3,j')}) \circ \Pi_1^{(r,j)} \ .$$

Observe that $\Pi_2^{(r,j)}$ and $(\Delta_{r,j'}, \Delta_{r,j'}, \Delta_{r,j'}, \Delta_{r,j'})$ commute and thus can be swapped. By applying the equation

$$(\Delta^{-1}_{r-1,isr(0,j')}, \Delta^{-1}_{r-1,isr(1,j')}, \Delta^{-1}_{r-1,isr(2,j')}, \Delta^{-1}_{r-1,isr(3,j')}) \circ \Pi_1^{(r,j)} =$$
$$\Pi_1^{(r,j)} \circ (\Delta^{-1}_{r-1,isr(0',j')}, \Delta^{-1}_{r-1,isr(1',j')}, \Delta^{-1}_{r-1,isr(2',j')}, \Delta^{-1}_{r-1,isr(3',j')}) \ ,$$

one gets the result of Lemma 3. □

From the discussion above it follows that Karroumi's white-box AES implementation and the white-box AES implementation of Chow et al. are the same. As a consequence, Karroumi's white-box AES implementation is vulnerable to the attack it was designed to resist.

4.3 Conclusion

The BGE attack [13] on the white-box AES implementation of Chow et al. extracts the AES key from such an implementation with a work factor of 2^{30}. Taking Tolhuizen's improvement of the most time-consuming phase of the BGE attack as the starting point, we have presented several improvements of the other phases of the BGE attack. When all improvements are combined, the work factor of the original BGE attack is reduced to 2^{22}. Additionally, we have proposed a method to extract the external encodings as well, which slightly increased the overall work factor to 2^{23}. Unlike the original BGE attack, the use of non-affine white-box encodings and the randomization in the order of the bytes of the intermediate AES results in the white-box implementation have a negligible contribution to the overall work factor of our improved BGE attack.

Karroumi's white-box AES implementation [53] was designed to withstand the BGE attack. We have shown that Karroumi's implementation in fact belongs to the class of white-box AES implementations specified by Chow et al. in [23]. As a result, Karroumi's white-box AES implementation remains vulnerable to the BGE attack and our improved version of this attack.

Chapter 5

Cryptanalysis of the Xiao-Lai White-Box AES Implementation

In 2009, Xiao and Lai [107] proposed a new white-box AES implementation that is claimed to be resistant against the BGE attack [13]. However, the white-box attack of Michiels et al. [75] can still be applied to their implementation. But, due to its generic nature, Michiels et al.'s attack is not optimized with regard to the Xiao-Lai white-box AES implementation (i.e., it has an estimated work factor of at least 2^{49}), so there is room for improvement.

In this chapter, we present a practical attack on the white-box AES implementation of Xiao and Lai. This research was published in [35]. We show how properties of AES as well as of the Xiao-Lai white-box implementation itself can be exploited in order to obtain a practical non-generic attack. The presented cryptanalysis efficiently extracts the embedded AES key together with the external encodings from the Xiao-Lai white-box AES implementation with a work factor of 2^{32}. A modified variant of the linear equivalence (LE) algorithm presented by Biryukov et al. [14] is used as a building block. Additionally, we consider design generalizations of the Xiao-Lai white-box AES implementation and discuss their impact on the work factor of our cryptanalysis.

5.1 The Xiao-Lai White-Box AES Implementation

The approach of Xiao and Lai [107] in designing a secure white-box AES implementation (i.e., resistant to the BGE attack) differs from Karroumi's white-box AES design (Sect. 4.2.1) in the following ways:

- Karroumi introduced secrecy in the description of AES by means of generating a dual AES cipher and keeping the randomly chosen 'dual' transformations secret. Next, he applied the white-box techniques of Chow et al. in the same way as specified in [23] to the dual AES cipher.
- Xiao and Lai use the same description of AES (Fig. 2.1b, p. 21) as the one used by Chow et al., however, they apply the white-box techniques of Chow et al. in a specific way different from the one specified in [23]. The differences are listed in the following:
    1. all secret white-box encodings are solely $\mathbb{F}_2$-linear. As a result, all XOR operations are executed on encoded data, i.e., they are not represented as a network of encoded nibble XOR tables;
    2. the secret white-box encodings operate on at least two bytes simultaneously instead of at least four bits (in the case of non-linear encodings) or at least a byte (in the case of linear encodings) in [23];
    3. the `ShiftRows` operation is implemented explicitly as a matrix-vector multiplication instead of implicitly including it into the data-flow of the white-box implementation as in [23].

Recall from Sect. 3.2 that the process of constructing a white-box implementation using the generic white-box techniques of Chow et al. comprises two phases. The two corresponding phases of the Xiao-Lai white-box AES implementation [107] are described below, given the alternative description of AES (Fig. 2.1b, p. 21). This chapter assumes throughout and without loss of generality that AES-128 is used.

Phase 1 and 2: Translate AES into a series of lookup tables and apply secret bijective $\mathbb{F}_2$-linear white-box encodings

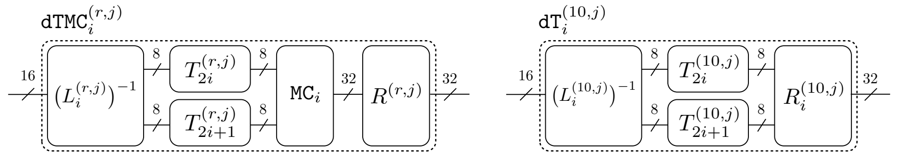
First, as was also the case for Chow et al.'s white-box AES implementation (Sect. 3.3.2), the `AddRoundKey` and `SubBytes` operations of all AES rounds ($1 \leq r \leq 10$) are composed, resulting in 16 8-bit bijective T-boxes for each round. The T-boxes are defined as

$$\begin{aligned} T_i^{(r,j)}(x) &= S(x \oplus \hat{k}_i^{(r,j)}) && \text{for } 0 \leq i,j \leq 3 \text{ and } 1 \leq r \leq 9\ , \\ T_i^{(10,j)}(x) &= S(x \oplus \hat{k}_i^{(10,j)}) \oplus k_i^{(11,j)} && \text{for } 0 \leq i,j \leq 3\ . \end{aligned}$$

Second, concerning the `MixColumns` operation of rounds $1 \leq r \leq 9$, the 4×4 matrix `MC` over $\mathbb{F}_{256}$ is split into two 4×2 submatrices over $\mathbb{F}_{256}$: $\texttt{MC}_0$ is defined as the first two columns of $\texttt{MC}$ and $\texttt{MC}_1$ is defined as the remaining two columns of $\texttt{MC}$. Using this notation, the `MixColumns` matrix-vector multiplication is decomposed into a XOR of two 32-bit values and is given by

$$\begin{pmatrix} \text{state}_{0,j} \\ \text{state}_{1,j} \\ \text{state}_{2,j} \\ \text{state}_{3,j} \end{pmatrix} \leftarrow \texttt{MC}_0 \cdot \begin{pmatrix} \text{state}_{0,j} \\ \text{state}_{1,j} \end{pmatrix} \oplus \texttt{MC}_1 \cdot \begin{pmatrix} \text{state}_{2,j} \\ \text{state}_{3,j} \end{pmatrix} \quad \text{for } j = 0,1,2,3 \ .$$

The reason for this split into two submatrices is the fact that all $\mathbb{F}_2$-linear white-box encodings will operate on at least two bytes simultaneously (see below). Recall from Sect. 3.3.2 that for the white-box AES implementation of Chow et al., $\texttt{MC}$ was split into four submatrices since the $\mathbb{F}_2$-linear encodings operated on at least a byte and the non-linear encodings on at least a nibble.



(a) Composition of T-boxes and `MixColumns` operation for rounds $1 \leq r \leq 9$.

(b) Composition of final round T-boxes.

Figure 5.1: Two different types of encoded lookup tables of the Xiao-Lai white-box AES implementation.

For rounds $1 \leq r \leq 9$, the T-boxes and `MixColumns` operation are composed as depicted in Fig. 5.1a. Observe that this results in eight lookup tables per round, each table mapping 16 bits to 32 bits. To prevent an attacker from extracting the AES round keys from these tables, each table is composed with two secret white-box encodings $\left(L_i^{(r,j)}\right)^{-1}$ and $R^{(r,j)}$ as depicted in Fig. 5.1a:

The input of each table is encoded by a 16-bit linear mixing bijection $\left(L_i^{(r,j)}\right)^{-1}$ (represented by a non-singular 16×16 matrix over $\mathbb{F}_2$);

The output of each table is encoded by a 32-bit linear mixing bijection $R^{(r,j)}$ (represented by a non-singular 32×32 matrix over $\mathbb{F}_2$).

The resulting tables are referred to as $\texttt{dTMC}_i^{(r,j)}$ ($i = 0,1$ and $j = 0,1,2,3$) in the following, and map 16 bits to 32 bits. Observe that $R^{(r,j)}$ lacks the index i as this encoding is required to be the same for both $\texttt{dTMC}_i^{(r,j)}$ ($i = 0,1$) tables associated with each j.

For the final round ($r = 10$), the composition (Fig. 5.1b) is slightly different due to the omission of the `MixColumns` operation. Here, instead of the matrix MC, the output white-box encoding $R^{(10,j)}$ (represented by a non-singular 32×32 binary matrix) is split into two 32×16 submatrices over $\mathbb{F}_2$: $R_0^{(10,j)}$ is defined as the first 16 columns of $R^{(10,j)}$ and $R_1^{(10,j)}$ contains the remaining 16 columns. The resulting final round tables are referred to as $\mathtt{dT}_i^{(10,j)}$ ($i = 0, 1$ and $j = 0, 1, 2, 3$), each mapping 16 bits to 32 bits.

Third, a 128×128 non-singular matrix $M^{(r)}$ over $\mathbb{F}_2$ is associated with each round r ($2 \leq r \leq 10$). If SR denotes the 128×128 non-singular matrix over $\mathbb{F}_2$ representing the `ShiftRows` operation, then the matrix $M^{(r)}$ is defined by

$$M^{(r)} = \left(L_0^{(r,0)}, L_1^{(r,0)}, L_0^{(r,1)}, L_1^{(r,1)}, L_0^{(r,2)}, L_1^{(r,2)}, L_0^{(r,3)}, L_1^{(r,3)}\right) \circ \mathtt{SR} \circ \left(\left(R^{(r-1,0)}\right)^{-1}, \left(R^{(r-1,1)}\right)^{-1}, \left(R^{(r-1,2)}\right)^{-1}, \left(R^{(r-1,3)}\right)^{-1}\right) ,$$

for $r = 2, 3, \ldots, 10$.

Fourth, external secret white-box encodings are applied at the boundaries of the AES cipher as in Chow et al.'s white-box AES implementation [23], i.e. before the first round (plaintext) and after the final round (ciphertext). The input and output external encodings are defined as 128-bit linear mixing bijections $\mathtt{IN}^{-1}$ and OUT, respectively, each encoding being represented by a non-singular 128×128 matrix over $\mathbb{F}_2$. Next, associated to the first and final round, the non-singular 128×128 matrices $M^{(1)}$ and $M^{(11)}$ over $\mathbb{F}_2$ are defined by

$$M^{(1)} = \left(L_0^{(1,0)}, L_1^{(1,0)}, L_0^{(1,1)}, L_1^{(1,1)}, L_0^{(1,2)}, L_1^{(1,2)}, L_0^{(1,3)}, L_1^{(1,3)}\right) \circ \mathtt{SR} \circ \mathtt{IN}^{-1} ,$$
$$M^{(11)} = \mathtt{OUT} \circ \left(\left(R^{(10,0)}\right)^{-1}, \left(R^{(10,1)}\right)^{-1}, \left(R^{(10,2)}\right)^{-1}, \left(R^{(10,3)}\right)^{-1}\right) ,$$

respectively.

Summary. Using these notations and definitions, the structure of the Xiao-Lai white-box AES implementation is depicted in Fig. 5.2. In the white-box implementation, the operations $M^{(r)}$ ($1 \leq r \leq 10$) and $M^{(11)}$ are implemented as matrix-vector multiplications over $\mathbb{F}_2$ and the operations $\mathtt{dTMC}_i^{(r,j)}$ ($1 \leq r \leq 9$) and $\mathtt{dT}_i^{(10,j)}$ for $i = 0, 1$ and $j = 0, 1, 2, 3$ are implemented as lookup tables. Note that the output of two tables, which corresponds to the linearly encoded output of $\mathtt{MC}_0$ and $\mathtt{MC}_1$, is added modulo two (i.e., XOR'ed) in the white-box implementation. Observe that the white-box AES implementation is functionally equivalent to an encoded version of AES due to the application of the external encodings $\mathtt{IN}^{-1}$ and OUT.

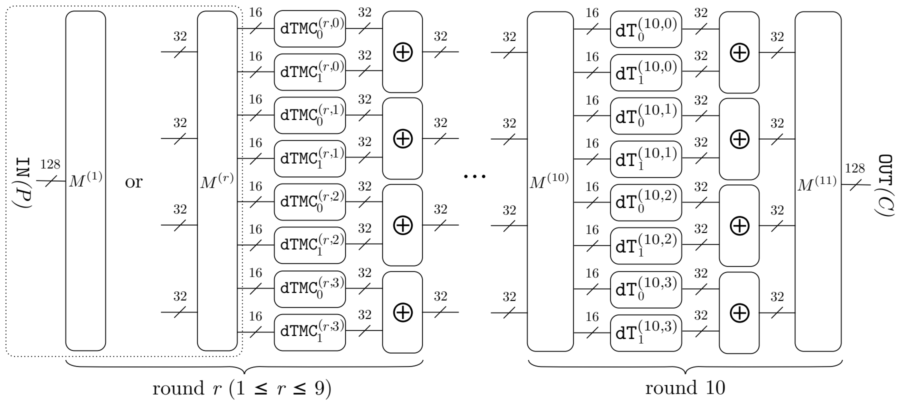


Figure 5.2: The Xiao-Lai white-box AES implementation.

Table 5.1: Overall size and performance of the Xiao-Lai white-box AES implementation.

Size				
Lookup Table / Binary Matrix				Total Size
#	Type		Size	
72	$\mathtt{dTMC}_i^{(r,j)}$	(16-to-32 bit)	18 432 kB	20 502 kB
8	$\mathtt{dT}_i^{(10,j)}$	(16-to-32 bit)	2048 kB	
11	$M^{(r)}, M^{(11)}$	(128 × 128)	22 kB	

Performance		
# of Table Lookups	# of XORs	# of Matrix Mult.
80	40	11

Table 5.1 gives an overview of the size and performance of the Xiao-Lai white-box AES implementation as specified in [107]. The performance is expressed in the number of table lookups, 32-bit XOR operations and matrix-vector multiplications over $\mathbb{F}_2$. Observe that the total implementation size is significantly increased when compared to the white-box AES implementation of Chow et al., i.e., 20 502 kB versus 752 kB. This is mainly caused by the 16-bit input size of all encoded lookup tables occurring in the Xiao-Lai white-box AES implementation; this has a noticeable impact on the implementation size since the storage requirement of a lookup table is exponential in its input size (Property 1, p. 16). On the positive side, the number of table lookups has significantly decreased, i.e., 80 versus 3008, because the XOR operations can be executed on encoded data (as a result of solely $\mathbb{F}_2$-linear encodings) and

hence need not be implemented as a network of encoded nibble XOR tables; a second reason is that each lookup table involves two bytes of the AES state simultaneously. As is discussed later in Chapter 7, the Xiao-Lai white-box AES implementation has the best performance of all white-box AES implementations considered in this thesis, and can moreover compete with the performance of the standard black-box software AES implementation (Sect. 2.3.2).

In [107], Xiao and Lai argue that their white-box AES implementation is resistant against the BGE attack [13]; their argument is highlighted in Sect. 5.5. However, they did not consider the generic white-box attack of Michiels et al. (Sect. 3.5.4), and as is explained in Sect. 5.5.1, this attack can be applied to their implementation. But, due to its generic nature, Michiels et al.'s attack is not optimized to cryptanalyze the Xiao-Lai white-box AES implementation (i.e., the preliminary estimation of the corresponding work factor is rather large, namely at least 2^{49}). Therefore, in Sect. 5.3, we present a practical non-generic attack on the Xiao-Lai white-box AES implementation by exploiting specific properties of both AES as well as the white-box implementation itself; the attack has a work factor of 2^{32} which shows a drastic improvement over Michiels et al.'s attack. This optimization leads to the fact that our attack only requires a modified version of the linear equivalence (LE) algorithm for S-boxes presented by Biryukov et al. in [14], whereas Michiels et al.'s attack requires the more complex affine equivalence (AE) algorithm for S-boxes [14] combined with a linear equivalence algorithm for matrices (LEPM) [75, Def. 6].

5.2 Linear Equivalence Algorithm

This section describes aspects of the linear equivalence algorithm (LE) proposed by Biryukov et al. [14] that are relevant to the cryptanalysis of the Xiao-Lai white-box AES implementation presented in Sect. 5.3.

Definition 29 (Linear equivalence of S-boxes)**.** *Two permutations on n bits (or S-boxes) S_1 and S_2 are called linearly equivalent if a pair of linear mappings (A, B) from n to n bits exists such that $S_2 = B \circ S_1 \circ A$.*

A pair (A, B) as in this definition is referred to as a *linear equivalence.* Notice that both linear mappings A and B of a linear equivalence are bijective. If $S_1 = S_2$, then the linear equivalences are referred to as *linear self-equivalences.*

The linear equivalence problem is: given two n-bit bijective S-boxes S_1 and S_2, determine if S_1 and S_2 are linearly equivalent. An algorithm for solving the linear equivalence problem is presented in [14] and is referred to as the linear equivalence algorithm (LE). The inputs to the algorithm are S_1 and S_2, and the output is either a linear equivalence (A, B) if S_1 and S_2 are linearly equivalent, or a message that such a linear equivalence does not exist. For an in-depth description of LE, refer to Biryukov et al. [14]. Below a variant of LE where it is assumed that both given S-boxes map 0 to itself, i.e., $S_1(0) = S_2(0) = 0$, is briefly described. This variant of LE is used as a building block for the cryptanalysis presented in Sect. 5.3.

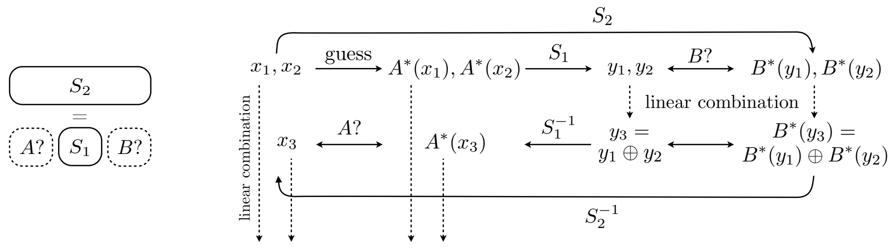


Figure 5.3: Illustration how the linear equivalence (LE) algorithm works.

To quote De Cannière [33, p. 76]: "The idea of LE is to guess the linear mapping A for as few input points as possible, and then use the linearity of the mappings A and B to follow the implications of these guesses as far as possible." Now, if $S_1(0) = S_2(0) = 0$, it is necessary to guess the mapping A for at least two distinct randomly selected point $x_1 \neq 0$ and $x_2 \neq 0$ in order to start LE, i.e., guess the values of $A(x_1)$ and $A(x_2)$. Based on these two initial guesses and the linearity of A and B (that are searched), candidates for the linear mappings A and B are incrementally built as far as possible; these candidate mappings are denoted by A^* and B^* in the following. The initial guesses $A^*(x_i)$ for the points x_i $(i = 1, 2)$ provide knowledge about B^* by computing $y_i = S_1(A^*(x_i))$ and $B^*(y_i) = S_2(x_i)$, which in turn provides possibly new information about A^* by computing the images of the linear combinations of y_i and $B^*(y_i)$ through S_1^{-1} and S_2^{-1}, respectively. This process is applied iteratively, where in each step of the process the linearity of the partially determined candidate mappings A^* and B^* is verified by a Gaussian elimination. Figure 5.3 illustrates this iterative process. If neither for A^* nor for B^* a set of n linearly independent inputs and outputs is obtained (and no linear inconsistencies occurred so far), it is necessary to guess the mapping A (or B) for an additional (not yet covered) point in order to continue LE.

If n linearly independent inputs and n linearly independent outputs to A^* are obtained, then a candidate for A (here also denoted by A^* for the sake of clarity) can be computed. Similar reasoning applies to B. Now, the correctness of the candidate linear equivalence (A^*, B^*) can be tested by verifying the relation $S_2 = B^* \circ S_1 \circ A^*$ for all possible inputs. If no candidate linear equivalence is found (due to linear inconsistencies occurred during the process), or if the candidate linear equivalence is incorrect, then the process is repeated with a different guess for $A(x_1)$ and/or for $A(x_2)$, and/or for any of the possibly additional guesses made during the execution of LE.

The original linear equivalence algorithm (LE) exits after finding one single linear equivalence, which already proves that both given S-boxes S_1 and S_2 are linearly equivalent. However, by running LE over all possible guesses, i.e., both initial guesses as well as the possibly additional guesses made during the execution of LE, also other linear equivalences (A, B) can be found. The work factor of this variant is at least $n^3 \cdot 2^{2n}$, i.e., a Gaussian elimination (n^3) for each possible pair of initial guesses (2^{2n}).

5.3 Cryptanalysis

This section presents a practical (non-generic) attack on the Xiao-Lai white-box AES implementation [107], efficiently extracting the embedded AES key and the external encodings with a work factor of 2^{32}. The cryptanalysis uses the linear equivalence algorithm (LE) as a building block; LE is described in Sect. 5.2. Additionally, the cryptanalysis exploits the internal structure of AES and the composition of the white-box implementation.

Assumption 1. *The cryptanalysis assumes that the order of the bytes of the intermediate results (states) of AES in the white-box implementation is known to the attacker. As a result, it suffices to extract one single round key that yields the AES key through the invertible AES key scheduling algorithm. The generic case in which the order of the bytes of the intermediate AES results is randomized in the white-box implementation is discussed in Sect. 5.4.*

5.3.1 Setup Phase

The cryptanalysis focuses on extracting the 128-bit first round key $\hat{k}^{(1)}$ contained within the eight key-dependent 16-to-32 bit lookup tables $\mathtt{dTMC}_i^{(1,j)}$ ($i = 0, 1$ and $j = 0, 1, 2, 3$) of the first round. Each table $\mathtt{dTMC}_i^{(1,j)}$ (Fig. 5.4a) is defined by

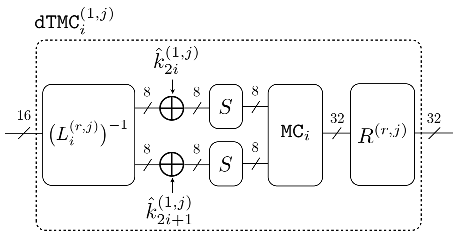
$$\mathtt{dTMC}_i^{(1,j)} = R^{(1,j)} \circ \mathtt{MC}_i \circ (S, S) \circ \oplus_{\left(\hat{k}_{2i}^{(1,j)} \| \hat{k}_{2i+1}^{(1,j)}\right)} \circ \left(L_i^{(1,j)}\right)^{-1} , \qquad (5.1)$$

where (S, S) denotes the 16-bit bijective S-box comprising two AES S-boxes in parallel. Given (5.1), the attacker knows that both S-boxes $S_1 = (S, S)$ and

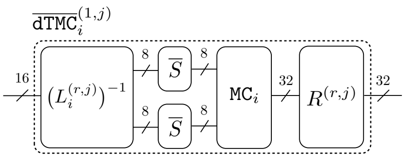
$S_2 = \mathtt{dTMC}_i^{(1,j)}$ are affine equivalent by the affine equivalence

$$(A, B) = \left(\oplus_{\left(\hat{k}_{2i}^{(1,j)} \| \hat{k}_{2i+1}^{(1,j)}\right)} \circ \left(L_i^{(1,j)}\right)^{-1}, R^{(1,j)} \circ \mathtt{MC}_i \right)$$

such that $S_2 = B \circ S_1 \circ A$. Note that only A is affine where the constant part equals the key-material contained within $\mathtt{dTMC}_i^{(1,j)}$. Hence by making $\mathtt{dTMC}_i^{(1,j)}$ key-independent (see Lemma 4), we can reduce the problem to finding linear instead of affine equivalences (where the latter is the case for the generic Michiels et al.'s attack), for which we apply the linear equivalence algorithm (LE).



(a) Key-material ($\hat{k}_{2i}^{(1,j)}$ and $\hat{k}_{2i+1}^{(1,j)}$)


$\overline{\mathtt{dTMC}}_i^{(1,j)}$
16
8
8
32
32
$\left(L_i^{(r,j)}\right)^{-1}$
$\overline{S}$
$\overline{S}$
$\mathtt{MC}_i$
$R^{(r,j)}$

(b) No key-material

Figure 5.4: Key-dependent table $\mathtt{dTMC}_i^{(1,j)}$ vs. key-independent table $\overline{\mathtt{dTMC}}_i^{(1,j)}$.

Lemma 4. *Given the key-dependent 16-to-32 bit lookup table* $\mathit{dTMC}_i^{(1,j)}$ *(defined by* (5.1) *and depicted in Fig. 5.4a), let* $x_{i,j}$ *be the 16-bit value such that* $\mathit{dTMC}_i^{(1,j)}(x_{i,j}) = 0$*, and let* $\overline{\mathit{dTMC}}_i^{(1,j)}$ *be defined as* $\overline{\mathit{dTMC}}_i^{(1,j)} = \mathit{dTMC}_i^{(1,j)} \circ \oplus_{x_{i,j}}$*. If* $\overline{S}$ *is defined as the 8-bit bijective S-box* $\overline{S} = S \circ \oplus_{52}$ *where* S *denotes the AES S-box, then*

$$\overline{\mathit{dTMC}}_i^{(1,j)} = R^{(1,j)} \circ \mathit{MC}_i \circ (\overline{S}, \overline{S}) \circ \left(L_i^{(1,j)}\right)^{-1} , \tag{5.2}$$

where $(\overline{S}, \overline{S})$ *denotes the 16-bit bijective S-box comprising two S-boxes* $\overline{S}$ *in parallel. The key-independent 16-to-32 bit lookup table* $\overline{\mathit{dTMC}}_i^{(1,j)}$ *is depicted in Fig. 5.4b.*

Proof. Given the fact that $\mathtt{dTMC}_i^{(1,j)}$ is encoded merely by linear input and output encodings $\left(L_i^{(1,j)}\right)^{-1}$ and $R^{(1,j)}$ (see (5.1)) and that $S(52) = 0$, the 16-bit value $x_{i,j}$ for which $\mathtt{dTMC}_i^{(1,j)}(x_{i,j}) = 0$ is given by

$$x_{i,j} = L_i^{(1,j)}\left(\left(\hat{k}_{2i}^{(1,j)} \oplus 52\right) \| \left(\hat{k}_{2i+1}^{(1,j)} \oplus 52\right)\right) , \tag{5.3}$$

In the rare case that $x_{i,j} = 0$, it immediately follows that both first round key bytes $\hat{k}_{2i}^{(1,j)}$ and $\hat{k}_{2i+1}^{(1,j)}$ are equal to 52. Now, based on $x_{i,j}$, one can construct the key-independent 16-to-32 bit lookup table $\overline{\mathtt{dTMC}}_i^{(1,j)}$ as follows:

$$\begin{aligned}\overline{\mathtt{dTMC}}_i^{(1,j)} &= \mathtt{dTMC}_i^{(1,j)} \circ \oplus_{x_{i,j}} \\ &= R^{(1,j)} \circ \mathtt{MC}_i \circ (S,S) \circ \oplus_{\left(\hat{k}_{2i}^{(1,j)} \| \hat{k}_{2i+1}^{(1,j)}\right)} \circ \oplus_{(L_i^{(1,j)})^{-1}(x_{i,j})} \circ \left(L_i^{(1,j)}\right)^{-1} \\ &= R^{(1,j)} \circ \mathtt{MC}_i \circ (S,S) \circ \oplus_{(\mathtt{52}\|\mathtt{52})} \circ \left(L_i^{(1,j)}\right)^{-1} \\ &= R^{(1,j)} \circ \mathtt{MC}_i \circ (\overline{S},\overline{S}) \circ \left(L_i^{(1,j)}\right)^{-1} \ . \qquad \square\end{aligned}$$

Note that the obtained key-independent tables $\overline{\mathtt{dTMC}}_i^{(1,j)}$ ($i = 0, 1$ and $j = 0, 1, 2, 3$) defined by (5.2) map 0 to itself, i.e., $\overline{\mathtt{dTMC}}_i^{(1,j)}(0) = 0$. This follows from the fact that the 8-bit bijective S-box $\overline{S}$ maps 0 to itself as well.

Linear equivalence algorithm (LE). By applying LE to the 16-bit bijective S-box $S_1 = (\overline{S}, \overline{S})$ and the key-independent 16-to-32 bit lookup table $S_2 = \overline{\mathtt{dTMC}}_i^{(1,j)}$ obtained through Lemma 4, a total of 128 linear equivalences

$$(A, B) = \left(A^s \circ \left(L_i^{(1,j)}\right)^{-1}, R^{(1,j)} \circ \mathtt{MC}_i \circ B^s\right)$$

can be found, where (A^s, B^s) denote the 128 linear self-equivalences of $(\overline{S}, \overline{S})$ (see Property 5 below), such that

$$\begin{aligned}\overline{\mathtt{dTMC}}_i^{(1,j)} &= R^{(1,j)} \circ \mathtt{MC}_i \circ (\overline{S},\overline{S}) \circ \left(L_i^{(1,j)}\right)^{-1} \\ &= R^{(1,j)} \circ \mathtt{MC}_i \circ B^s \circ (\overline{S},\overline{S}) \circ A^s \circ \left(L_i^{(1,j)}\right)^{-1} \\ &= B \circ (\overline{S},\overline{S}) \circ A \ .\end{aligned}$$

Property 5 (Linear self-equivalences of $(\overline{S}, \overline{S})$)**.** *Let the AES S-box S be defined as $S(x) = A(x^{-1})$ where A is a bijective affine mapping on $\mathbb{F}_2^8$ and x^{-1} denotes the inverse of x in $\mathbb{F}_{256}$ as defined in FIPS 197 [69] with* 00 $^{-1}$ = 00*, and let the 8-bit bijective S-box $\overline{S}$ be defined as $\overline{S} = S \circ \oplus_{\mathtt{52}}$. Further, let $m_c : \mathbb{F}_{256} \to \mathbb{F}_{256}$ with $c \in \mathbb{F}_{256}^*$ be defined by $m_c(x) = c \otimes x$, and let $f_t : \mathbb{F}_{256} \to \mathbb{F}_{256}$ be the automorphisms of $\mathbb{F}_{256}$ over $\mathbb{F}_2$ defined by $f_t(x) = x^{2^t}$ for $0 \le t \le 7$. If Φ_l denotes the set of exactly $\#_l = 8$ linear self-equivalences (α, β) of $\overline{S}$ such that $\overline{S} = \beta \circ \overline{S} \circ \alpha$ and is defined by*

$$\Phi_l = \left\{(\alpha = m_c \circ f_t, \beta = A \circ f_t^{-1} \circ m_c \circ A^{-1}) \mid (t, c) \in \mathcal{S}_l\right\} \qquad \textit{with}$$

$$\mathcal{S}_l = \{(0, \mathtt{01}), (1, \mathtt{05}), (2, \mathtt{13}), (3, \mathtt{60}), (4, \mathtt{55}), (5, \mathtt{f6}), (6, \mathtt{b2}), (7, \mathtt{66})\} \ ,$$

then the 16-bit bijective S-box comprising two identical S-boxes $\overline{S}$ in parallel, i.e. $(\overline{S},\overline{S})$, has $2 \cdot \#_l^2 = 128$ linear self-equivalences denoted by the pair of 16-bit bijective linear mappings (A^s, B^s) such that $(\overline{S},\overline{S}) = B^s \circ (\overline{S},\overline{S}) \circ A^s$, with the following diagonal structure:

$$A^s = \begin{pmatrix} \alpha_1 & 0_{8\times 8} \\ 0_{8\times 8} & \alpha_2 \end{pmatrix} , B^s = \begin{pmatrix} \beta_1 & 0_{8\times 8} \\ 0_{8\times 8} & \beta_2 \end{pmatrix} \textit{ or}$$

$$A^s = \begin{pmatrix} 0_{8\times 8} & \alpha_1 \\ \alpha_2 & 0_{8\times 8} \end{pmatrix} , B^s = \begin{pmatrix} 0_{8\times 8} & \beta_2 \\ \beta_1 & 0_{8\times 8} \end{pmatrix} ,$$

where both $(\alpha_1, \beta_1), (\alpha_2, \beta_2) \in \Phi_l$ and where $0_{8\times 8}$ denotes the 8×8 zero matrix. We applied the linear equivalence algorithm (implemented in C++) to $S_1 = S_2 = (\overline{S},\overline{S})$ and found exactly these 128 linear self-equivalences.

Since in this case both S-boxes $S_1 = (\overline{S},\overline{S})$ and $S_2 = \overline{\mathtt{dTMC}}_i^{(1,j)}$ map 0 to itself, recall from Sect. 5.2 that at least two initial 16-bit guesses $A(x_n)$ for two distinct points $x_n \neq 0$ $(n = 1, 2)$ of A are necessary to execute LE, and hence the work factor becomes at least 2^{44}, i.e., $n^3 \cdot 2^{2n}$ for $n = 16$. Furthermore, the attacker is only interested in one single linear equivalence, referred to as the *desired linear equivalence* in the following, and denoted by

$$(A, B)_d = \left(\left(L_i^{(1,j)}\right)^{-1}, R^{(1,j)} \circ \mathtt{MC}_i \right) .$$

Observe that the desired linear equivalence corresponds to the one with the linear self-equivalence $(A^s, B^s) = (I_{16}, I_{16})$, where I_{16} denotes the 16-bit identity matrix over $\mathbb{F}_2$. The attacker's goal is to recover $(A, B)_d$ since it contains the secret linear input encoding $\left(L_i^{(1,j)}\right)^{-1}$ that allows him to extract both first round key bytes $\hat{k}_{2i}^{(1,j)}$ and $\hat{k}_{2i+1}^{(1,j)}$ out of the 16-bit value $x_{i,j}$ given by (5.3).

Our Goal. Below, we present a way how to modify the linear equivalence algorithm when applied to $S_1 = (\overline{S},\overline{S})$ and $S_2 = \overline{\mathtt{dTMC}}_i^{(1,j)}$ such that only the single desired linear equivalence $(A, B)_d$ is given as output. At the same time, the work factor decreases as well. This modification exploits both the internal structure of AES as well as the composition of the white-box implementation.

5.3.2 Phase 1: Obtain leaked information about the linear input encoding $(L_i^{(1,j)})^{-1}$.

Due to the inherent structure of the Xiao-Lai white-box AES implementation, partial information about the linear input encoding $\left(L_i^{(1,j)}\right)^{-1}$ of the key-independent tables $\overline{\texttt{dTMC}}_i^{(1,j)}$ of the first round is leaked. In Phase 2, this leaked information is used to modify the linear equivalence algorithm in order to achieve the goal described above.

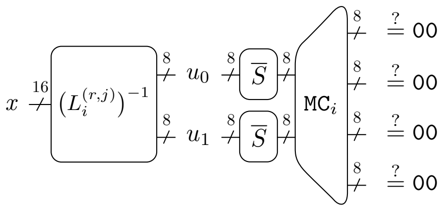


Figure 5.5: Exploited vulnerability of the Xiao-Lai white-box AES implementation: verify if each of the four output bytes of $\texttt{MC}_i$ equals 00.

For each linear input encoding $\left(L_i^{(1,j)}\right)^{-1}$ ($i = 0, 1$ and $j = 0, 1, 2, 3$), this leaked information comprises four sets of 2^8 16-bit encoded values x for which the underlying unencoded bytes u_0, u_1 (i.e., $\left(L_i^{(1,j)}\right)^{-1}(x) = u_0 \| u_1$) share a known 8-bit bijective function. This information is leaked out of the white-box implementation because the attacker can separately monitor whether or not each of the four output bytes of the submatrix $\texttt{MC}_i$ contained within $\overline{\texttt{dTMC}}_i^{(1,j)}$ (Fig. 5.4b) equals 00. Figure 5.5 illustrates this fact. In the following, we describe how the attacker obtains this ability in the Xiao-Lai white-box AES implementation and how this enables him to retrieve leaked information about $\left(L_i^{(1,j)}\right)^{-1}$ of each $\overline{\texttt{dTMC}}_i^{(1,j)}$ table for $i = 0, 1$ and $j = 0, 1, 2, 3$.

First, one builds an implementation that only consists of the single key-independent table $\overline{\texttt{dTMC}}_i^{(1,j)}$ followed by the matrix multiplication over $\mathbb{F}_2$ with $M^{(2)}$. This implementation is depicted in detail in Fig. 5.6 for $(i, j) = (0, 2)$, where the internal states U, V and Y are indicated as well: the 2-byte state $U = (u_0, u_1)$ corresponds to the 2-byte input to $(\overline{S}, \overline{S})$ and the 4-byte state $V = (v_0, v_1, v_2, v_3)$ corresponds to the 4-byte output of $\texttt{MC}_i$. Hence, the relation between U and V is given by

$$mc_{l,0}^i \otimes \overline{S}(u_0) \oplus mc_{l,1}^i \otimes \overline{S}(u_1) = v_l \quad \text{for} \quad l = 0, 1, 2, 3 \ ,$$

where the pair of bytes $(mc_{l,0}^i, mc_{l,1}^i)$ corresponds to the $\texttt{MixColumns}$ coefficients on row l of $\texttt{MC}_i$ for $l = 0, 1, 2, 3$, i.e., $(mc_{l,0}^i, mc_{l,1}^i) \in \mathcal{S}_{\texttt{MC}}$ with

$$\mathcal{S}_{\texttt{MC}} = \{(\texttt{02}, \texttt{03}), (\texttt{01}, \texttt{02}), (\texttt{01}, \texttt{01}), (\texttt{03}, \texttt{01})\} \ .$$

Then, the 16-byte input to $M^{(2)}$ is given by all zeros except for the output of the $\overline{\texttt{dTMC}}_i^{(1,j)}$ table. This is illustrated in our example (Fig. 5.6). This

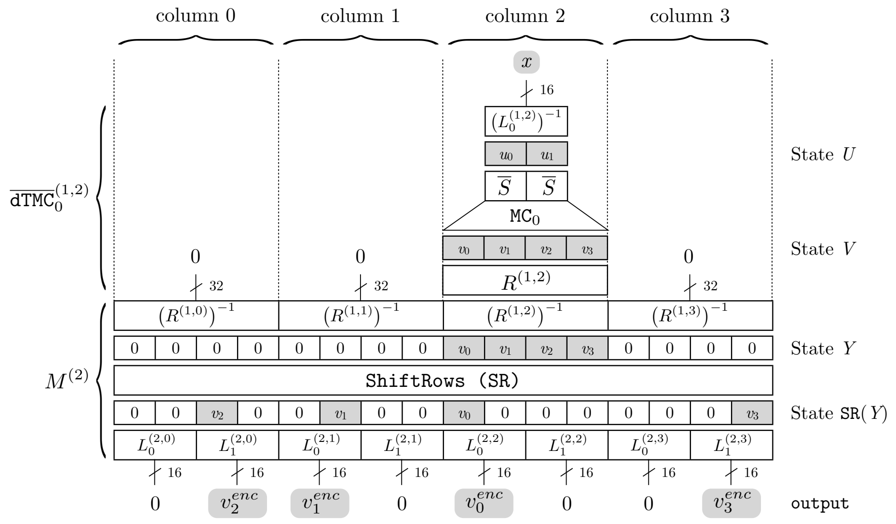

column 0
column 1
column 2
column 3
x
16
$(L_0^{(1,2)})^{-1}$
u_0 u_1
State U
$\overline{S}$ $\overline{S}$
$\overline{\mathtt{dTMC}}_0^{(1,2)}$
$\mathtt{MC}_0$
v_0 v_1 v_2 v_3
State V
$R^{(1,2)}$
32
$(R^{(1,0)})^{-1}$ $(R^{(1,1)})^{-1}$ $(R^{(1,2)})^{-1}$ $(R^{(1,3)})^{-1}$
State Y
$M^{(2)}$
ShiftRows (SR)
State SR(Y)
$L_0^{(2,0)}$ $L_1^{(2,0)}$ $L_0^{(2,1)}$ $L_1^{(2,1)}$ $L_0^{(2,2)}$ $L_1^{(2,2)}$ $L_0^{(2,3)}$ $L_1^{(2,3)}$
v_2^{enc} v_1^{enc} v_0^{enc} v_3^{enc}
output

Figure 5.6: Implementation associated with $\overline{\mathtt{dTMC}}_i^{(1,j)}$ for $(i,j) = (0,2)$: identifying the four values v_l^{enc} $(l = 0,1,2,3)$ in order to build the sets $\mathcal{S}_l^{(i,j)}$.

ensures that the corresponding values of the state Y remain zero except for the 4-byte state V (which corresponds to the unencoded output of $\overline{\mathtt{dTMC}}_i^{(1,j)}$). The `ShiftRows` operation ensures that the four bytes v_l $(l = 0,1,2,3)$ of V are spread over all four columns of the internal (AES) state $\mathtt{SR}(Y)$, which are then each encoded by a different linear encoding, i.e., v_l is encoded by $L_{\lfloor l/2 \rfloor}^{(2,sr(l,j))}$ for $l = 0,1,2,3$. Recall from Sect. 2.3.1 that the function $sr(l,j) = (j-l) \bmod 4$ defines the `ShiftRows` operation. Hence the `output` state (i.e., the output of $M^{(2)}$) contains four 16-bit 0-values, whereas the other four 16-bit output values v_l^{enc} $(l = 0,1,2,3)$ each correspond to one of the four bytes v_l of V in a linearly encoded form. Therefore, if $v_l^{enc} = 0$, then the associated byte $v_l = \mathtt{00}$ as well, such that we have a known 8-bit bijective function $f_l^{(i,j)}$ between the bytes u_0, u_1 of U, which is defined by

$$u_1 = f_l^{(i,j)}(u_0) \text{ with } f_l^{(i,j)} = \overline{S}^{-1} \circ \otimes_{(mc_{l,1}^i)^{-1}} \circ \otimes_{mc_{l,0}^i} \circ \overline{S} \ . \qquad (5.4)$$

This function follows out of the equation $mc_{l,0}^i \otimes \overline{S}(u_0) \oplus mc_{l,1}^i \otimes \overline{S}(u_1) = \mathtt{00}$.

Now, for the linear input encoding $\left(L_i^{(1,j)}\right)^{-1}$ of $\overline{\mathtt{dTMC}}_i^{(1,j)}$, four sets $\mathcal{S}_l^{(i,j)}$ $(l = 0,1,2,3)$ are built as follows. First associate one of the four values v_l^{enc} with each set $\mathcal{S}_l^{(i,j)}$. Then, for each set $\mathcal{S}_l^{(i,j)}$, store the 16-bit value x, given as input to $\overline{\mathtt{dTMC}}_i^{(1,j)}$, for which the associated output value $v_l^{enc} = 0$. Do this for all $x \in \mathbb{F}_2^{16}$. As a result, each set $\mathcal{S}_l^{(i,j)}$ is composed of 2^8 16-bit encoded values x for which the underlying unencoded bytes u_0, u_1 share the known bijective function $f_l^{(i,j)}$ given by (5.4):

$$\mathcal{S}_l^{(i,j)} = \left\{ x \in \mathbb{F}_2^{16} \mid \left(L_i^{(1,j)}\right)^{-1}(x) = u_0 \| u_1 \wedge u_1 = f_l^{(i,j)}(u_0) \right\} \ , \tag{5.5}$$

with $|\mathcal{S}_l^{(i,j)}| = 2^8$. So with each set $\mathcal{S}_l^{(i,j)}$ $(l = 0, 1, 2, 3)$, a known bijective function $f_l^{(i,j)}$ is associated.

5.3.3 Phase 2: Find the desired linear equivalence $(A, B)_d$ and retrieve the linear input encoding $(L_i^{(1,j)})^{-1}$.

After Phase 1, four sets $\mathcal{S}_l^{(i,j)}$ $(l = 0, 1, 2, 3)$ defined by (5.5) are obtained, associated with the secret linear input encoding $\left(L_i^{(1,j)}\right)^{-1}$ of each $\overline{\texttt{dTMC}}_i^{(1,j)}$ table ($i = 0, 1$ and $j = 0, 1, 2, 3$) of the first round. For each element $x \in \mathcal{S}_l^{(i,j)}$, the underlying unencoded bytes u_0, u_1 share a specific known bijective function $f_l^{(i,j)}$ given by (5.4). Now, by exploiting this leaked information about $\left(L_i^{(1,j)}\right)^{-1}$, we present an efficient algorithm for computing the desired linear equivalence

$$(A, B)_d = \left(\left(L_i^{(1,j)}\right)^{-1}, R^{(1,j)} \circ \texttt{MC}_i \right) \ .$$

This enables the attacker to obtain the secret linear input encoding $A = \left(L_i^{(1,j)}\right)^{-1}$ of $\overline{\texttt{dTMC}}_i^{(1,j)}$, which also corresponds to the input encoding of $\texttt{dTMC}_i^{(1,j)}$.

Algorithm for Finding the Desired Linear Equivalence $(\mathbf{A}, \mathbf{B})_\mathbf{d}$

Since $A = \left(L_i^{(1,j)}\right)^{-1}$ in the desired linear equivalence, we exploit the leaked information about $\left(L_i^{(1,j)}\right)^{-1}$ obtained in Phase 1 in order to make the two initial guesses $A(x_n)$ for two distinct points $x_n \neq 0$ $(n = 1, 2)$ of A. Only two out of four sets $\mathcal{S}_l^{(i,j)}$ are considered, i.e., those where the pair of `MixColumns` coefficients $(mc_{l,0}^i, mc_{l,1}^i)$ of the associated function $f_l^{(i,j)}$ equals $(\texttt{01}, \texttt{02})$ or $(\texttt{02}, \texttt{03})$. We choose one of both sets and simply denote it by $\mathcal{S}$. Below we elaborate on which of the four sets $\mathcal{S}_l^{(i,j)}$ $(l = 0, 1, 2, 3)$ are most suitable to be used in the algorithm.

Now, select two distinct points $x_n \neq 0$ $(n = 1, 2)$ out of the chosen set, i.e., $x_n \in \mathcal{S}$. Based on definition (5.5) of $\mathcal{S}$, these points are defined as $x_n = L_i^{(1,j)}\big(u_n \| f(u_n)\big)$ for some unknown distinct 8-bit values $u_n \in \mathbb{F}_2^8 \setminus \{0\}$, where f denotes the known function associated with $\mathcal{S}$. Now, based on this knowledge and the fact that we want to find $A = \big(L_i^{(1,j)}\big)^{-1}$, the two initial guesses $A(x_n)$ are made as follows: $A(x_n) = a_n \| f(a_n)$ for all $a_n \in \mathbb{F}_2^8 \setminus \{0\}$ $(n = 1, 2)$. Hence, even if $A(x_n)$ is a 16-bit value, we only need to guess the 8-bit value a_n such that the total number of guesses becomes 2^{16} (i.e. $2^{\frac{2n}{2}}$ with $n = 16$). For each possible pair of initial guesses $\big(A(x_n) = a_n \| f(a_n)\big)_{n=1,2}$, LE is executed on $S_1 = (\overline{S}, \overline{S})$ and $S_2 = \overline{\mathtt{dTMC}_i^{(1,j)}}$. All found linear equivalences are stored in the set $\mathcal{S}_{\mathrm{LE}}$.

It is assumed that at least $(A, B)_d \in \mathcal{S}_{\mathrm{LE}}$, which occurs when $a_n = u_n$ for $n = 1, 2$. It is possible that additional linear equivalences $(A, B) = \big(A^s \circ \big(L_i^{(1,j)}\big)^{-1}, R^{(1,j)} \circ \mathtt{MC}_i \circ B^s\big)$ with $A^s \neq I_{16}$ can be found as well such that $|\mathcal{S}_{\mathrm{LE}}| > 1$. In that case, the procedure needs to be repeated for two new distinct points $x_n^* \neq 0$ $(n = 1, 2)$ out of the chosen set $\mathcal{S}$ that are also distinct from the original chosen points $x_n \neq 0$ $(n = 1, 2)$. This results in a second set $\mathcal{S}_{\mathrm{LE}}^*$. Assuming that all possible linear equivalences between $S_1 = (\overline{S}, \overline{S})$ and $S_2 = \overline{\mathtt{dTMC}_i^{(1,j)}}$ are given by $(A, B) = \big(A^s \circ \big(L_i^{(1,j)}\big)^{-1}, R^{(1,j)} \circ \mathtt{MC}_i \circ B^s\big)$ where the pair (A^s, B^s) denotes the linear self-equivalences of $(\overline{S}, \overline{S})$, it can be shown that for both considered sets, it is impossible that a linear equivalence with $A^s \neq I_{16}$ is given as output during both executions of the procedure (see below under 'Choice of Set $\mathcal{S}$'). Hence taking the intersection of both sets $\mathcal{S}_{\mathrm{LE}}$ and $\mathcal{S}_{\mathrm{LE}}^*$ results in the desired linear equivalence $(A, B)_d$.

Algorithm 1 gives a detailed description of the whole procedure. It has a work factor of 2^{29}, i.e., $2 \cdot n^3 \cdot 2^{\frac{2n}{2}}$ for $n = 16$.

Implementation. Algorithm 1 has been implemented in C++ and tests have been conducted on an Intel Core2 Quad @ 3.00 GHz. For the tests, we chose the set $\mathcal{S}_l^{(i,j)}$ where $(mc_{l,0}^i, mc_{l,1}^i) = (\mathtt{02}, \mathtt{03})$. We ran the implementation 3000 times in total, each time for different randomly chosen encodings $\big(L_i^{(1,j)}\big)^{-1}$ and $R^{(1,j)}$. Only four times the procedure '`search-LE`' needed to be repeated since two linear equivalences were found during the first execution. The implementation always succeeded in finding the single desired linear equivalence $(A, B)_d$, which required on average approximately one minute. It should be noted that the implementation was not optimized for speed, hence improvements are possible. The implementation also showed that each pair of initial guesses as defined above were sufficient in order to execute LE, i.e., no additional guesses were required.

Algorithm 1 Finding the desired linear equivalence $(A, B)_d$

```
Input: $S_1 = (\overline{S}, \overline{S})$, $S_2 = \overline{\mathtt{dTMC}}_i^{(1,j)}$, $\mathcal{S}$, $f$
Output: $(A, B)_d$
1: select two distinct points $x_1, x_2 \in \mathcal{S}$ with $x_n \neq 0$ $(n = 1, 2)$
2: call search-LE$(x_1, x_2, f, S_1, S_2) \to \mathcal{S}_{\mathrm{LE}}$
3: if $|\mathcal{S}_{\mathrm{LE}}| > 1$ then
4:     select two distinct points $x_1^*, x_2^* \in \mathcal{S}$ with $x_n^* \neq 0$ and $x_n^* \neq x_m$ for $n = 1, 2$
       and $m = 1, 2$
5:     call search-LE$(x_1^*, x_2^*, f, S_1, S_2) \to \mathcal{S}_{\mathrm{LE}}^*$
6:     $\mathcal{S}_{\mathrm{LE}} \leftarrow \mathcal{S}_{\mathrm{LE}} \cap \mathcal{S}_{\mathrm{LE}}^*$
7: end if
8: return $\mathcal{S}_{\mathrm{LE}}$
where Procedure search-LE:
Input: $x_1, x_2, f, S_1, S_2$
Output: $\mathcal{S}_{\mathrm{LE}}$
1: $\mathcal{S}_{\mathrm{LE}} \leftarrow \varnothing$
2: for all $a_1 \in \mathbb{F}_2^8 \setminus \{0\}$ do
3:     $A(x_1) \leftarrow a_1 \| f(a_1)$
4:     for all $a_2 \in \mathbb{F}_2^8 \setminus \{0\}$ do
5:         $A(x_2) \leftarrow a_2 \| f(a_2)$
6:         call LE on $S_1$ and $S_2$ with initial guesses $A(x_1), A(x_2) \to \mathcal{S}_{\mathrm{LE}}$
7:     end for
8: end for
```

Choice of Set $\mathcal{S}$

Here, we elaborate on the fact that not all four sets $\mathcal{S}_l^{(i,j)}$ $(l = 0, 1, 2, 3)$ are equally suitable to be used in Algorithm 1. To each of the four sets $\mathcal{S}_l^{(i,j)}$ $(l = 0, 1, 2, 3)$, a pair of `MixColumns` coefficients $(mc_{l,0}^i, mc_{l,1}^i) \in \mathcal{S}_{\mathtt{MC}}$ (see (5.3.2)) of the associated function $f_l^{(i,j)}$ is related. Let us denote this relation by $\mathcal{S}_l^{(i,j)} \leftrightarrow (mc_{l,0}^i, mc_{l,1}^i)$.

$\mathcal{S}_l^{(i,j)} \leftrightarrow$ (`01`, `01`): the associated function $f_l^{(i,j)}$ is the identity function such that the pair of initial guesses becomes $\big(A(x_n) = a_n \| a_n\big)_{n=1,2}$ with $a_n \in \mathbb{F}_2^8 \setminus \{0\}$. When executing LE on $S_1 = (\overline{S}, \overline{S})$ and $S_2 = \overline{\mathtt{dTMC}}_i^{(1,j)}$ for any such pair, we only find at most eight linearly independent inputs and output to A (or B). This can be explained by the fact that linear combinations of $a_n \| a_n$ (or of $\overline{S}(a_n) \| \overline{S}(a_n)$) span at most an 8-dimensional space. In order to continue executing LE, an additional guess for a new point x of A (or B) is required which increases the work factor. Hence we avoid using this set.

$\mathcal{S}_l^{(i,j)} \leftrightarrow \{(\texttt{01},\texttt{02}),(\texttt{02},\texttt{03}),(\texttt{03},\texttt{01})\}$: computer simulations show that all three remaining sets can be used in Algorithm 1 without requiring an additional guess during the execution of LE. However, in the worst case scenario, using the set $\mathcal{S}_l^{(i,j)} \leftrightarrow (\texttt{03},\texttt{01})$ requires that the procedure '`search-LE`' needs to be executed four times in total in order to find the single desired linear equivalence $(A,B)_d$, instead of at most two times for the sets $\mathcal{S}_l^{(i,j)} \leftrightarrow (\texttt{01},\texttt{02})$ or $\mathcal{S}_l^{(i,j)} \leftrightarrow (\texttt{02},\texttt{03})$. This can be explained as follows.

The procedure '`search-LE`' needs to be repeated if the returned set $\mathcal{S}_{\text{LE}}$ contains additional linear equivalences different from $(A,B)_d$, i.e., $(A,B) = \left(A^s \circ \left(L_i^{(1,j)}\right)^{-1}, R^{(1,j)} \circ \texttt{MC}_i \circ B^s\right)$ with $A^s \neq I_{16}$. Given that the chosen points $x_n \in \mathcal{S}_l^{(i,j)}$ $(n=1,2)$ are defined by $\left(L_i^{(1,j)}\right)^{-1}(x_n) = u_n \| f_l^{(i,j)}(u_n)$, then $(A,B)_d$ is found when $a_n = u_n$ $(n=1,2)$ such that the pair of initial guesses becomes $\left(A(x_n) = u_n \| f_l^{(i,j)}(u_n)\right)_{n=1,2}$. In order to find an additional linear equivalence with $A^s \neq I_{16}$, there must exist two distinct values $a_n \in \mathbb{F}_2^8 \setminus \{0\}$ for $n=1,2$ such that the initial guesses become

$$A(x_n) = \left(a_n \parallel f_l^{(i,j)}(a_n)\right) = A^s \cdot \left(u_n \parallel f_l^{(i,j)}(u_n)\right) \ , \tag{5.6}$$

for $n=1,2$. Now, in order for the same additional linear equivalence to appear during two executions of the procedure '`search-LE`', and given that the newly chosen points $x_n^* \in \mathcal{S}_l^{(i,j)}$ $(n=1,2)$ are defined by $\left(L_i^{(1,j)}\right)^{-1}(x_n^*) = u_n^* \| f_l^i(u_n^*)$, there must exist two additional distinct values $a_n^* \in \mathbb{F}_2^8 \setminus \{0\}$ for $n=1,2$ such that the initial guesses become

$$A(x_n^*) = \left(a_n^* \parallel f_l^{(i,j)}(a_n^*)\right) = A^s \cdot \left(u_n^* \parallel f_l^{(i,j)}(u_n^*)\right) \ , \tag{5.7}$$

for $n=1,2$. Hence in total four distinct values $a_n, a_n^* \in \mathbb{F}_2^8 \setminus \{0\}$ for $n=1,2$ must exist that satisfy the requirements (5.6)-(5.7), where in both requirements A^s is the same. We can reduce this problem to the following problem statement:

Problem Statement 1. *Given $x \in \mathbb{F}_2^8 \setminus \{0\}$, does there exist a $y \in \mathbb{F}_2^8 \setminus \{0\}$ with $x \neq y$ such that $\left(x \parallel f_l^{(i,j)}(x)\right) = A^s \cdot \left(y \parallel f_l^{(i,j)}(y)\right)$ where $A^s \neq I_{16}$?*

For the sets $\mathcal{S}_l^{(i,j)} \leftrightarrow \{(\texttt{01},\texttt{02}),(\texttt{02},\texttt{03})\}$ and its associated functions $f_l^{(i,j)}$, there are at most three distinct x values for each possible $A^s \neq I_{16}$ that satisfy the above problem statement. This ensures that it is impossible for an additional linear equivalence with $A^s \neq I_{16}$ to appear during two executions of the procedure '`search-LE`' such that $(A, B)_d$ can always be filtered out by repeating the procedure '`search-LE`' if $|\mathcal{S}_{\mathrm{LE}}| > 1$ after the first execution.

However, for the set $\mathcal{S}_l^{(i,j)} \leftrightarrow (\mathtt{03}, \mathtt{01})$ and its associated function $f_l^{(i,j)}$, there are at most six distinct x values for each possible $A^s \neq I_{16}$ that satisfy the above problem statement. This means that in the worst case scenario, an additional linear equivalence with $A^s \neq I_{16}$ can appear during three executions of the procedure '`search-LE`' such that the procedure needs to be repeated four times in order to filter out $(A, B)_d$. This increases the work factor, hence we discard this set.

5.3.4 Phase 3: Extract the embedded AES key and the external encodings.

After Phases 1-2, the linear input encodings $\left(L_i^{(1,j)}\right)^{-1}$ of all $\mathtt{dTMC}_i^{(1,j)}$ tables ($i = 0, 1$ and $j = 0, 1, 2, 3$) of the first round are retrieved.

Extract the embedded AES key. Given the 16-bit value $x_{i,j}$ defined by $\mathtt{dTMC}_i^{(1,j)}(x_{i,j}) = 0$ (see (5.3)), the attacker can extract both first round key bytes $\hat{k}_{2i}^{(1,j)}$ and $\hat{k}_{2i+1}^{(1,j)}$ associated with each $\mathtt{dTMC}_i^{(1,j)}$ table by computing

$$\hat{k}_{2i}^{(1,j)} \parallel \hat{k}_{2i+1}^{(1,j)} = \left(L_i^{(1,j)}\right)^{-1}(x_{i,j}) \oplus \left(\mathsf{52} \parallel \mathsf{52}\right) \ .$$

By doing so for each $\mathtt{dTMC}_i^{(1,j)}$ ($i = 0, 1$ and $j = 0, 1, 2, 3$) and taking into account the data-flow of the white-box implementation of the first round, the attacker is able to retrieve the first round key $\hat{k}^{(1)}$, which after applying the inverse `ShiftRows` operation to it, results in the actual first round key $k^{(1)}$. Recall from Sect. 2.3.1 that for AES-128, $k^{(1)}$ corresponds to the AES key k.

Extract the external encodings. The external 128-bit linear input encoding $\mathtt{IN}^{-1}$ can be extracted from the 128×128 binary matrix $M^{(1)}$ by computing

$$\begin{aligned}\mathtt{IN}^{-1} = \mathtt{SR}^{-1} \circ \Big(&\left(L_0^{(1,0)}\right)^{-1}, \left(L_1^{(1,0)}\right)^{-1}, \left(L_0^{(1,1)}\right)^{-1}, \left(L_1^{(1,1)}\right)^{-1}, \\ &\left(L_0^{(1,2)}\right)^{-1}, \left(L_1^{(1,2)}\right)^{-1}, \left(L_0^{(1,3)}\right)^{-1}, \left(L_1^{(1,3)}\right)^{-1}\Big) \circ M^{(1)} \ .\end{aligned}$$

The external 128-bit linear output encoding `OUT` can be extracted once both the AES key k and $\mathtt{IN}^{-1}$ have been recovered. If e_i ($1 \leq i \leq 128$) denotes the i-th unit vector in $\mathbb{F}_2^{128}$, then calculate for unit vector e_i the 128-bit value $y_i = \mathrm{WBAES}_k\big(\mathtt{IN}\big(\mathrm{AES}_k^{-1}(e_i)\big)\big)$, where WBAES_k denotes the given white-box AES implementation defined by $\mathrm{WBAES}_k = \mathtt{OUT} \circ \mathrm{AES}_k \circ \mathtt{IN}^{-1}$ and AES_k^{-1} denotes the inverse standard AES implementation, both instantiated with the AES key k:

$$y_i = \underbrace{\mathtt{OUT}(\mathrm{AES}_k(\mathtt{IN}^{-1}}_{\mathrm{WBAES}_k}(\mathtt{IN}(\mathrm{AES}_k^{-1}(e_i))))) = \mathtt{OUT}(e_i) \ .$$

The above has been illustrated in Fig. 5.7. Observe that y_i corresponds to the image of e_i under the external 128-bit linear output encoding OUT. Hence OUT is completely defined by calculating all pairs (e_i, y_i) for $1 \leq i \leq 128$.

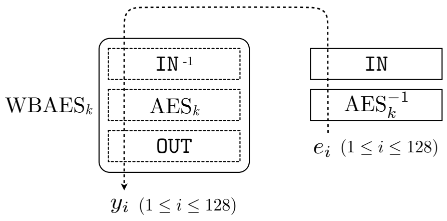


Figure 5.7: Recovery of the external output encoding of the Xiao-Lai white-box AES implementation.

5.3.5 Work Factor

The overall work factor of the cryptanalysis of the Xiao-Lai white-box AES implementation is dominated by the execution of Algorithm 1 in order to obtain the linear input encodings $\big(L_i^{(1,j)}\big)^{-1}$ of all eight $\mathtt{dTMC}_i^{(1,j)}$ tables ($i = 0, 1$ and $j = 0, 1, 2, 3$) of the first round. The algorithm has a work factor of about 2^{29}. Thus, executing the algorithm on $S_1 = (\overline{S}, \overline{S})$ and $S_2 = \overline{\mathtt{dTMC}_i^{(1,j)}}$ for $i = 0, 1$ and $j = 0, 1, 2, 3$ leads to an overall work factor of about $8 \cdot 2^{29} = 2^{32}$.

5.4 The Generic Case

The cryptanalysis presented in Sect. 5.3 was based on Assumption 1, which stated that the order of the bytes of the intermediate results in the white-box implementation is known to the attacker. However, although not specified in [107] by Xiao and Lai, one can use a randomization of the order of the subrounds in an AES round and in the order of the bytes within each subround to add confusion to the implementation. This can be implemented in the same manner as explained in Sect. 3.3.3 for the white-box implementation of Chow et al., i.e., by annihilating each *wide* mixing bijection up to an unknown permutation on the indices of the involved bytes and accounting for the unknown permutations in the data-flow of the implementation. In the case of the Xiao-Lai white-box AES implementation, this involves all mixing bijections as they all operate on at least two bytes simultaneously. As a result, this randomization can be implemented 'for free', i.e., without increasing the size and without decreasing the performance of the white-box implementation.

Assumption 2. *In the generic case, it is assumed that the order of the four subrounds in an AES round as well as the order of the four bytes within each subround are randomized and that this randomization is kept secret.*

Assumption 2 complicates the cryptanalysis presented in Sect. 5.3. Below, we elaborate on these complications and provide a generic version of our cryptanalysis taking into account Assumption 2.

5.4.1 Generic Cryptanalysis

This section elaborates on the impact of Assumption 2 on each phase of the cryptanalysis presented in Sect. 5.3 resulting in a generic cryptanalysis.

Setup Phase and Phase 1

The setup phase is independent of the secret randomization and hence remains the same, i.e., the eight $\mathtt{dTMC}_i^{(1,j)}$ tables ($i = 0, 1$ and $j = 0, 1, 2, 3$) can still be made key-independent based on Lemma 4. With regard to Phase 1, the attacker is still able to construct four sets $\mathcal{S}_l^{(i,j)}$ ($l = 0, 1, 2, 3$) as defined by (5.5) comprising leaked information for each linear input encoding $\left(L_i^{(1,j)}\right)^{-1}$ for $i = 0, 1$ and $j = 0, 1, 2, 3$; however, the associated function $f_l^{(i,j)}$ with each set $\mathcal{S}_l^{(i,j)}$ is no longer known due to the secret randomization. Instead, the associated function can be any element of the known set

$$\mathcal{S}_f = \left\{ f = \overline{S}^{-1} \circ \otimes_{mc_1^{-1}} \circ \otimes_{mc_0} \circ \overline{S} \;\middle|\; (mc_0, mc_1) \in \mathcal{S}^*_{\mathtt{MC}} \right\} \qquad \text{with}$$

$$\mathcal{S}^*_{\mathtt{MC}} = \{(\mathtt{01}, \mathtt{02}), (\mathtt{02}, \mathtt{03}), (\mathtt{03}, \mathtt{01}), (\mathtt{01}, \mathtt{01}), (\mathtt{01}, \mathtt{03}), (\mathtt{03}, \mathtt{02}), (\mathtt{02}, \mathtt{01})\} \ .$$

The set $\mathcal{S}^*_{\mathtt{MC}}$ comprises all possible pairs formed out of the four `MixColumns` coefficients appearing on each row of the 4×4 matrix `MC`, i.e., out of the set

$\{\texttt{01}, \texttt{01}, \texttt{02}, \texttt{03}\}$. Since two `MixColumns` coefficients are equal to `01` for AES encryption, we have that $|\mathcal{S}^*_{\texttt{MC}}| = 7$ and as a result also $|\mathcal{S}_f| = 7$.

Phase 2

The second phase retrieves the secret linear input encodings $\big(L_i^{(1,j)}\big)^{-1}$ for $i = 0, 1$ and $j = 0, 1, 2, 3$. Originally, this was achieved by using Algorithm 1 of Phase 2 (i.e., the algorithm for finding the desired linear equivalence $(A, B)_d$) which required as inputs one of the four sets $\mathcal{S}_l^{(i,j)}$ $(l = 0, 1, 2, 3)$ and its associated function $f_l^{(i,j)}$. However, as mentioned above, $f_l^{(i,j)}$ is unknown in the generic case and thus we need to guess $\tilde{f}_l^{(i,j)} \in \mathcal{S}_f$. Now, the question remains: "can we filter out the incorrect guesses of $\tilde{f}_l^{(i,j)} \neq f_l^{(i,j)}$ and obtain $(A, B)_d$?". This is discussed in the following, where $\mathcal{S}^{(i,j)} = \{\mathcal{S}_0^{(i,j)}, \mathcal{S}_1^{(i,j)}, \mathcal{S}_2^{(i,j)}, \mathcal{S}_3^{(i,j)}\}$.

First, randomly select a set $\mathcal{S} \in \mathcal{S}^{(i,j)}$ without knowing the associated function f. Given that the chosen two distinct points $x_n \in \mathcal{S}$ $(n = 1, 2)$ are defined by $\big(L_i^{(1,j)}\big)^{-1}(x_n) = u_n \| f(u_n)$, Algorithm 1 finds a linear equivalence if there exist two distinct values $a_n \in \mathbb{F}_2^8 \setminus \{0\}$ for $n = 1, 2$ such that the initial guesses for A become

$$A(x_n) = \big(a_n \| \tilde{f}(a_n)\big) = A^s \cdot \big(u_n \| f(u_n)\big) \quad \text{for } n = 1, 2 \ ,$$

for some guess of $\tilde{f} \in \mathcal{S}_f$ and for some A^s (see Property 5). This problem can be reduced to the following problem statement:

Problem Statement 2. *Given $x \in \mathbb{F}_2^8 \setminus \{0\}$, does there exist a $y \in \mathbb{F}_2^8 \setminus \{0\}$ such that*

$$\big(x \| \tilde{f}(x)\big) = A^s \cdot \big(y \| f(y)\big) \ , \tag{5.8}$$

for any A^s (see Property 5) and for any pair of functions $(f, \tilde{f}) \in \mathcal{S}_f \times \mathcal{S}_f$?

Table 5.2 (left entries if applicable) lists the maximum number of x-values for which there exists a y satisfying (5.8) for each possible A^s and for any pair $(f, \tilde{f})$. As a result of a certain symmetry within the set $\mathcal{S}_f$ and A^s, the entries of 255 in Table 5.2 on both 'diagonals' can be explained by the following:

1. If $\tilde{f} = f$ and $A^s = I_{16}$, where I_{16} denotes the 16-bit identity matrix over $\mathbb{F}_2$, then (5.8) becomes $\big(x \| f(x)\big) = \big(y \| f(y)\big)$ such that for each $x \in \mathbb{F}_2^8 \setminus \{0\}$ there exist a y satisfying the equation, i.e., $y = x$. This is considered to be the *trivial case*; if we guess f correctly, then at least the desired linear equivalence $(A, B)_d$ with $A = \big(L_i^{(1,j)}\big)^{-1}$ is given as output.

Table 5.2: For any pair of functions $(f, \tilde{f}) \in \mathcal{S}_f \times \mathcal{S}_f$ listing the maximum number of $x \in \mathbb{F}_2^8 \setminus \{0\}$ for which there exists a $y \in \mathbb{F}_2^8 \setminus \{0\}$ satisfying (5.8) taken over all possible A^s.

$\tilde{f}$ f	$(\texttt{01}, \texttt{02})$	$(\texttt{02}, \texttt{03})$	$(\texttt{03}, \texttt{01})$	$(\texttt{01}, \texttt{03})$	$(\texttt{03}, \texttt{02})$	$(\texttt{02}, \texttt{01})$
$(\texttt{01}, \texttt{02})$	$255 \mid 3$	6	4	4	6	$255 \mid 3$
$(\texttt{02}, \texttt{03})$	4	$255 \mid 3$	4	4	$255 \mid 3$	4
$(\texttt{03}, \texttt{01})$	4	5	$255 \mid 6$	$255 \mid 6$	5	4
$(\texttt{01}, \texttt{01})$	3	4	3	3	4	3
$(\texttt{01}, \texttt{03})$	4	5	$255 \mid 6$	$255 \mid 6$	5	4
$(\texttt{03}, \texttt{02})$	4	$255 \mid 3$	4	4	$255 \mid 3$	4
$(\texttt{02}, \texttt{01})$	$255 \mid 3$	6	4	4	6	$255 \mid 3$

2. If $\tilde{f} = f^{-1}$ and $A^s = \begin{pmatrix} 0_{8\times 8} & I_8 \\ I_8 & 0_{8\times 8} \end{pmatrix}$, where $0_{8\times 8}$ denotes the 8×8 zero matrix and I_8 denotes the 8-bit identity matrix over $\mathbb{F}_2$, then (5.8) becomes $\big(x \| f^{-1}(x)\big) = \big(f(y) \| y\big)$ such that for each $x \in \mathbb{F}_2^8 \setminus \{0\}$ there exist a y satisfying the equation, i.e., $y = f^{-1}(x)$. Hence, if we guess the inverse of f, then at least the linear equivalence (A, B) with $A = A^s \cdot \big(L_i^{(1,j)}\big)^{-1}$ is given as output where A^s is as specified above. Let us denote this specific linear equivalence by $(A, B)'_d$ in the following.

Excluding the above two cases results in the right entries (if applicable) of Table 5.2. This shows that there are at most six distinct x-values for each possible A^s and for any pair $(f, \tilde{f})$ (excluding the above cases) for which there exists a y satisfying (5.8). Observe that the grey-colored entries correspond to the cases discussed in Sect. 5.3.3 to determine the best choice of the set $\mathcal{S}$ selected out of $\mathcal{S}^{(i,j)}$ in order to execute Algorithm 1.

Note that in Table 5.2 the identity function I_8 (i.e., the function with $(mc_0, mc_1) = (\texttt{01}, \texttt{01})$) is left out as a possible guess for $\tilde{f}$. The reason for this omission is that the identity function requires additional guesses during the execution of LE, which is undesirable since it increases the work factor. This was already discussed in Sect. 5.3.3.

Generic algorithm for finding $(A, B)_d$ and $(A, B)'_d$. Here, we present a generic algorithm for finding the linear equivalences $(A, B)_d$ and $(A, B)'_d$ that eventually yield the secret linear input encoding $\big(L_i^{(1,j)}\big)^{-1}$. From Table 5.2 and the above observations it follows that if LE is repeated four times for a certain chosen set $\mathcal{S} \in \mathcal{S}^{(i,j)}$ and for all six guesses of $\tilde{f} \in \mathcal{S}_f \setminus \{I_8\}$ we get either

1. *no solutions* which shows that the chosen set is $\mathcal{S} \leftrightarrow (\texttt{01}, \texttt{01})$. In this case we need to chose a different set $\mathcal{S}^* \in \mathcal{S}^{(i,j)}$ and repeat the whole procedure for this new set;
2. *exactly two solutions*, i.e., $(A, B)_d$ and $(A, B)'_d$, out of which we can easily filter out the linear input encoding $\left(L_i^{(1,j)}\right)^{-1}$ as explained below.

The reason for repeating LE four times is to exclude additional linear equivalences except for $(A, B)_d$ and $(A, B)'_d$. From Table 5.2 it follows that such additional linear equivalences can only occur during at most three executions of LE. Note that it is only required to repeat LE four times if at least one linear equivalence is found during the first execution of LE.

Algorithm 2 gives a detailed description of the whole procedure for finding both linear equivalences $(A, B)_d$ and $(A, B)'_d$ contained within the returned set $\mathcal{S}_{(A,B)}$. Although the attacker cannot distinguish both elements in $\mathcal{S}_{(A,B)}$, he knows that both A's of the found pairs of linear equivalences have the form

$$A_1 = \left(L_i^{(1,j)}\right)^{-1} \quad \text{and} \quad A_2 = C \cdot \left(L_i^{(1,j)}\right)^{-1} \quad \text{with} \quad C = \begin{pmatrix} 0_{8\times 8} & I_8 \\ I_8 & 0_{8\times 8} \end{pmatrix} ,$$

or vice versa. Hence by verifying whether $A_1 \cdot A_2^{-1}$ or $A_2 \cdot A_1^{-1}$ equals C, the attacker is able to retrieve the secret linear input encoding $\left(L_i^{(1,j)}\right)^{-1}$.

Phase 3

After the setup phase and Phases 1-2, the attacker retrieved all encodings $\left(L_i^{(1,j)}\right)^{-1}$ ($i = 0, 1$ and $j = 0, 1, 2, 3$) of the first round. This enables him to extract the round key bytes of the first round as described in Sect. 5.3.4. However, due to the secret randomization, there exists an ambiguity about the order of the round key bytes. Therefore, as was done in the BGE attack, the attacker needs to extract the round key bytes of the second round as well. This can be achieved by repeating the setup phase and Phases 1-2 for the second round. Observe that the generic cryptanalysis presented above can be applied to any two consecutive rounds r and $r + 1$ for some value of r with $1 \leq r \leq 8$ and is not restricted to the first two rounds.

After that, the values of the round key bytes of two consecutive rounds are known, though with an unknown order of the round key bytes associated with each subround and an unknown order of the four subrounds. Phase 4 of the improved BGE attack (Sect. 4.1.3) provides an efficient method to determine the correct order of the round key bytes and to extract the secret AES key.

Algorithm 2 Finding the linear equivalences $(A, B)_d$ and $(A, B)'_d$

```
Input: $S_1 = (\overline{S}, \overline{S})$, $S_2 = \overline{\mathtt{dTMC}}_i^{(1,j)}$, $\mathcal{S}^{(i,j)}$, $\mathcal{S}_f \setminus \{I_8\}$
Output: $(A, B)_d$ and $(A, B)'_d$
 1: choose $\mathcal{S} \in \mathcal{S}^{(i,j)}$
 2: $\mathcal{S}_{(A,B)} \leftarrow \varnothing$
 3: for all $\tilde{f} \in \mathcal{S}_f \setminus \{I_8\}$ do
 4:     select 8 distinct points $x_n^{(i)} \in \mathcal{S}$ with $x_n^{(i)} \neq 0$ for $n = 1, 2$ and $0 \leq i \leq 3$
 5:     call search-LE$\big(x_1^{(0)}, x_2^{(0)}, \tilde{f}, S_1, S_2\big) \rightarrow \mathcal{S}_{\text{LE}}$
 6:     if $|\mathcal{S}_{\text{LE}}| > 0$ then
 7:         for $i = 1$ to 3 do
 8:             call search-LE$\big(x_1^{(i)}, x_2^{(i)}, \tilde{f}, S_1, S_2\big) \rightarrow \mathcal{S}^*_{\text{LE}}$
 9:             $\mathcal{S}_{\text{LE}} \leftarrow \mathcal{S}_{\text{LE}} \cap \mathcal{S}^*_{\text{LE}}$
10:         end for
11:     end if
12:     $\mathcal{S}_{(A,B)} \leftarrow \mathcal{S}_{(A,B)} \cup \mathcal{S}_{\text{LE}}$
13: end for
14: if $|\mathcal{S}_{(A,B)}| = \varnothing$ then
15:     choose $\mathcal{S}^* \in \mathcal{S}^{(i,j)}$ with $\mathcal{S} \neq \mathcal{S}^*$
16:     repeat steps 3–13 with the set $\mathcal{S}^*$
17: end if
18: return $\mathcal{S}_{(A,B)}$
where Procedure search-LE is as specified in Algorithm 1.
```

With regard to the external encodings $\mathtt{IN}^{-1}$ and $\mathtt{OUT}$, both bijective linear mappings on $\mathbb{F}_2^{128}$, it suffices to say that the attacker is in possession of the AES key (such that he can construct a standard AES encryption/decryption routine instantiated with the extracted key) and furthermore can observe a plain intermediate AES result that gives him access to the raw plaintext and ciphertext. This enables the attacker to determine the image of the external encodings for each i-th unit vector e_i in $\mathbb{F}_2^{128}$.

5.4.2 Work Factor

The overall work factor of the generic cryptanalysis is dominated by the execution of Algorithm 2 to obtain the eight secret linear input encodings of two consecutive rounds. As a result, the work factor is upper bounded by $2 \cdot 8 \cdot 2 \cdot 6 \cdot 4 \cdot 2^{28} < 2^{38}$, where the first two factors $2 \cdot 8$ denote the number of secret linear input encodings, the third factor 2 refers to the case when the chosen set $\mathcal{S} \leftrightarrow (\mathtt{01}, \mathtt{01})$, the fourth factor 6 refers to the number of guesses for $\tilde{f}$ and the last two factors $4 \cdot 2^{28}$

indicate the maximum number of executions of LE and the work factor of LE, respectively.

5.5 What about Other Types of Encodings?

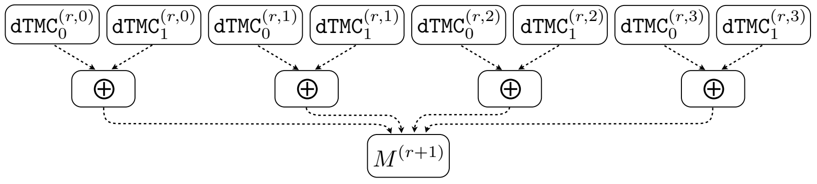
In this section, we discuss the scenario in which the $\mathbb{F}_2$-linear white-box encodings of the Xiao-Lai white-box AES implementation are replaced by either $\mathbb{F}_2$-affine or non-affine encodings. Since the cryptanalysis presented in Sect. 5.3 (of which a generic version is presented in Sect. 5.4) heavily relies on the linearity of the encodings (i.e., the fact that the encodings map zero to itself), the attack is no longer applicable in the case of non-linear encodings (either affine or non-affine).

In the following, we show that even though the Xiao-Lai white-box AES implementation is claimed to be resistant against the BGE attack [13], the generic white-box attack of Michiels et al. [75] can be applied to it, regardless of the type of used encodings. The corresponding work factor is also discussed.

Encoded AES round function. In the original Xiao-Lai white-box AES implementation [107], the attacker has access to the encoded AES round functions for rounds $1 \leq r \leq 9$ by composing the following three operations as depicted in Fig. 5.8a: (i) the eight key-dependent $\mathtt{dTMC}_i^{(r,j)}$ ($i = 0, 1$ and $j = 0, 1, 2, 3$) lookup tables associated with round r, (ii) four 32-bit XOR operations, and (iii) the matrix-vector multiplication over $\mathbb{F}_2$ by the 128×128 non-singular binary matrix $M^{(r+1)}$. Such an encoded AES round function, denoted by $\mathtt{AES}_{enc}^{(r)}$ ($1 \leq r \leq 9$) and depicted in Fig. 5.8b, has the following properties:

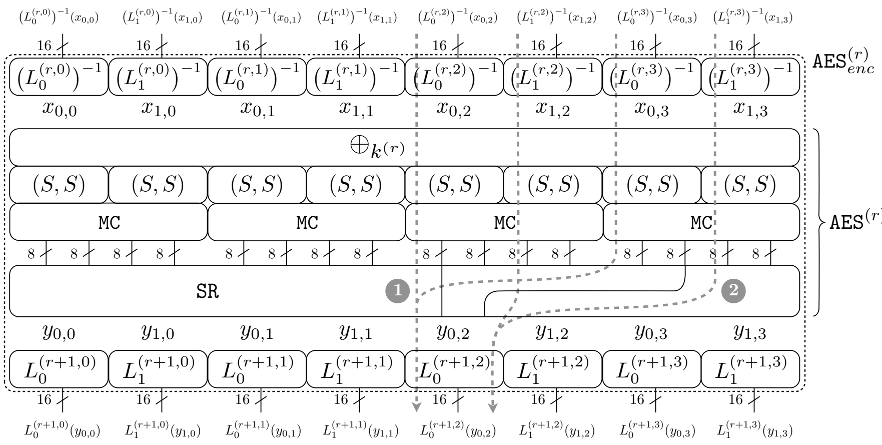
1. the input encodings $\left(L_i^{(r,j)}\right)^{-1}$ and output encodings $L_i^{(r+1,j)}$ operate on two bytes of the AES state simultaneously;
2. the `ShiftRows` operation is explicitly present in the composition of the encoded AES round functions such that each 16-bit encoded output word depends on two output bytes coming from two different AES subrounds (this is illustrated in Fig. 5.8b).

As a result, the bijective mappings listed in Vulnerability 2 exploited by the BGE attack (p. 94) are no longer accessible to the attacker. With this argument, Xiao and Lai [107] claim that their white-box AES implementation is resistant against the BGE attack.



$\mathtt{dTMC}_0^{(r,0)}$
$\mathtt{dTMC}_1^{(r,0)}$
$\mathtt{dTMC}_0^{(r,1)}$
$\mathtt{dTMC}_1^{(r,1)}$
$\mathtt{dTMC}_0^{(r,2)}$
$\mathtt{dTMC}_1^{(r,2)}$
$\mathtt{dTMC}_0^{(r,3)}$
$\mathtt{dTMC}_1^{(r,3)}$
$\oplus$
$M^{(r+1)}$

(a) Composition of operations of the Xiao-Lai white-box AES implementation.



$(L_0^{(r,0)})^{-1}(x_{0,0})$
$(L_1^{(r,0)})^{-1}(x_{1,0})$
$(L_0^{(r,1)})^{-1}(x_{0,1})$
$(L_1^{(r,1)})^{-1}(x_{1,1})$
$(L_0^{(r,2)})^{-1}(x_{0,2})$
$(L_1^{(r,2)})^{-1}(x_{1,2})$
$(L_0^{(r,3)})^{-1}(x_{0,3})$
$(L_1^{(r,3)})^{-1}(x_{1,3})$
16
$\mathtt{AES}_{enc}^{(r)}$
$x_{0,0}$ $x_{1,0}$ $x_{0,1}$ $x_{1,1}$ $x_{0,2}$ $x_{1,2}$ $x_{0,3}$ $x_{1,3}$
$\oplus_{k^{(r)}}$
(S,S)
MC
8
SR
1
2
$\mathtt{AES}^{(r)}$
$y_{0,0}$ $y_{1,0}$ $y_{0,1}$ $y_{1,1}$ $y_{0,2}$ $y_{1,2}$ $y_{0,3}$ $y_{1,3}$
$L_0^{(r+1,0)}$ $L_1^{(r+1,0)}$ $L_0^{(r+1,1)}$ $L_1^{(r+1,1)}$ $L_0^{(r+1,2)}$ $L_1^{(r+1,2)}$ $L_0^{(r+1,3)}$ $L_1^{(r+1,3)}$
$L_0^{(r+1,0)}(y_{0,0})$
$L_1^{(r+1,0)}(y_{1,0})$
$L_0^{(r+1,1)}(y_{0,1})$
$L_1^{(r+1,1)}(y_{1,1})$
$L_0^{(r+1,2)}(y_{0,2})$
$L_1^{(r+1,2)}(y_{1,2})$
$L_0^{(r+1,3)}(y_{0,3})$
$L_1^{(r+1,3)}(y_{1,3})$

(b) Encoded AES round function for rounds $1 \leq r \leq 9$.

Figure 5.8: Composition of operations of the Xiao-Lai white-box AES implementation [107] in order to obtain access to the encoded AES round functions for rounds $1 \leq r \leq 9$.

The encodings $\left(L_i^{(r,j)}\right)^{-1}$ and $L_i^{(r+1,j)}$ ($i = 0, 1$ and $j = 0, 1, 2, 3$) of $\mathtt{AES}_{enc}^{(r)}$ are solely $\mathbb{F}_2$-linear in the original Xiao-Lai white-box AES implementation [107]. However, in the following it is assumed that each encoding can be any random bijective mapping on $\mathbb{F}_2^{16}$. How this assumption affects the composition or construction of the white-box implementation is briefly discussed in the following, where naturally the original Xiao-Lai white-box AES implementation specified in [107] covers the case where the encodings are $\mathbb{F}_2$-linear. In the case that the encodings are $\mathbb{F}_2$-affine, the composition of the white-box implementation stays almost identical with the only difference that the operations $M^{(r)}$ ($1 \leq r \leq 10$) and $M^{(11)}$ are now affine, i.e., they comprise a 128×128 non-singular binary matrix and a 128-bit constant vector. As a result, the Xiao-Lai white-box AES implementation with linear or affine encodings has almost identical overall implementation size and performance. Finally, in the case that the encodings are non-affine, all 32-bit XOR operations and the operations $M^{(r)}$ ($1 \leq r \leq 10$)

and $M^{(11)}$ need to be implemented as a network of lookup tables, which will have a significant impact on both the implementation size and performance. However, as is explained in the following section, the latter case (i.e., non-affine encodings) does not provide a higher level of white-box security when compared with the case of affine encodings and hence can be disregarded.

5.5.1 Michiels et al.'s Generic White-Box Attack

Recall from Sect. 3.5.4 that Michiels et al. [75] developed an algebraic attack on white-box implementations of a generic class of SLT ciphers (Def. 25) if some requirements on the diffusion operator of the SLT cipher (Property 2) and on the white-box implementation (Properties 3 and 4) are satisfied. In the following, we show that the encoded AES round functions $\mathtt{AES}_{enc}^{(r)}$ $(1 \leq r \leq 9)$ of the Xiao-Lai white-box AES implementation (Fig. 5.8b) meet all the necessary requirements of Michiels et al.'s cryptanalysis.

1. *Satisfying Def. 25 and Property 2.* Each AES round function, denoted by $\mathtt{AES}^{(r)}$ $(1 \leq r \leq 9)$, is a bijective function on $(\mathbb{F}_2^{16})^8$ mapping the 128-bit input $(x_{0,0}, x_{1,0}, x_{0,1}, x_{1,1}, x_{0,2}, x_{1,2}, x_{0,3}, x_{1,3})$ onto the 128-bit output $(y_{0,0}, y_{1,0}, y_{0,1}, y_{1,1}, y_{0,2}, y_{1,2}, y_{0,3}, y_{1,3})$, where $x_{i,j}, y_{i,j} \in \mathbb{F}_2^{16}$ for $i = 0, 1$ and $j = 0, 1, 2, 3$. It comprises the following three consecutive operations:

   a) a XOR with a 128-bit round key $k^{(r)}$;

   b) the confusion layer consisting of eight 16-bit bijective S-boxes (S, S) in parallel where (S, S) is defined as two AES S-boxes S in parallel;

   c) the diffusion layer represented by the 128×128 binary matrix $D^{(r)} = \mathtt{SR} \circ (\mathtt{MC}, \mathtt{MC}, \mathtt{MC}, \mathtt{MC})$ where $\mathtt{SR}$ and $\mathtt{MC}$ denote the 128×128 non-singular binary matrix representing `ShiftRows` and the 32×32 non-singular binary matrix representing `MixColumns`, respectively.

   All components are included in the cipher's specification expect for the secret round key $k^{(r)}$. Furthermore, since each $\mathtt{MC}$ is a 4×4 MDS matrix over $\mathbb{F}_{256}$ and $\mathtt{SR}$ represents a byte transposition, the diffusion layer satisfies Property 2 as illustrated in Fig. 5.8b: e.g., given the disjoint sets of input words $U = \{x_{0,2}, x_{0,3}\}$ and $V = \{x_{1,2}, x_{1,3}\}$, then the mappings $U \mapsto y_{0,2}$ and $V \mapsto y_{0,2}$ through $D^{(r)}$ (while fixing the uninvolved input words to a constant) are surjective on $\mathbb{F}_2^{16}$.

2. *Satisfying Properties 3 and 4.* For each input word $x_{i,j}$ and output word $y_{i,j}$ $(i = 0, 1$ and $j = 0, 1, 2, 3)$ of $\mathtt{AES}^{(r)}$ $(1 \leq r \leq 9)$, the attacker has access to their fixed encoded versions $\left(L_i^{(r,j)}\right)^{-1}(x_{i,j})$ and $L_i^{(r+1,j)}(y_{i,j})$

($i = 0, 1$ and $j = 0, 1, 2, 3$) since he has access the encoded AES round functions $\texttt{AES}_{enc}^{(r)}$ ($1 \leq r \leq 9$) as explained above. This satisfies Property 3. As a result, Property 4 is met as well.

This concludes that Michiels et al.'s generic white-box attack can be applied to the Xiao-Lai white-box AES implementation, regardless of the type of the encodings (i.e., either $\mathbb{F}_2$-linear, $\mathbb{F}_2$-affine or non-affine). Although this observation on its own is interesting, it remains to be seen whether this leads to a practical attack since the required work factor can become large, especially when keeping in mind that the attack is generic. In the following an estimation of the overall work factor of Michiels et al.'s attack on the Xiao-Lai white-box AES implementation with non-affine encodings is given; note that Michiels et al. [75] do not provide a clean discussion on the work factor of their generic attack. As assumed in [75, Property 1], Assumption 1 applies such that it suffices to extract a single AES round key $k^{(r)}$ by applying the three phases of Michiels et al.'s attack to any encoded AES round function $\texttt{AES}_{enc}^{(r)}$ for some value of r with $3 \leq r \leq 9$. Note that all encodings of $\texttt{AES}_{enc}^{(r)}$ are affine after Phase 1, hence Phase 1 is redundant in the case of the Xiao-Lai white-box AES implementation with solely linear or affine encodings.

Estimating the work factor of Michiels et al.'s attack. The work factor of the first phase is given by $3 \cdot 8 \cdot \left(8 + 16 \cdot 2^{16} + (3 + 4 \cdot 16) \cdot 2^{16}\right) \approx 2^{27}$, where $3 \cdot 8$ denotes the number of involved output encodings, $8 + 16 \cdot 2^{16}$ refers to the work factor required to identify two bijective mappings on $\mathbb{F}_2^{16}$, and $(3 + 4 \cdot 16) \cdot 2^{16}$ is the work factor of Tolhuizen's method [101] in order to remove the non-affine part of the encodings.

The work factor of the second phase is given by $2 \cdot (8 \cdot 2^{16} + 8 \cdot 128^3) \approx 2^{25}$ where $8 \cdot 2^{16}$ equals the work factor to transform the affine encoded AES round function into eight tables mapping 16 bits to 128 bits and $8 \cdot 128^3$ equals the work factor to transform the obtained tables into a generic SAT cipher (see [75, Def. 4]) round function (using the Gauss-Jordan elimination).

With regard to the third phase, the work factor required to execute the affine equivalence algorithm (AE) is given by $2 \cdot 8 \cdot 2^{44} = 2^{48}$, where $2 \cdot 8$ denotes the number of times that AE needs to be executed, and 2^{44} equals the work factor of AE (using n^3 as the work factor to perform Gaussian elimination on an $n \times n$ binary matrix). The work factor of LEPM is at least $8 \cdot (2 \cdot 2040^2)^2 \approx 2^{49}$, where $2 \cdot 2040^2$ denotes the number of affine self-equivalences of the 16-bit bijective S-box (S, S) given by Property 6 below. However, it remains unclear how many candidates for the round key $k^{(r)}$ are given as output of the algorithm presented in [75, Fig. 1], therefore the only statement that can be made is that the overall work factor of Phase 3 is at least 2^{49}.

As a result, the estimated overall work factor of Michiels et al.'s attack is dominated by the work factor of Phase 3 and thus is at least 2^{49}. As mentioned before, the use of non-affine encodings has a negligible impact on the overall work factor of Michiels et al.'s attack, i.e., the work factor of Phase 1 is significantly less than the work factor of Phase 3. Consequently, the Xiao-Lai white-box AES implementations with affine or non-affine encodings have the same white-box security. But, recall that these results were obtained under Assumption 1. Hence it remains an open question to what extent Assumption 2 has an impact on the overall work factor of Michiels et al.'s attack.

Property 6 (Affine self-equivalences of (S, S))**.** *Let Φ_a denote the set of exactly 2040 affine self-equivalences (α, β) of the AES S-box S as defined by Biryukov et al. [14, Sect. 5.1] such that $S = \beta \circ S \circ \alpha$, then the 16-bit bijective S-box (S, S) comprising two identical AES S-boxes S in parallel has at least $2 \cdot 2040^2$ affine self-equivalences denoted by the pair of 16-bit bijective affine mappings (A^s, B^s) such that $(S, S) = B^s \circ (S, S) \circ A^s$, with the following diagonal structure:*

$$A^s = \begin{pmatrix} \alpha_1 & 0_{8\times 8} \\ 0_{8\times 8} & \alpha_2 \end{pmatrix}, \; B^s = \begin{pmatrix} \beta_1 & 0_{8\times 8} \\ 0_{8\times 8} & \beta_2 \end{pmatrix} \; \text{or}$$

$$A^s = \begin{pmatrix} 0_{8\times 8} & \alpha_1 \\ \alpha_2 & 0_{8\times 8} \end{pmatrix}, \; B^s = \begin{pmatrix} 0_{8\times 8} & \beta_2 \\ \beta_1 & 0_{8\times 8} \end{pmatrix},$$

where both $(\alpha_1, \beta_1), (\alpha_2, \beta_2) \in \Phi_a$ and where $0_{8\times 8}$ denotes the 8×8 zero matrix.

5.6 Conclusion

We have presented in detail a practical attack on the white-box AES implementation of Xiao and Lai [107]. The cryptanalysis exploits both specific properties of AES as well as the composition of the white-box AES implementation (such as the linearity of the secret white-box encodings). It uses a modified variant of the linear equivalence algorithm presented by Biryukov et al. [14], which is built by exploiting information leaked from the white-box implementation. The attack efficiently extracts the AES key from the Xiao-Lai white-box AES implementation with a work factor of about 2^{32}. In addition to extracting the AES key, which is the main goal in the cryptanalysis of white-box implementations, our cryptanalysis is also able to recover the external input and output encodings. As a result, the Xiao-Lai white-box AES implementation is proven to be WBKR-insecure. Crucial parts of the cryptanalysis have been implemented in `C++` and verified by computer experiments. The implementation furthermore shows that both the 128-bit AES key as well as the external encodings can be extracted from the white-box implementation in just a few minutes on a modern PC.

We have also presented an attack on a generic version of the original Xiao-Lai white-box AES implementation; in the generic case, it is assumed that the order of the bytes of intermediate AES results is randomized in the white-box implementation. This randomization, which can be implemented 'for free', causes the work factor to increase to 2^{38} but nevertheless it remains practical.

Furthermore, we have shown that the Xiao-Lai white-box AES implementation with any type of encodings (either linear, affine or non-affine) satisfies all necessary requirements in order to apply Michiels et al.'s attack [75]. However, due to its generic nature, Michiels et al.'s attack is not optimized (i.e., it does not exploit specific properties of AES or the white-box implementation itself) and has a rather large estimated work factor of at least 2^{49}. Since the use of non-affine encodings has a negligible impact on the overall work factor (due to Tolhuizen's improvement), the work factor is the same for the Xiao-Lai implementation with either linear, affine or non-affine encodings. However, for the linear case (which corresponds with the original white-box AES implementation specified by Xiao and Lai in [107]), we presented an optimized practical attack with a work factor of 2^{32}. Now, with regard to the Xiao-Lai white-box AES implementation with affine encodings, there are still two open questions: (i) To what extent can Michiels et al.'s attack be optimized by exploiting specific properties of AES and the white-box implementation?, and (ii) What is the impact on the overall work factor of Michiels et al.'s attack when the bytes of intermediate AES results are randomized in the white-box implementation?

Chapter 6

Cryptanalysis of Bringer et al.'s Perturbated White-Box AES Implementation

In response to the BGE attack [13] and Michiels et al.'s attack [75] on the white-box AES implementation of Chow et al. [23], Bringer, Chabanne and Dottax [20] proposed in 2006 a novel white-box strategy that strongly differs from the typical lookup-table-based white-box technique of Chow et al. The novel technique comprises the injection of well-controlled faults (so-called *perturbations*) in the round function computations that are only canceled out in the final round with a high probability. Each round of the obtained perturbated white-box implementation is represented by a system of multivariate polynomial equations over a finite field instead of a network of encoded lookup tables. In [20], Bringer et al. apply their novel technique to a variant of AES-128 with non-standard key-dependent S-boxes, referred to as the *Advanced Encryption Standard without standard S-boxes* and denoted by AES* (observe that the notation AEw/oS is used in [20]). The resulting implementation is referred to as the perturbated white-box AES* implementation.

In this chapter, we present an efficient cryptanalysis of the perturbated white-box AES* implementation. This research was published in [38]. We show how a set of *equivalent keys* can be extracted from the perturbated white-box AES* implementation; each equivalent key yields an invertible implementation that is functionally equivalent to the original AES*. The presented cryptanalysis extends naturally to perturbated white-box AES implementations.

6.1 Bringer et al.'s Novel White-Box Technique

In 2003, Billet and Gilbert [12] proposed a traceable block cipher by constructing many user-specific functionally equivalent implementations of the same instance of an 'iterated block cipher' based on the commutativity of the underlying building blocks. The security of the building blocks of the proposed traceable cipher relies on the practical difficulty of the linear variant of the Isomorphisms of Polynomials (IP) problem with two secrets (cf. Patarin [84]). However, in 2006, Faugère and Perret [42] presented an efficient equivalence algorithm for solving IP; they solved the challenge proposed by Billet and Gilbert as the basic building block of their traceable cipher in less than a second [42]. Based on the idea proposed by Ding [40] to introduce *perturbations* to reinforce an IP-based cryptosystem, Bringer, Chabanne and Dottax [19] reinforced the traceable block cipher of Billet and Gilbert. Based on a similar approach, Bringer, Chabanne and Dottax [20] proposed a *perturbated* white-box AES implementation. In the following, the novel white-box technique based on perturbations by Bringer et al. is described when applied to iterated block ciphers.

As mentioned in the introductory part of this chapter, the white-box techniques of Binger et al. [20] and of Chow et al. [24, 23], both comprising two phases, differ significantly. Their differences are briefly highlighted below. For a detailed description of the generic white-box techniques of Chow et al., refer to Sect. 3.2. In the following, the novel technique of Bringer et al. is described.

	Bringer et al.'s technique	Chow et al.'s technique
Phase 1	Represent each round function of the block cipher as a system of multivariate polynomial equations over a finite field.	Represent the block cipher as a series of lookup tables by merging several steps of the round functions.
Phase 2	Extend each round function with a system of perturbation and random polynomial equations and apply annihilating linear encodings between successive rounds to mix all systems together.	Apply secret invertible white-box encodings to the input and output of all lookup tables in a pairwise annihilating manner between successive tables.

In a nutshell, the second phase of Bringer et al.'s novel white-box technique comprises the following three steps:

Step 1: Extend each round function with a system of *perturbation* equations as follows. Perturbations are introduced in the first round, which are carried through all intermediate rounds and are canceled out in the final round with a high probability. In other words, a well-controlled fault is injected in the first round that will only be corrected for in the final round. In order to guarantee the overall functionality of the block cipher, it is necessary to implement four instances in parallel with correlated perturbation functions such that a *majority vote* can distinguish the correct output.

Step 2: Further extend each round function (except for the final round) with a system of *random* equations in order to mask all internal round operations. After Steps 1-2, *perturbated round functions* are obtained.

Step 3: Apply annihilating linear encodings between successive perturbated round functions and represent each encoded perturbated round function as a system of multivariate polynomial equations over a finite field.

The idea behind the introduction of perturbations is to hide the algebraic structure of the round functions of the block cipher from the attacker in order to preclude the direct application of algebraic attacks. Furthermore, carrying the introduced perturbations through all rounds ensures that all rounds are strongly interleaved with one another and that all intermediate values between successive rounds are 'false' (i.e., different from their original value).

Bringer et al.'s perturbation white-box technique. Provided with the general description given above, a more in-depth description of Bringer et al.'s method is given below. For details, refer to Bringer et al. [20].

First some notation is introduced. Let $\mathbb{L}$ denote the finite field $\mathbb{F}_{2^q}$ with $q \in \mathbb{N}^*$. Let E be an R-round iterated block cipher mapping a plaintext $P = (p_0, p_1, \ldots, p_{n-1})$ onto a ciphertext $C = (c_0, c_1, \ldots, c_{n-1})$ with $p_i, c_i \in \mathbb{L}$ $(0 \le i \le n-1)$ such that $C = E(P)$. Further, let $E^{(r)}$ denote the r-th round of E and let $Y^{(r)} = (y_0^{(r)}, y_1^{(r)}, \ldots, y_{n-1}^{(r)})$ with $y_i^{(r)} \in \mathbb{L}$ $(0 \le i \le n-1)$ denote the output of $E^{(r)}$ for $1 \le r \le R$ such that

$$\begin{array}{rcll} Y^{(1)} & = & E^{(1)}(P) & \text{for the first round} \ , \\ Y^{(r)} & = & E^{(r)}(Y^{(r-1)}) & \text{for rounds } 2 \le r \le R \text{ with } Y^{(R)} = C \ . \end{array}$$

Phase 1: Represent each round function of E as a system of n multivariate polynomial equations in n variables over $\mathbb{L}$. For the sake of clarity, the system of equations representing the r-th round is also denoted by $E^{(r)}$.

Phase 2: Extend each $E^{(r)}$ with a system of perturbation and random equations and encode the resulting *perturbated round function*:

Step 2.1: a perturbation initialization system Φ is included in the first round and comprises s polynomial equations in n variables over $\mathbb{L}$. The system Φ takes as input the plaintext P and outputs either a predefined value $(\varphi_1, \varphi_2, \ldots, \varphi_s)$ with $\varphi_i \in \mathbb{L}$ $(1 \leq i \leq s)$ or a random value otherwise. In particular, Φ is defined as

$$\Phi(P) = \left(\tilde{0}(P) \oplus \varphi_1, \tilde{0}(P) \oplus \varphi_2, \ldots, \tilde{0}(P) \oplus \varphi_s\right) \ ,$$

where $\tilde{0}(\cdot)$ denotes a polynomial in n variables over $\mathbb{L}$ that 'often' vanishes, i.e., the $\tilde{0}$-polynomial outputs zero for a well chosen subset of the plaintext space $\mathbb{P} = \mathbb{F}_2^{q \cdot n}$ and a random value for the complement of the chosen subset. This is discussed later in more detail (see 'Majority vote' on p. 159). For the construction of $\tilde{0}$-polynomials, refer to Bringer et al. [19, 20].

The value of $\Phi(P)$ is carried through all intermediate rounds $1 < r < R$ such that all intermediate values are perturbated and all rounds are closely linked. Finally, a perturbation cancellation system O_Φ is included in the final round and comprises n polynomial equations in s variables over $\mathbb{L}$. The system O_Φ takes as input the value of $\Phi(P)$ and either vanishes if $\Phi(P) = (\varphi_1, \varphi_2, \ldots, \varphi_s)$, i.e.,

$$O_\Phi(\varphi_1, \varphi_2, \ldots, \varphi_s) = 0 \ ,$$

or outputs a random value if $\Phi(P) \neq (\varphi_1, \varphi_2, \ldots, \varphi_s)$. The value of $O_\Phi\big(\Phi(P)\big)$ is XOR-ed to the ciphertext C.

Step 2.2: a random system $\texttt{Ran}^{(r)}$ is included in all rounds except for the final round and comprises t polynomial equations in either n variables (for the first round) or $n + s + t$ variables (for rounds $2 \leq r < R$) over $\mathbb{L}$. The system $\texttt{Ran}^{(r)}$ takes as input the plaintext P (for the first round) or the output of the preceding encoded perturbated round (for rounds $2 \leq r < R$ – see later) and outputs a random value for any given input. Such random systems mask all internal operations such as the original round functions and the added perturbations.

After Steps 2.1-2.2, the so-called *perturbated round functions* are obtained, denoted by $\widetilde{E}^{(r)}$, consisting of the parallel composition of the original round functions $E^{(r)}$ together with the perturbation and random systems. As a result, each perturbated round function maps $n + s + t$ input words in $\mathbb{L}$ onto $n + s + t$ output words in $\mathbb{L}$, except for the input of the first round and the output of the final round which remain the original plaintext and ciphertext, respectively.

Step 2.3: secret bijective linear input and output encodings (denoted by $\big(M^{(r-1)}\big)^{-1}$ and $M^{(r)}$, respectively) are applied to each perturbated round function $\widetilde{E}^{(r)}$ $(1 \leq r \leq R)$ with the exception of the input of the first round and the output of the final round that remain unencoded. In other words, no external encodings are applied. Each invertible linear encoding $M^{(r)}$ $(1 \leq r < R)$ is represented by a non-singular $N \times N$ matrix over $\mathbb{L}$ with $N = n + s + t$.

After Step 2.3, the so-called *encoded perturbated round functions* are obtained, denoted by $\overline{E}^{(r)}$ $(1 \leq r \leq R)$ and defined as

$$\begin{array}{lcll} \overline{E}^{(1)} & = & M^{(1)} \circ \big(E^{(1)} \ , \ \Phi \ , \ \mathtt{Ran}^{(1)}\big) & \text{for the first round ,} \\ \overline{E}^{(r)} & = & M^{(r)} \circ \big(E^{(r)} \ , \ I_s \ , \ \mathtt{Ran}^{(r)}\big) \circ \big(M^{(r-1)}\big)^{-1} & \text{for rounds } 1 < r < R \text{ ,} \\ \overline{E}^{(R)} & = & \bigoplus \circ \big(E^{(R)} \ , \ O_\Phi\big) \circ \big(M^{(R-1)}\big)^{-1} & \text{for the final round ,} \end{array}$$

where I_s denotes the $s \times s$ identity matrix over $\mathbb{L}$. Bringer et al. [20] claim that the presence of the secret linear encodings and the extra perturbation and random systems make it harder to recover any secret information from the original round functions $E^{(r)}$. The input and output of $\overline{E}^{(r)}$ are denoted by $Z^{(r-1)}$ and $Z^{(r)}$, respectively. Observe that $Z^{(0)} = P$ and $Z^{(R)} = C \oplus O_\Phi\big(\Phi(P)\big)$.

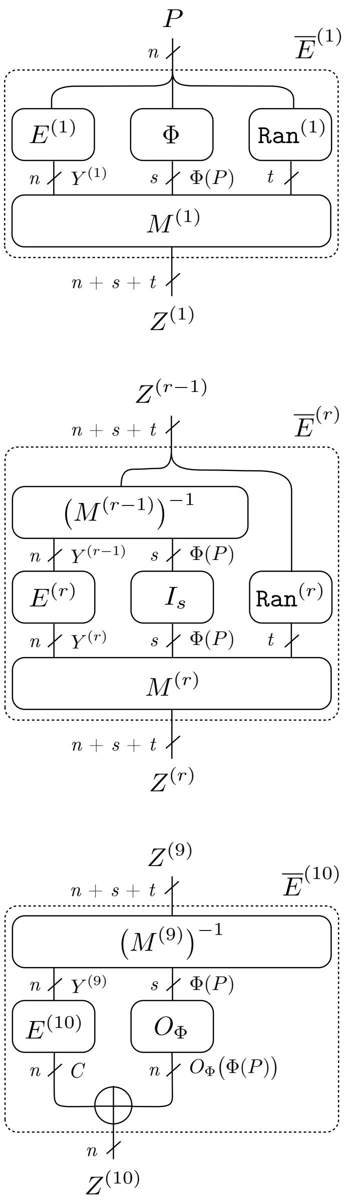
Figure 6.1a depicts an overview of Bringer et al.'s novel white-box technique based on perturbations. Each encoded perturbated round function $\overline{E}^{(r)}$ $(1 \leq r \leq R)$ is represented by a system of N (or n if $r = R$) multivariate polynomial equations in N (or n if $r = 1$) variables over $\mathbb{L}$, where $N = n + s + t$. This leads to a polynomial-based perturbated white-box implementation.

Majority vote: obtaining the correct result. Due to the introduction of the perturbation in the first round through the system Φ, there is a probability that the output of the final round $Z^{(R)}$ is incorrect, i.e., $Z^{(R)} \neq C$. Recall that $Z^{(R)} = C \oplus O_\Phi\big(\Phi(P)\big)$, thus if $\Phi(P) \neq (\varphi_1, \varphi_2, \ldots, \varphi_s)$ then $O_\Phi\big(\Phi(P)\big) \neq 0$. In order to obtain the correct ciphertext for any given plaintext, four correlated instances of the perturbated white-box implementation in parallel are generated. Each instance contains a correlated perturbation initialization system Φ, constructed as follows. Subdivide the plaintext space $\mathbb{P}$ twice into two sets of similar size, i.e.,

$$\mathbb{P} = \mathbb{P}_1 \cup \overline{\mathbb{P}}_1 = \mathbb{P}_2 \cup \overline{\mathbb{P}}_2 \ ,$$

and construct four correlated $\tilde{0}$-polynomials $(\tilde{0}_1, \overline{0}_1, \tilde{0}_2, \overline{0}_2)$ such that

$$\forall P \in \mathbb{P}_k : \tilde{0}_k(P) = 0 \quad \text{and} \quad \forall P \in \overline{\mathbb{P}}_k : \overline{0}_k(P) = 0 \qquad \text{for } k = 1, 2 \ .$$



(a) Overview of Bringer et al.'s novel white-box technique based on perturbations when applied to an iterated block cipher.

$E^{(1)}$

```
(a)  AddRoundKey $k^{(1)}$
(b)  SubBytes $\{S_0^{(1)}, S_1^{(1)}, \dots, S_{15}^{(1)}\}$
(c)  ShiftRows
(d)  MixColumns
```

$E^{(r)}$

```
for $r$ from 2 to 9
(a)  AddRoundKey $k^{(r)}$
(b)  SubBytes $\{S_0^{(r)}, S_1^{(r)}, \dots, S_{15}^{(r)}\}$
(c)  ShiftRows
(d)  MixColumns
```

$E^{(10)}$

```
(a)  AddRoundKey $k^{(10)}$
(b)  SubBytes $\{S_0^{(10)}, S_1^{(10)}, \dots, S_{15}^{(10)}\}$
(c)  ShiftRows
(d)  AddRoundKey $k^{(11)}$
```

(b) Original AES* round functions.

Figure 6.1: Perturbated white-box AES* implementation: $(n, s, t) = (16, 4, 23)$.

Based on these four correlated $\tilde{0}$-polynomials, construct four correlated perturbation initialization systems:

$$\begin{array}{rcl} \Phi_1(P) & = & \big(\tilde{0}_1(P) \oplus \varphi_1, \tilde{0}_1(P) \oplus \varphi_2, \ldots, \tilde{0}_1(P) \oplus \varphi_s\big) \\ \overline{\Phi}_1(P) & = & \big(\overline{0}_1(P) \oplus \varphi_1, \overline{0}_1(P) \oplus \varphi_2, \ldots, \overline{0}_1(P) \oplus \varphi_s\big) \\ \Phi_2(P) & = & \big(\tilde{0}_2(P) \oplus \varphi_1, \tilde{0}_2(P) \oplus \varphi_2, \ldots, \tilde{0}_2(P) \oplus \varphi_s\big) \\ \overline{\Phi}_2(P) & = & \big(\overline{0}_2(P) \oplus \varphi_1, \overline{0}_2(P) \oplus \varphi_2, \ldots, \overline{0}_2(P) \oplus \varphi_s\big) \ . \end{array}$$

For any given plaintext $P \in \mathbb{P}$, exactly two of the systems above output the predefined value $(\varphi_1, \varphi_2, \ldots, \varphi_s)$ while the other two output two different random values with an overwhelming probability. As a result, exactly two instances of the perturbated white-box implementation output the correct ciphertext C while the other two instances output two different random values. A *majority vote* can then be used to distinguish the correct result.

6.2 Perturbated White-Box AES* Implementation

In [20], Bringer et al. apply their novel white-box technique to a variant of AES-128 in which all S-boxes are non-standard (i.e., different from the standard AES S-box) and key-dependent. Recall that this variant is called the Advanced Encryption Standard without standard S-boxes and is referred to as AES*. In the following, it is assumed that the AES* state is represented by the 16-byte vector $[\text{STATE}_l]_{0 \le l \le 15}$ instead of the conventional 4×4 byte array $[\text{state}_{i,j}]_{0 \le i,j \le 3}$. The relation between both representations is given by

$$\text{STATE}_{4j+i} = \text{state}_{i,j} \qquad \text{for } 0 \le i,j \le 3 \ .$$

The AES* Block Cipher

AES* is identical to AES-128 (Sect. 2.3.1) except for the `SubBytes` operation. While AES-128 uses a single publicly known 8-bit bijective non-linaer S-box (i.e., the AES S-box S) and applies it to each byte of the AES state for all 10 rounds, AES* uses 160 different 8-bit bijective non-linear S-boxes $S_i^{(r)}$ ($0 \le i \le 15$ and $1 \le r \le 10$). Furthermore, all 160 S-boxes of AES* are non-standard (i.e., $S_i^{(r)} \neq S$ for $0 \le i \le 15$ and $1 \le r \le 10$) and key-dependent. As a result, the secret key of AES* comprises the 11 128-bit AES round keys (considering the expanded key) as well as the 160 key-dependent S-boxes. The size of the secret AES* key equals $11 \cdot 128 + 160 \cdot 2^8 \cdot 8 = 329\,088$ bits under the assumption that each S-box is represented by a lookup table mapping 8 bits to 8 bits.

The description of AES* used to obtain a perturbated white-box implementation is depicted in Fig. 6.2. Compared with the lookup-table suitable description of AES-128 (Fig. 2.1b), it is no longer required to bundle the `AddRoundKey`, `SubBytes` and `MixColumns` steps together in order to simplify their merging into lookup tables, since each AES* round will now be represented by a system of multivariate polynomials as is explained below.

```
state ← plaintext
for $r = 1$ to 9 do
    state ← AddRoundKey(state,$k^{(r)}$)
    state ← SubBytes(state,$\{S_0^{(r)}, S_1^{(r)}, \ldots, S_{15}^{(r)}\}$)
    state ← ShiftRows(state)
    state ← MixColumns(state)
end for
state ← AddRoundKey(state,$k^{(10)}$)
state ← SubBytes(state,$\{S_0^{(10)}, S_1^{(10)}, \ldots, S_{15}^{(10)}\}$)
state ← ShiftRows(state)
state ← AddRoundKey(state,$k^{11}$)
ciphertext ← state
```

Figure 6.2: Description of AES*.

Applying Bringer et al.'s White-Box Technique to AES*

First, Phase 1 represents each AES* round function by a system $E^{(r)}$ of 16 multivariate polynomial equations over $\mathbb{F}_{256}$; each system $E^{(r)}$ $(1 \leq r \leq 10)$ is defined over 16 variables in $\mathbb{F}_{256}$. It is assumed that the finite field $\mathbb{F}_{256}$ is as defined in FIPS 197 [69]. The choice of $\mathbb{F}_{256}$ seems natural since AES (and hence AES* as well) was designed with the field $\mathbb{F}_{256}$ in mind.

Second, Phase 2 constructs the encoded perturbated round functions $\overline{E}^{(r)}$ $(1 \leq r \leq 10)$. For this purpose, $s = 4$ and $t = 23$ such that $N = n + s + t = 16 + 4 + 23 = 43$. As a result, each mapping $\overline{E}^{(r)}$ $(1 \leq r < 10)$ is represented by a system of 43 multivariate polynomials over $\mathbb{F}_{256}$; the final round $\overline{E}^{(10)}$ by a system of 16 multivariate polynomials over $\mathbb{F}_{256}$. Each system is defined over 43 variables, except for the first round that is defined over the 16 bytes of the plaintext P. These extra equations and variables originate from the extension of the AES* round functions with a perturbation system Φ of four polynomials over $\mathbb{F}_{256}$ and a random system $\texttt{Ran}^{(r)}$ of 23 polynomials over $\mathbb{F}_{256}$.

As mentioned before in Step 2.3 of Phase 2, secret pairwise annihilating linear input and output encodings $M^{(r)}$ are applied between successive perturbated round functions to ensure that all systems are interleaved to make analysis hard: e.g., to prevent an attacker from distnguishing the perturbation/random system from the original round function. The encodings $M^{(r)}$ $(1 \leq r \leq 9)$ are represented by non-singular 43×43 diagonal block matrices over $\mathbb{F}_{256}$ and are constructed as

$$M^{(r)} = \pi^{(r)} \circ \begin{pmatrix} A_1^{(r)} & & & & \\ & A_2^{(r)} & & & \\ & & \ddots & & \\ & & & A_7^{(r)} & \\ & & & & B^{(r)} \end{pmatrix} \circ \sigma^{(r)} \qquad \text{for } 1 \leq r \leq 9 \ ,$$

where

- $A_i^{(r)}$ $(1 \leq i \leq 7)$ are random non-singular 5×5 matrices over $\mathbb{F}_{256}$ of which the inverses have exactly two non-zero coefficients in $\mathbb{F}_{256}$ on each row;
- $B^{(r)}$ is a random non-singular 8×8 matrices over $\mathbb{F}_{256}$ of which the inverse has at least 7 non-zero coefficients in $\mathbb{F}_{256}$ on each row;
- $\pi^{(r)} : (\mathbb{F}_{256})^{43} \to (\mathbb{F}_{256})^{43}$ is a random permutation on the order of the 43 output bytes of $\left(A_1^{(r)}, A_2^{(r)}, \ldots, A_7^{(r)}, B^{(r)}\right)$;
- $\sigma^{(r)} : (\mathbb{F}_{256})^{43} \to (\mathbb{F}_{256})^{43}$ is a random permutation on the order of the 43 input bytes of $M^{(r)}$ defined such that the $A_i^{(r)}$ $(1 \leq i \leq 7)$ matrices mix the 16 polynomials of the original round system $E^{(r)}$ with 19 polynomials of the random system $\texttt{Ran}^{(r)}$, whereas $B^{(r)}$ mixes the four polynomials of the perturbation system Φ (or I_4) with the remaining four polynomials of the random system $\texttt{Ran}^{(r)}$.

For the determination of the constraints on the matrices and permutations, refer to Bringer et al. [20]. The cryptanalysis presented in Sect. 6.3 exploits these characteristics of the linear encodings $M^{(r)}$ $(1 \leq r \leq 9)$.

Summary. The perturbated white-box AES* implementation consists of four instances in parallel, each based on a correlated perturbation initialization system such that a majority vote can reveal the correct result. Each instance consists of 10 encoded perturbated round functions $\overline{E}^{(r)}$ $(1 \leq r \leq 10)$. Figure 6.1 depicts an overview of one instance of the white-box AES* implementation.

Assuming standard AES instead of AES*, i.e., assuming that all 160 secret S-boxes of AES* are in fact the known AES S-box, the implementation size of each instance equals ≈ 142 MB such that the overall implementation size of the perturbated white-box AES implementation is given by ≈ 568 MB (cf. Bringer et al. [20]). Note that the implementation size with respect to AES* depends on the polynomial representation of the chosen S-boxes, thus different S-boxes may result in an increase or decrease of the overall implementation size.

6.3 Cryptanalysis

In this section, we present a cryptanalysis of the perturbated white-box AES* implementation. The goal of the attack is to extract the secret AES* key comprising the 11 128-bit AES round keys and the 160 key-dependent 8-bit bijective non-linear S-boxes. Instead of retrieving the actual secret key, the attack extracts a set of *equivalent keys*; each equivalent key comprises 160 8-bit bijective S-boxes that allows the attacker to contruct a simpler (in terms of storage requirement) and invertible implementation that is functionally equivalent to the original AES* block cipher.

Since the cryptanalysis exploits the characteristics of the secret linear encodings between successive rounds, we start by elaborating on these encodings.

Inverse linear encodings. With regard to the inverse linear encodings $\left(M^{(r-1)}\right)^{-1}$ $(2 \leq r \leq 10)$, the following two observations can be made:

1. The 35-byte output of the $\left(A_i^{(r-1)}\right)^{-1}$ $(1 \leq i \leq 7)$ matrices contains the 16-byte input $Y^{(r-1)}$ of the round function $E^{(r)}$. Further, recall that each row of the $\left(A_i^{(r-1)}\right)^{-1}$ $(1 \leq i \leq 7)$ matrices contains exactly two non-zero coefficients in $\mathbb{F}_{256}$. As a result, each input byte $y_i^{(r-1)}$ $(i = 0, 1, \ldots, 15)$ of $E^{(r)}$ is a linear combination in $\mathbb{F}_{256}$ of exactly two bytes of the 43-byte input $Z^{(r-1)}$ of the encoded perturbated round function $\overline{E}^{(r)}$.

2. The 8-byte output of $\left(B^{(r-1)}\right)^{-1}$ contains the 4-byte perturbation value $\Phi(P)$ (initialized in the first round) that is carried through all intermediate rounds. Further, recall that each row of $\left(B^{(r-1)}\right)^{-1}$ contains at least 7 non-zero coefficients in $\mathbb{F}_{256}$. As a result, $\Phi(P)$ depends on at least 7 (and most likely 8) bytes of $Z^{(r-1)}$.

The first 16 rows of $\left(M^{(r-1)}\right)^{-1}$, denoted by $\left(M^{(r-1)}\right)^{-1}_{[0-15]}$, generate the 16-byte input $Y^{(r-1)}$ of the original round function $E^{(r)}$, i.e.,

$$Y^{(r-1)} = \left(M^{(r-1)}\right)^{-1}_{[0-15]}\left(Z^{(r-1)}\right) \ ,$$

and thus each row of $\left(M^{(r-1)}\right)^{-1}_{[0-15]}$ contains exactly two non-zero coefficients in $\mathbb{F}_{256}$ according to the first observation.

Brief overview of the cryptanalysis. During the attack, it is shown how the secret linear encodings can be eliminated from the encoded perturbated round functions and how the extra systems (i.e., the perturbation and random systems) can be distinguished from the implementation. The cryptanalysis comprises the following four phases preceded by an initial setup phase:

Phase 1 distinguishes the perturbation cancellation system O_Φ from the final AES* round function $E^{(10)}$ and recovers the secret linear input encoding $\left(M^{(9)}\right)^{-1}$ up to an unknown constant diagonal mapping such that the linear equivalent input of the final round $E^{(10)}$ can be observed.

Phase 2 eliminates the `MixColumns` operation from the penultimate AES* round function $E^{(9)}$ such that the unknown coefficients of the diagonal mapping of the linear equivalent output of $E^{(9)}$ can be included in the 16 secret S-boxes $S_i^{(9)}$ $(0 \leq i \leq 15)$.

Phase 3 structurally decomposes all remaining rounds; this includes the retrieval of all secret linear encodings up to an unknown constant diagonal mapping and the elimination of the `MixColumns` operation within all rounds.

Phase 4 finally extracts 160 linear equivalent key-dependent S-boxes that yield a functionally equivalent implementation.

Observe that not all information of the secret 160 S-boxes and linear encodings can be extracted since there are many equivalent keys that yield the same white-box implementation. Indeed, the input/output of an S-box can be multiplied by a fixed constant which is compensated for in the adjacent linear encoding. Therefore, at best an equivalent key can be extracted.

6.3.1 Setup Phase

Encrypt a randomly chosen plaintext P with the four correlated instances of the perturbated white-box AES* implementations and select one of both instances that result into the correct ciphertext (using the majority vote). For that instance, store the 16-byte plaintext P, the 16-byte ciphertext C and all 43-byte intermediate states $Z^{(r)}$ $(1 \leq r \leq 9)$ which are clearly accessible in the white-box attack context.

6.3.2 Phase 1: Analyze the final round

The first phase is applied to the encoded perturbated final round function $\overline{E}^{(10)}$ and extracts the first 16 rows of the secret linear input encoding $\big(M^{(9)}\big)^{-1}$ up to an unknown 16×16 diagonal matrix $\Lambda^{(9)}$ over $\mathbb{F}_{256}$. This enables the attacker to observe the linear equivalent input $\Lambda^{(9)}\big(Y^{(9)}\big)$ of the original final round $E^{(10)}$. Each step of Phase 1 is described below in detail.

Step 1.1: Find the pair of bytes in $Z^{(9)}$ associated with each byte of $Y^{(9)}$

As mentioned above, due to the specific characteristics of the secret linear input encoding $\big(M^{(9)}\big)^{-1}$, each input byte $y_i^{(9)}$ $(0 \leq i \leq 15)$ of the original final round $E^{(10)}$ is a linear combination in $\mathbb{F}_{256}$ of exactly two bytes of the 43-byte input $Z^{(9)}$ of the encoded perturbated final round $\overline{E}^{(10)}$. In the following, the pair of bytes of $Z^{(9)}$ associated with $y_i^{(9)}$ is denoted by $\big(z_{i_1}^{(9)}, z_{i_2}^{(9)}\big)$ for $i = 0, 1, \ldots, 15$. Below, we present a method to identify these pairs of bytes.

In particular, the method obtains the following sets:

$\mathcal{S}_i^{(9)}$ $(0 \leq i \leq 15)$: the set containing the pair of indices (i_1, i_2) indicating the bytes $\big(z_{i_1}^{(9)}, z_{i_2}^{(9)}\big)$ associated with each ciphertext byte c_i $(0 \leq i \leq 15)$. Due to the omission of the `MixColumns` operation in the final round, each output byte of $E^{(10)}$ depends on a single input byte. As a result, the sets $\mathcal{S}_i^{(9)}$ can easily be assigned to the corresponding input bytes $y_i^{(9)}$ $(0 \leq i \leq 15)$ by applying the inverse `ShiftRows` operation;

$\mathcal{S}^{\Phi}$: the set containing the indices of the bytes of $Z^{(9)}$ that affect the 4-byte input $\Phi(P)$ of O_Φ. Recall that $\Phi(P)$ depends on at least 7 (and most likely 8) bytes of $Z^{(9)}$ due to specific characteristics of $\big(M^{(9)}\big)^{-1}$ such that the set $\mathcal{S}^{\Phi}$ contains either 7 or 8 indices.

The method described below exploits certain capabilities of an attacker in the white-box attack context such as having access to the system of polynomials representing $\overline{E}^{(10)}$ and the ability to manipulate (e.g., injecting faults) the internal state $Z^{(9)}$. Hence, he can freely choose to modify bytes of $Z^{(9)}$ and observe the corresponding output $Z^{(10)}$ of $\overline{E}^{(10)}$. In a nutshell, the method identifies those bytes of $Z^{(9)}$ that affect each output byte $z_i^{(10)}$ $(0 \leq i \leq 15)$ of $\overline{E}^{(10)}$. Recall that the 16-byte output $Z^{(10)}$ of $\overline{E}^{(10)}$ is defined as $Z^{(10)} = C \oplus O_\Phi\big(\Phi(P)\big)$, therefore it is crucial that an attacker can distinguish the output of the perturbation cancellation system O_Φ from the ciphertext C in order to construct the sets defined above. This is achieved by the following two steps, which need to be repeated for each byte $z_l^{(9)}$ $(0 \le l \le 42)$ of $Z^{(9)}$ one at a time:

Step 1.1.1: Make $z_l^{(9)}$ active by injecting a fault $\delta \in \mathbb{F}_{256} \backslash \{0\}$, i.e, $\bar{z}_l^{(9)} = z_l^{(9)} \oplus \delta$, and keep all remaining 42 bytes of $Z^{(9)}$ fixed to their initial value. The modified 43-byte value is denoted by $\overline{Z}^{(9)}$;

Step 1.1.2: Compute $\overline{E}^{(10)}\big(\overline{Z}^{(9)}\big)$ and compare its output $Z^{(10)}$ with the stored ciphertext C, i.e., count the number F of active output bytes $z_i^{(10)}$.

$$\text{If} \begin{cases} 1) \quad F = 0 & , \text{ then do nothing .} \\ 2) \quad 0 < F \le 5 & , \text{ then assign } l \text{ to each set } \mathcal{S}_i^{(9)} \text{ associated with} \\ & \text{ the active output bytes } z_i^{(10)} \text{ .} \\ 3) \quad F > 5 & , \text{ then assign } l \text{ to the set } \mathcal{S}^\Phi \text{ .} \end{cases}$$

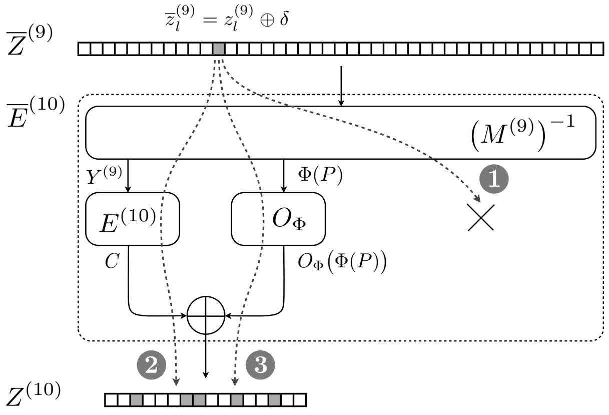


Figure 6.3: The effect of an active byte $\bar{z}_l^{(9)}$ on the bytes of the output $Z^{(10)}$.

The three different cases that occur in Step 1.1.2 are explained in the following (Fig. 6.3). In the case that the active byte $\bar{z}_l^{(9)}$ only affects the input of the random system of polynomials – which has been discarded in $\overline{E}^{(10)}$ – the number of affected output bytes is zero [case (1)]. In the case that the active byte $\bar{z}_l^{(9)}$ affects the input $Y^{(9)}$ of $E^{(10)}$ through one of the $\left(A_i^{(9)}\right)^{-1}$ matrices, the maximum number of affected input bytes of $E^{(10)}$ equals five since $\left(A_i^{(9)}\right)^{-1}$ are 5×5 non-singular matrices over $\mathbb{F}_{256}$. This naturally translates to a maximum of five affected ciphertext bytes due to the omission of the `MixColumns` operation in $E^{(10)}$ and accordingly to a maximum of five affected output bytes in $Z^{(10)}$ [case (2)]. So the case there are more than five affected output bytes in $Z^{(10)}$ only occurs when the active byte $\bar{z}_l^{(9)}$ influences the input $\Phi(P)$ of O_Φ through $\left(B^{(9)}\right)^{-1}$, which causes the output of O_Φ to change in more than five bytes with a high probability [case (3)]. However, with a very low probability, only five or less bytes are affected in the output of O_Φ which introduces a false positive. Such a false positive corresponds to the incorrect assignment of l to the sets $\mathcal{S}_i^{(9)}$ instead of the set $\mathcal{S}^\Phi$. The presence of false positives is discussed below.

The probability of false positives. Recall from the setup phase (Sect. 6.3.1) that we selected an instance for which the randomly chosen plaintext resulted in the correct ciphertext. As a result, the 4-byte input $\Phi(P)$ of O_Φ equals the predefined value $(\varphi_1, \varphi_2, \varphi_3, \varphi_4)$ such that O_Φ outputs zero and thus cancels the introduced perturbation. However, in case (3) described above, the active byte $\bar{z}_l^{(9)}$ modifies the predefined value $(\varphi_1, \varphi_2, \varphi_3, \varphi_4)$ such that O_Φ outputs a 16-byte random value. This random value is XOR'ed with the ciphertext C resulting in the 16-byte output $Z^{(10)}$. Hence, the probability that the number of affected output bytes in $Z^{(10)}$ is five or less is given by

$$\sum_{i=1}^{5} \binom{16}{i} (1/2^8)^{16-i}(1-1/2^8)^i \approx 1/2^{76} \approx 0 \ .$$

This also corresponds with the probability that false positives occur. Now, recall that the input of O_Φ can be affected by either 7 or 8 different bytes of $Z^{(9)}$. Hence, the probability that no false positives occur is given by

$$\text{Prob(no false positives)} \approx (1-1/2^{76})^a \approx 1 \qquad \text{with } a = 7 \text{ or } 8 \ .$$

As a result, with a probability of almost one, the sets $\mathcal{S}_i^{(9)}$ $(0 \le i \le 15)$ contain exactly the pair of indices (i_1, i_2) indicating bytes $\left(z_{i_1}^{(9)}, z_{i_2}^{(9)}\right)$ associated with $y_i^{(9)}$, and the set $\mathcal{S}^\Phi$ contains the 7 or 8 indices indicating the bytes of $Z^{(9)}$ affecting the input of O_Φ. The 'very unlikely' scenario in which false positives do occur, is discussed in [38, Appendix A.1].

Remark 1 (Circumventing the perturbations). *Observe that the obtained set $\mathcal{S}^\Phi$ allows an attacker to circumvent the introduced perturbations. Indeed, by keeping the 7 or 8 bytes of $Z^{(9)}$ identified by $\mathcal{S}^\Phi$ fixed to their original value ensures that the 4-byte input $\Phi(P)$ of O_Φ remains unmodified, i.e. $(\varphi_1, \varphi_2, \varphi_3, \varphi_4)$. As a consequence, the output of O_Φ remains zero and hence the attacker always obtains the correct ciphertext $Z^{(10)} = C$ for any given plaintext.*

Step 1.2: Partially extract the secret linear input encoding $\left(M^{(9)}\right)^{-1}$

The second step of Phase 1 leads to the following result. The details of Step 1.2 are described below.

Result 1. *Given the sets $\mathcal{S}_i^{(9)}$ ($0 \le i \le 15$) and the capability to detect collisions in each input byte of the original final round $E^{(10)}$, then the attacker is able to retrieve the* 16×43 *matrix $G^{(9)}$ over $\mathbb{F}_{256}$ defined as*

$$G^{(9)} = \Lambda^{(9)} \circ \left(M^{(9)}\right)^{-1}_{[0-15]} \quad \textit{with} \quad \Lambda^{(9)} = diag(\alpha_0^{(9)}, \alpha_1^{(9)}, \ldots, \alpha_{15}^{(9)}) \ ,$$

where $diag(\cdot)$ denotes a diagonal matrix.

After Step 1.1, the attacker has identified the pair of bytes $\left(z_{i_1}^{(9)}, z_{i_2}^{(9)}\right)$ of $Z^{(9)}$ (based on the pair of indices (i_1, i_2) contained in $\mathcal{S}_i^{(9)}$) associated with each input byte $y_i^{(9)}$ $(0 \le i \le 15)$ of $E^{(10)}$ for which there exists an unknown linear combination in $\mathbb{F}_{256}$ such that

$$a_i \otimes z_{i_1}^{(9)} \oplus b_i \otimes z_{i_2}^{(9)} = y_i^{(9)} \quad \text{with} \quad a_i, b_i \in \mathbb{F}_{256} \setminus \{0\} \ ,$$

where both constants a_i, b_i are secret. Recall that $\oplus$ and $\otimes$ denote the addition and multiplication in $\mathbb{F}_{256}$ as specified in FIPS 197 [69]. Further, recall that the constants a_i, b_i are the only non-zero coefficients on row i of the secret linear input encoding $\left(M^{(9)}\right)^{-1}$ for $i = 0, 1, \ldots, 15$. Step 1.2 retrieves both secret constants a_i, b_i up to an unknown factor $\alpha_i^{(9)} \in \mathbb{F}_{256} \setminus \{0\}$ that enables the attacker to observe the linear equivalent input byte $\alpha_i^{(9)} \otimes y_i^{(9)}$ for $i = 0, 1, \ldots, 15$.

In order to achieve this, Step 1.2 exploits collisions in the ciphertext byte c_i that naturally translates to collisions in the corresponding input byte $y_i^{(9)}$ since there is a one-to-one relation between input and output bytes for the final round $E^{(10)}$. Recall from Remark 1 that the attacker has access to the ciphertext for any modified value of $Z^{(9)}$ as long as the 7 or 8 bytes of $Z^{(9)}$ identified by $\mathcal{S}^{\Phi}$ remain unchanged.

Now, the following two steps need to be repeated for $i = 0, 1, \ldots, 15$ (Fig. 6.4):

Step 1.2.1: Let $\left(z_{i_1}^{(9)}, z_{i_2}^{(9)}\right)$ denote the initial stored values for the pair of bytes associated with $y_i^{(9)}$. Next, find a new pair of values $\left(\bar{z}_{i_1}^{(9)}, \bar{z}_{i_2}^{(9)}\right)$ that yield a collision in $y_i^{(9)}$ by fixing $\bar{z}_{i_1}^{(9)} = z_{i_1}^{(9)} \oplus \texttt{01}$ and vary $\bar{z}_{i_2}^{(9)}$ over all 2^8 possible values until a collision is detected in the ciphertext byte c_i corresponding to $y_i^{(9)}$. As a result, the attacker obtains

$$\begin{aligned} a_i \otimes z_{i_1}^{(9)} \oplus b_i \otimes z_{i_2}^{(9)} &= y_i^{(9)} \text{ and} \\ a_i \otimes \left(z_{i_1}^{(9)} \oplus \texttt{01}\right) \oplus b_i \otimes \bar{z}_{i_2}^{(9)} &= y_i^{(9)} \ , \end{aligned}$$

from which it follows that $a_i = \varepsilon_i \otimes b_i$ with $\varepsilon_i = z_{i_2}^{(9)} \oplus \bar{z}_{i_2}^{(9)}$.

<u>*Step 1.2.2:*</u> By exploiting the relation between a_i and b_i obtained in Step 1.2.1, we get

$$\varepsilon_i \otimes z_{i_1}^{(9)} \oplus \texttt{01} \otimes z_{i_2}^{(9)} = \alpha_i^{(9)} \otimes y_i^{(9)} \quad \text{with} \quad \alpha_i^{(9)} = b_i^{-1} \ ,$$

where $\alpha_i^{(9)}$ is still unknown. Hence, only the linear equivalent input byte $\alpha_i^{(9)} \otimes y_i^{(9)}$ can be recovered.

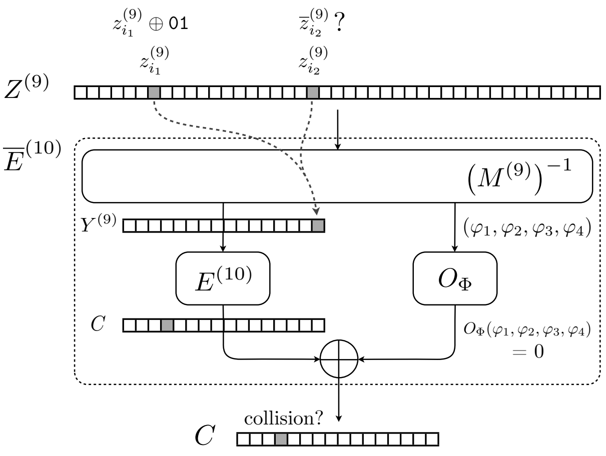

$z_{i_1}^{(9)} \oplus \texttt{01}$
$\bar{z}_{i_2}^{(9)}$?
$z_{i_1}^{(9)}$
$z_{i_2}^{(9)}$
$Z^{(9)}$
$\overline{E}^{(10)}$
$\left(M^{(9)}\right)^{-1}$
$Y^{(9)}$
$(\varphi_1, \varphi_2, \varphi_3, \varphi_4)$
$E^{(10)}$
O_Φ
C
$O_\Phi(\varphi_1, \varphi_2, \varphi_3, \varphi_4) = 0$
collision?
C

Figure 6.4: Finding collisions in a ciphertext byte c for two different pairs of bytes of $Z^{(9)}$ associated with c.

After Steps 1.2.1 and 1.2.2, the attacker is able to obtain an expression of the first 16 rows of $\left(M^{(9)}\right)^{-1}$ up to a 16×16 diagonal matrix $\Lambda^{(9)}$ over $\mathbb{F}_{256}$ containing the unknown coefficients $\alpha_i^{(9)}$ $(0 \le i \le 15)$ on its diagonal, i.e., he recovers the 16×43 matrix $G^{(9)}$ over $\mathbb{F}_{256}$ defined as

$$G^{(9)} = \Lambda^{(9)} \circ \left(M^{(9)}\right)^{-1}_{[0-15]} \ .$$

The i-th row of $G^{(9)}$ is all 00's except for the values ε_i and 01 in the columns i_1 and i_2, respectively, where the pair of indices (i_1, i_2) are in the set $\mathcal{S}_i^{(9)}$ for $0 \le i \le 15$. As a result, the attacker is able to observe the linear equivalent input $\Lambda^{(9)}\left(Y^{(9)}\right)$ of $E^{(10)}$ by computing

$$\Lambda^{(9)}\left(Y^{(9)}\right) = G^{(9)}\left(Z^{(9)}\right) \ .$$

6.3.3 Phase 2: Eliminate the `MixColumns` step from the penultimate round

After Phase 1, the attacker has access to the linear equivalent input $\Lambda^{(9)}\big(Y^{(9)}\big)$ of the original final round $E^{(10)}$ that naturally corresponds to the linear equivalent output of the original preceding (penultimate) round $E^{(9)}$. As a result, the attacker has access to the mapping $e^{(9)} : \mathbb{F}_{256}^{43} \to \mathbb{F}_{256}^{16}$, defined as

$$\begin{aligned} e^{(9)} &= G^{(9)} \circ \overline{E}^{(9)} \\ &= \Lambda^{(9)} \circ \texttt{MixColumns} \circ \texttt{ShiftRows} \circ \big(S_0^{(9)}, \ldots, S_{15}^{(9)}\big) \circ \bigoplus_{k^{(9)}} \circ \big(M^{(8)}\big)^{-1}_{[0-15]} \ . \end{aligned}$$

The mapping $e^{(9)}$ takes as input the 43-byte value $Z^{(8)}$ and outputs the 16-byte value $\Lambda^{(9)}\big(Y^{(9)}\big)$. Recall that the round key $k^{(9)}$ as well as the 16 different S-boxes $S_i^{(9)}$ $(0 \le i \le 15)$ are part of the secret AES* key.

The objective of the second phase of the cryptanalysis is to include the unknown coefficients $\alpha_i^{(9)}$ $(0 \le i \le 15)$ on the diagonal of $\Lambda^{(9)}$ in the secret S-boxes $\big(S_0^{(9)}, \ldots, S_{15}^{(9)}\big)$. This can be achieved by eliminating the `MixColumns` operation from the penultimate round $E^{(9)}$. However, this is not a trivial task since the `MixColumns` operation is an invertible linear transformation that operates on four bytes of the AES state simultaneously. This section addresses this problem and presents the following result.

Result 2. *With black-box access to the mapping $e^{(9)}$, the attacker is able to find an invertible linear mapping $F^{(9)} : \mathbb{F}_{256}^{16} \to \mathbb{F}_{256}^{16}$ such that* `MixColumns` *and $\tilde{\Lambda}^{(9)} = F^{(9)} \circ \Lambda^{(9)}$ commute. As a result, the mapping $e^{(9)}$ is transformed into the mapping $\tilde{e}^{(9)} : \mathbb{F}_{256}^{43} \to \mathbb{F}_{256}^{16}$ defined as*

$$\begin{aligned} \tilde{e}^{(9)} &= F^{(9)} \circ e^{(9)} \\ &= \textit{MixColumns} \circ \tilde{\Lambda}^{(9)} \circ \textit{ShiftRows} \circ \big(S_0^{(9)}, \ldots, S_{15}^{(9)}\big) \circ \bigoplus_{k^{(9)}} \circ \big(M^{(8)}\big)^{-1}_{[0-15]} \ . \end{aligned}$$

Observe that `MixColumns` *becomes the final operation and thus can be eliminated. Furthermore, since* `ShiftRows` *is just a byte transposition (i.e., a permutation on byte level), the unknown diagonal entries (bytes) of $\tilde{\Lambda}^{(9)}$ can be included in the secret S-boxes.*

Recall from Sect. 2.3.1 that `MixColumns` applies four instances of the `MixColumns` operation in parallel; it is represented by a 16×16 non-singular block diagonal matrix over $\mathbb{F}_{256}$ defined as $(\texttt{MC}, \texttt{MC}, \texttt{MC}, \texttt{MC})$, where `MC` denotes the 4×4 non-singular matrix over $\mathbb{F}_{256}$ representing the `MixColumns` operation. Hence, the only way to let $\Lambda^{(9)}$ and `MixColumns` commute is to ensure that the four diagonal entries of $\Lambda^{(9)}$ associated with each `MixColumns` operation are identical, i.e., multiplication with a diagonal matrix with all the same elements is a commutative operation in the group of square matrices.

More specifically, in accordance with the matrix $(\mathtt{MC}, \mathtt{MC}, \mathtt{MC}, \mathtt{MC})$, the 16×16 diagonal matrix $\Lambda^{(9)}$ can be partitioned into four 4×4 diagonal submatrices $\Lambda_i^{(9)}$ $(0 \leq i \leq 3)$ such that $\Lambda^{(9)}$ is defined as the block diagonal matrix $(\Lambda_0^{(9)}, \Lambda_1^{(9)}, \Lambda_2^{(9)}, \Lambda_3^{(9)})$. Each submatrix $\Lambda_i^{(9)}$ $(0 \leq i \leq 3)$ is defined as

$$\Lambda_i^{(9)} = diag(\alpha_{4i}^{(9)}, \alpha_{4i+1}^{(9)}, \alpha_{4i+2}^{(9)}, \alpha_{4i+3}^{(9)}) \ .$$

Below, we present a method that finds a linear transformation f_i such that

$$f_i \circ \Lambda_i^{(9)} = \tilde{\Lambda}_i^{(9)} = diag(\alpha_{4i}^{(9)}, \alpha_{4i}^{(9)}, \alpha_{4i}^{(9)}, \alpha_{4i}^{(9)}) \ ,$$

for $0 \leq i \leq 3$. As a result, $\tilde{\Lambda}_i^{(9)} \circ \mathtt{MC} = \mathtt{MC} \circ \tilde{\Lambda}_i^{(9)}$ from which follows that

$$\begin{aligned}(f_0, f_1, f_2, f_3) \circ \Lambda^{(9)} \circ \mathtt{MixColumns} &= (\tilde{\Lambda}_0^{(9)}, \tilde{\Lambda}_1^{(9)}, \tilde{\Lambda}_2^{(9)}, \tilde{\Lambda}_3^{(9)}) \circ (\mathtt{MC}, \mathtt{MC}, \mathtt{MC}, \mathtt{MC}) \\ &= (\mathtt{MC}, \mathtt{MC}, \mathtt{MC}, \mathtt{MC}) \circ (\tilde{\Lambda}_0^{(9)}, \tilde{\Lambda}_1^{(9)}, \tilde{\Lambda}_2^{(9)}, \tilde{\Lambda}_3^{(9)}) \\ &= \mathtt{MixColumns} \circ \tilde{\Lambda}^{(9)} \ .\end{aligned}$$

Hence the mapping $\tilde{e}^{(9)} = F^{(9)} \circ e^{(9)}$ with $F^{(9)} = (f_0, f_1, f_2, f_3)$. This concludes the algorithm for Result 2.

Method for finding f_i for $0 \leq i \leq 3$. In the following, let the four parallel `MixColumns` operations be numbered from left to right such that $\mathtt{MC}_i$ denotes the i-th `MixColumns` operation for $0 \leq i \leq 3$. Recall that $Z^{(8)}$ and $\Lambda^{(9)}(Y^{(9)})$ denote the 43-byte input and 16-byte output of the mapping $e^{(9)}$. Now, the method starts by constructing the following sets:

$\mathcal{S}_i$ $(0 \leq i \leq 3)$**:** the set containing the indices of the bytes of $Z^{(8)}$ that affect the 4-byte input of the i-th `MixColumns` operation $\mathtt{MC}_i$.

Each set $\mathcal{S}_i$ $(0 \leq i \leq 3)$ is constructed by making each byte of $Z^{(8)}$ one at a time active and observe the corresponding active output bytes in $\Lambda^{(9)}(Y^{(9)})$. Due to the specific characteristics of the partially secret linear input encoding $(M^{(8)})^{-1}_{[0-15]}$, i.e., each row contains exactly two non-zero elements in $\mathbb{F}_{256}$, making one of the bytes of $Z^{(8)}$ identified by $\mathcal{S}_i$ active results in making one or two out of four input bytes of $\texttt{MC}_i$ active in most cases.

Next, repeat the following steps for each $\texttt{MixColumns}$ operation $\texttt{MC}_i$ $(0 \leq i \leq 3)$:

Step 2.1: Given the unmodified 43-byte value of $Z^{(8)}$, compute $e^{(9)}\big(Z^{(8)}\big)$ and store that part of the output that is associated with $\texttt{MC}_i$. Denote the stored 4-byte value by Y_i, given by

$$Y_i = \big(\alpha^{(9)}_{4i} \otimes y^{(9)}_{4i}, \alpha^{(9)}_{4i+1} \otimes y^{(9)}_{4i+1}, \alpha^{(9)}_{4i+2} \otimes y^{(9)}_{4i+2}, \alpha^{(9)}_{4i+3} \otimes y^{(9)}_{4i+3}\big) \ .$$

Step 2.2: Make one of the bytes of $Z^{(8)}$ identified by $\mathcal{S}_i$ active and denote the modified 43-byte value by $\overline{Z}^{(8)}$. Compute $e^{(9)}\big(\overline{Z}^{(8)}\big)$ and store that part of the output that is associated with $\texttt{MC}_i$. Denote the stored 4-byte value by $\overline{Y}_i$. By exploiting that the branch number of the $\texttt{MixColumns}$ operation is equal to five, the case that less than three bytes between Y_i and $\overline{Y}_i$ have become active can be discarded since this indicates that at least three out of four input bytes of $\texttt{MC}_i$ have become active. In that case, go to Step 2.4.

Step 2.3: Given the pair of 4-byte values $(Y_i, \overline{Y}_i)$ associated with the output of $\texttt{MC}_i$, apply the 4×4 matrix D over $\mathbb{F}_{256}$ defined as

$$D = \texttt{MC}^{-1} \circ diag(\texttt{01}, a, b, c)$$

to both values $(Y_i, \overline{Y}_i)$ for each possible triplet $(a, b, c) \in (\mathbb{F}_{256} \setminus \{0\})^3$. In other words, keep $\alpha^{(9)}_{4i}$ fixed to its original value while varying the other three coefficients $\alpha^{(9)}_{4i+1}, \alpha^{(9)}_{4i+2}, \alpha^{(9)}_{4i+3}$ each over $\mathbb{F}_{256} \setminus \{0\}$ and invert the $\texttt{MixColumns}$ operation. Next, for each possible triplet (a, b, c), count the number $\#_{(a,b,c)}$ of active bytes between $D(Y_i)$ and $D(\overline{Y}_i)$ and construct the set

$$\mathcal{T} = \big\{(a, b, c) \mid \#_{(a,b,c)} = 1 \vee \#_{(a,b,c)} = 2\big\} \ .$$

Step 2.4: Repeat Steps 2.2-2.3 for each byte of $Z^{(8)}$ identified by $\mathcal{S}_i$ one at a time.

Step 2.5: After Steps 2.1-2.4, a set $\mathcal{T}$ is constructed for each modified byte in $Z^{(8)}$ identified by $\mathcal{S}_i$. Next, take these sets $\mathcal{T}$ pairwise together and derive their intersections. Since the size of $\mathcal{S}_i$ is at most eight (i.e., two bytes of $Z^{(8)}$ are associated with each of the four input bytes of $\texttt{MC}_i$), we get at most a total of $\binom{8}{2} = 28$ intersections. Finally, select the triplet (a, b, c) that appears most often in all intersections and denote this triplet by (A, B, C). For (A, B, C) then holds that

$$(A \otimes \alpha^{(9)}_{4i+1}, B \otimes \alpha^{(9)}_{4i+2}, C \otimes \alpha^{(9)}_{4i+3}) = (\alpha^{(9)}_{4i}, \alpha^{(9)}_{4i}, \alpha^{(9)}_{4i})$$

with overwhelming probability. The linear transformation f_i is then defined as

$$f_i = diag(\texttt{01}, A, B, C) \ .$$

To elaborate on the method described above, we discuss the following two cases that can be distinguished after making one of the bytes of $Z^{(8)}$ identified by $\mathcal{S}_i$ active:

1. the 'most likely' case: one or two out of four input bytes of $\texttt{MC}_i$ have become active. The constructed set $\mathcal{T}$ obtained in Step 2.3 is then considered to be a *valid set* and contains (A, B, C) since the triplet (A, B, C) would result in $\#_{(a,b,c)} = 1$ or 2 which is considered in the construction of $\mathcal{T}$. In fact, the triplet (A, B, C) is contained within all valid sets.

2. the 'less likely' case: at least three out of four input bytes of $\texttt{MC}_i$ have become active and the case has not been discarded in Step 2.2. The constructed set $\mathcal{T}$ obtained in Step 2.3 is then considered to be an *invalid set* since it cannot contain (A, B, C), i.e., the triplet (A, B, C) would result in $\#_{(a,b,c)} = 3$ or 4 which is discarded in the construction of $\mathcal{T}$.

As a result, only the intersections between valid sets contain (A, B, C) during the execution of Step 2.5.

Implementation. The method described above to find f_i $(0 \leq i \leq 3)$ has been successfully implemented in `C++` and confirmed by computer experiments. The computer experiments showed that each set $\mathcal{T}$ obtained in Step 2.3 contains either 1510 of 1530 triplets (a, b, c). Considering the size of the search space (i.e., $(2^8 - 1)^3$), each set $\mathcal{T}$ contains a fraction (i.e., less than 0.01%) of all possible triplets. Further, the implementation always succeeded in finding the single correct triplet (A, B, C) by taking pairwise intersections of all sets $\mathcal{T}$.

6.3.4 Phase 3: Structurally decompose all rounds

The third phase fully structurally decomposes the perturbated white-box AES* implementation; this involves the recovery of all secret linear encodings up to an unknown diagonal mapping that can be included in the key-dependent S-boxes. This is achieved by means of the following main result.

Result 3. *With black-box access to the mapping* $\tilde{e}^{(r)} : \mathbb{F}_{256}^{43} \rightarrow \mathbb{F}_{256}^{16}$ *defined as*

$$\tilde{e}^{(r)} = \texttt{MixColumns} \circ \tilde{\Lambda}^{(r)} \circ \texttt{ShiftRows} \circ \left(S_i^{(r)}\right)_{0 \leq i \leq 15} \circ \bigoplus_{k^{(r)}} \circ \left(M^{(r-1)}\right)^{-1}_{[0-15]} \ ,$$

where $\tilde{\Lambda}^{(r)}$ *is an unknown* 16×16 *diagonal matrix over* $\mathbb{F}_{256}$*, the attacker is able to retrieve the partially secret linear input encoding* $\left(M^{(r-1)}\right)^{-1}_{[0-15]}$ *up to an unknown* 16×16 *diagonal matrix* $\tilde{\Lambda}^{(r-1)}$ *over* $\mathbb{F}_{256}$*. Furthermore,* `MixColumns` *and* $\tilde{\Lambda}^{(r-1)}$ *commute. This allows the attacker to obtain access to the mapping* $\tilde{e}^{(r-1)} : \mathbb{F}_{256}^{43} \to \mathbb{F}_{256}^{16}$ *(or* $\mathbb{F}_{256}^{16} \to \mathbb{F}_{256}^{16}$ *if* $r = 2$*) defined as*

$$\tilde{e}^{(r-1)} = \tilde{\Lambda}^{(r-1)} \circ \left(M^{(r-1)}\right)^{-1}_{[0-15]} \circ \overline{E}^{(r-1)} \ ,$$

whose composition is given by

$$\mathtt{MixColumns} \circ \tilde{\Lambda}^{(r-1)} \circ \mathtt{ShiftRows} \circ \left(S_i^{(r-1)}\right)_{0 \le i \le 15} \circ \bigoplus_{k^{(r-1)}} \circ \left(M^{(r-2)}\right)^{-1}_{[0-15]} \ ,$$

for $2 \le r \le 9$*, where* $\left(M^{(r-2)}\right)^{-1}_{[0-15]}$ *is omitted if* $r = 2$*.*

Observe that Result 3 is recursive by nature, i.e., it can be applied to rounds $2 \le r \le 9$ one at a time from the bottom up. Hence, the starting point of the recursion is the mapping $\tilde{e}^{(9)}$ to which the adverary already has access after Phases 1 and 2 (see Result 2).

The algorithm that leads to Result 3 consists of the following steps:

Step 3.1: The construction of the sets $\mathcal{S}_i^{(r-1)}$ $(0 \le i \le 15)$ containing the indices indicating the pair of bytes of $Z^{(r-1)}$ associated with each input byte $y_i^{(r-1)}$ $(0 \le i \le 15)$ of the original round function $E^{(r)}$.

Step 3.2: The retrieval of the 16×43 matrix $G^{(r-1)}$ over $\mathbb{F}_{256}$ defined as

$$G^{(r-1)} = \Lambda^{(r-1)} \circ \left(M^{(r-1)}\right)^{-1}_{[0-15]} \ ,$$

where $\Lambda^{(r-1)} = diag(\alpha_0^{(r-1)}, \alpha_1^{(r-1)}, \ldots, \alpha_{15}^{(r-1)})$. This is identical to Result 1, but then applied to the general case r. Now, the attacker has access to the mapping $e^{(r-1)}$ defined as $e^{(r-1)} = G^{(r-1)} \circ \overline{E}^{(r-1)}$.

Step 3.3: The retrieval of the invertible linear mapping $F^{(r-1)} : \mathbb{F}_{256}^{16} \to \mathbb{F}_{256}^{16}$ such that `MixColumns` and $\tilde{\Lambda}^{(r-1)} = F^{(r-1)} \circ \Lambda^{(r-1)}$ commute. This is identical to Result 2, but then applied to the general case r. Now, the attacker has access to the mapping $\tilde{e}^{(r-1)}$ defined as

$$\tilde{e}^{(r-1)} = F^{(r-1)} \circ e^{(r-1)} = \tilde{\Lambda}^{(r-1)} \circ \left(M^{(r-1)}\right)^{-1}_{[0-15]} \circ \overline{E}^{(r-1)} \ .$$

This concludes the algorithm to achieve Result 3.

For a detailed description of the steps above, refer to us [38, Appendix A.2]. Note that the methodology of Steps 3.2 and 3.3 is almost identical to the descriptions given for Phase 1 (Step 1.2) and Phase 2, but then applied to the general case r. Observe that the diffusion effect of `MixColumns` can be eliminated by applying the inverse `MixColumns` operation to the output of $\tilde{e}^{(r)}$. As a result, there is a one-to-one relation between input and output bytes for $E^{(r)}$ such that an active input byte (through the corresponding active output byte – necessary for Step 3.1) or collision in an input byte (through a collision in the corresponding output byte – necessary for Step 3.2) can be observed. Further, note that Step 3.3 is significantly simpler if $r = 2$ since the first round lacks a secret linear input encoding.

6.3.5 Phase 4: Extract an equivalent key

After Phases 1-3, the attacker can observe the linear equivalent input and output of each original round function $E^{(r)}$ $(1 \leq r \leq 10)$, denoted by

$$\tilde{\Lambda}^{(r)}\big(Y^{(r)}\big) \quad \text{and} \quad \tilde{\Lambda}^{(r-1)}\big(Y^{(r-1)}\big) \ ,$$

respectively, with the exception of the plaintext and ciphertext for the first and final round, respectively. Hence, the attacker has access to the mappings $\hat{E}^{(r)} : \mathbb{F}_{256}^{16} \to \mathbb{F}_{256}^{16}$ defined as

$$\begin{aligned}
\hat{E}^{(1)} &= \texttt{MixColumns} \circ \texttt{ShiftRows} \circ \tilde{\Lambda}^{(1)}_{\text{sr}} \circ \big(S_i^{(1)}\big)_{0\leq i\leq 15} \circ \oplus_{k^{(1)}} \ , \\
\hat{E}^{(r)} &= \texttt{MixColumns} \circ \texttt{ShiftRows} \circ \tilde{\Lambda}^{(r)}_{\text{sr}} \circ \big(S_i^{(r)}\big)_{0\leq i\leq 15} \circ \oplus_{k^{(r)}} \circ \big(\tilde{\Lambda}^{(r-1)}\big)^{-1}, \\
\hat{E}^{(10)} &= \texttt{ShiftRows} \circ \oplus_{k^{(11)}_{\text{sr}}} \circ \big(S_i^{(10)}\big)_{0\leq i\leq 15} \circ \oplus_{k^{(10)}} \circ \big(\tilde{\Lambda}^{(9)}\big)^{-1} \ ,
\end{aligned}$$

with $k^{(11)}_{\text{sr}} = \texttt{SR}^{-1}(k^{(11)})$ and $\tilde{\Lambda}^{(r)}_{\text{sr}} = \texttt{SR}^{-1} \circ \tilde{\Lambda}^{(r)} \circ \texttt{SR}$ for $1 \leq r \leq 9$, where the mapping $\texttt{SR} : \mathbb{F}_{256}^{16} \to \mathbb{F}_{256}^{16}$ denotes the permutation defining `ShiftRows`. Furthermore, observe that

$$\hat{E}^{(10)} \circ \hat{E}^{(9)} \circ \cdots \circ \hat{E}^{(1)} = E^{(10)} \circ E^{(9)} \circ \cdots \circ E^{(1)} \ .$$

Next, since `ShiftRows` and `MixColumns` are known to the attacker, he can eliminate these operations at the output of the mappings $\hat{E}^{(r)}$ $(1 \leq r \leq 10)$ by applying their inverses. All remaining operations can be merged into 16 8-bit bijective S-boxes $\hat{S}_i^{(r)}$ $(0 \leq i \leq 15)$ in parallel per round $1 \leq r \leq 10$, defined as

$$\begin{aligned}
\hat{S}_i^{(1)} &= \otimes_{\lambda^{(1)}_{sr^{-1}(i)}} \circ S_i^{(1)} \circ \oplus_{k_i^{(1)}} && \text{for } 0 \leq i \leq 15 \ , \\
\hat{S}_i^{(r)} &= \otimes_{\lambda^{(r)}_{sr^{-1}(i)}} \circ S_i^{(r)} \circ \oplus_{k_i^{(r)}} \circ \otimes_{\left(\lambda_i^{(r-1)}\right)^{-1}} && \text{for } 0 \leq i \leq 15 \text{ and } 2 \leq r \leq 9 \ , \\
\hat{S}_i^{(10)} &= \oplus_{k^{(11)}_{sr^{-1}(i)}} \circ S_i^{(10)} \circ \oplus_{k_i^{(10)}} \circ \otimes_{\left(\lambda_i^{(9)}\right)^{-1}} && \text{for } 0 \leq i \leq 15 \ ,
\end{aligned}$$

where λ_i^r denotes the i-th diagonal element of $\tilde{\Lambda}^{(r)}$ and $sr^{-1} : \{0, 1, \dots, 15\} \rightarrow \{0, 1, \dots, 15\}$ denotes the permutation on the indices of a 16-byte vector as a result of the application of the inverse ShiftRows operation $\texttt{SR}^{-1}$. Observe that these S-boxes are equivalent to the original key-dependent S-boxes $S_i^{(r)}$. The set containing the 160 S-boxes $\{\hat{S}_i^{(r)}\}_{0 \le i \le 15; 1 \le r \le 10}$ forms the *equivalent secret key*. This equivalent key allows the attacker to construct a simpler (in terms of storage requirement) and invertible implementation (Fig. 6.5) that is functionally equivalent to the original AES* block cipher, i.e., the mapping from plaintexts onto ciphertexts is identical. The size of this implementation is given by the storage requirement of all 160 S-boxes $\hat{S}_i^{(r)}$ ($0 \le i \le 15$ and $1 \le r \le 10$) and hence equals $160 \cdot 2^8 \cdot 8$ bits or 40 kB.

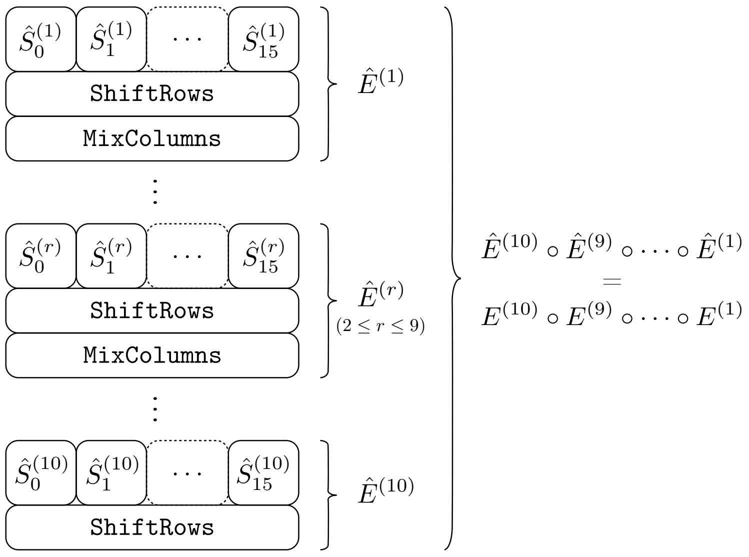


Figure 6.5: Each equivalent key $\hat{S}_i^{(r)}$ ($0 \le i \le 15$ and $1 \le r \le 10$) yields a functionally equivalent AES* implementation.

6.3.6 Work Factor

The work factor of the attack on the perturbated white-box AES* implementation is expressed in the number of evaluations of the system of multivariate polynomial equations over $\mathbb{F}_{256}$ representing the encoded perturbated round functions $\overline{E}^{(r)}$ ($1 \le r \le 10$). Other computations in the attack are assumed to have a negligible work factor when compared to the system evaluations and thus are discarded for the sake of simplicity. The work factor of each phase of the cryptanalysis is listed below.

Setup Phase	$4 \cdot 10$
Phase 1	$43 + 16 \cdot 2^8$
Phase 2	$43 + 4 \cdot 8$
Phase 3	$8 \cdot (43 + 16 \cdot 2^8) + 7 \cdot (43 + 4 \cdot 8) + 4 \cdot 4$
Phase 4	$10 \cdot 16 \cdot 2^8$
Total work factor	$78\,867 \approx 2^{16}$

The overall work factor of the cryptanalysis is dominated by the work factors of Phase 3 and 4 and equals $78\,867 \approx 2^{16}$ evaluations.

6.4 Conclusion

In response to the proven insecurity of the white-box AES implementation of Chow et al., Bringer et al. [20] proposed a novel white-box technique and applied their technique to AES*, a variant of AES-128 in which all 160 S-boxes are different and key-dependent, resulting in a perturbated white-box AES* implementation. In this chapter, we presented an attack on the perturbated white-box AES* implementation. The work factor is given by $\approx 2^{16}$ and is expressed in the number of evaluations of systems of multivariate polynomials over $\mathbb{F}_{256}$. Naturally, the presented cryptanalysis also applies to a perturbated white-box AES implementation.

The presented attack extracts an *equivalent key* comprising a set of 160 8-bit bijective S-boxes. Many equivalent keys are possible: each equivalent key can be used to construct an implementation that is functionally equivalent to the standard AES*, i.e., both have identical plaintext/ciphertext behavior. Each functionally equivalent implementation has a size of 40 kB. Furthermore, the obtained implementation is invertible (in contrast to the white-box implementation). Note that similar results are obtained in the SASAS cryptanalysis by Biryukov and Shamir [15], i.e., equivalent keys comprising equivalent representations of all key-dependent operations are extracted that yield the same plaintext to ciphertext mapping as the original SASAS cipher.

Although the presented cryptanalysis is specific to the particular composition of the white-box implementation, some methods are of independent interest such as the second phase. Since the cryptanalysis exploits the diffusion operators (`ShiftRows` and `MixColumns`) of AES as well as the specific properties of the secret linear encodings, countermeasures against the attack include modifying these specifications such as making the diffusion operator key-dependent.

Chapter 7

State-of-the-Art and Q&A

This chapter presents the state-of-the-art of lookup-table-based white-box AES implementations published in the academic literature. Both their size and performance as well as their level of white-box security are discussed.

Further, this chapter addresses some questions with regard to the design of new white-box AES implementations. First, a promising technique proposed by Michiels and Gorissen [74] is discussed that may lead to secure white-box AES implementations. Second, two dynamic-key white-box techniques proposed by Michiels [71, 72] are described that allow to update the cryptographic key of white-box implementations. As a contribution to the latter, we present a new dynamic-key white-box technique, published in [91], that allows to update the key in a more secure way than all known techniques.

7.1 State-of-the-Art

Up to now, three different lookup-table-based white-box AES implementations have been discussed: the ones of Chow et al. [23] (Chapter 3), of Karroumi [53] (Chapter 4) and of Xiao and Lai [107] (Chapter 5).

7.1.1 Size and Performance

Table 7.1 presents an overview of the size and performance of these three white-box AES implementations. The performance is not measured in the

Table 7.1: White-box AES implementations show an increased implementation size and a decreased performance (exception is the Xiao-Lai implementation) when compared with a standard AES implementation. This trade-off needs to be considered when designing secure AES implementations in the white-box setting.

Implementation	**Size**				**Performance** (# and type of operations)		
	Lookup Table / Binary Matrix			Total Size	Table Lookup	XOR	Matrix Mult.
	#	Type	Size				
Standard AES Implementation							
Daemen and Rijmen [31]	4	$\mathtt{SMC}_i$ (8-to-32 bit)	4 kB	4 kB	160	152	–
White-Box AES Implementations							
1. Chow, Eisen, Johnson and van Oorschot [23] 2. Karroumi [53]	144	$\mathcal{L}\text{-}\mathtt{II}_i^{(r,j)}$ (8-to-32 bit)	144 kB	752 kB	3008	–	–
	144	$\mathcal{L}\text{-}\mathtt{III}_i^{(r,j)}$ (8-to-32 bit)	144 kB				
	16	$\mathcal{L}\text{-}\mathtt{Ia}_i^{(1,j)}$ (8-to-128 bit)	64 kB				
	16	$\mathcal{L}\text{-}\mathtt{Ib}_i^{(10,j)}$ (8-to-128 bit)	64 kB				
	2688	$\mathcal{L}\text{-}\mathtt{IV}$ (8-to-4 bit)	336 kB				
3. Xiao and Lai [107]	72	$\mathtt{dTMC}_i^{(r,j)}$ (16-to-32 bit)	18 432 kB	20 502 kB	80	40	11
	8	$\mathtt{dT}_i^{(10,j)}$ (16-to-32 bit)	2048 kB				
	11	$M^{(r)}, M^{(11)}$ (128×128)	22 kB				

conventional unit ‘cycles/byte’, but instead is measured in the number and type of performed operations. A distinction is made between three different types of operations: (i) table lookups, (ii) 32-bit XOR operations, and (iii) matrix-vector multiplications over $\mathbb{F}_2$. Additionally, the standard black-box AES implementation proposed by Daemen and Rijmen [31] for processors with word lengths 32 bits or higher (Sect. 2.3.2) is given as a reference. It should be noted that the size of real-world implementations may slightly differ from the values listed in Table 7.1 because of the additional code required to determine the data-flow of the implementation. Interpreting Table 7.1 gives rise to the following observations.

Reuse of lookup tables. Observe that the total number of lookup tables equals the total number of table lookups for all three white-box AES implementations. This implies that none of the lookup tables are reused. However, the reuse of tables may lead to a significant reduction in the total implementation size while leaving the performance (i.e., the number of table lookups) unchanged. As one might expect, only key-*in*dependent lookup tables are eligible for reuse since they do not contain any unique secret key-material. However, it might be possible to convert key-dependent tables into key-*in*dependent tables by including the secret key into other parts of the white-box implementation, as explained below for the Xiao-Lai white-box AES implementation.

With regard to Chow et al.'s white-box AES implementation (recall that Karroumi's implementation belongs to the class of white-box AES implementations specified by Chow et al.), Plasmans described the reuse of lookup tables of this implementation in [86, Sect. 6.5]. There are three types of key-independent lookup tables, i.e., Types Ia, III and IV. For the detailed description of each type, refer to Sect. 3.3.2 and Fig. 3.3. For each type of key-independent tables, it is discussed below to what extent they can be reused.

Type Ia. Recall that the non-singular 128×128 matrix $\mathtt{IN}^{-1}$ over $\mathbb{F}_2$ representing the bijective external input encoding was partitioned into 16 128×8 submatrices $\mathtt{IN}_l^{-1}$ $(0 \leq l \leq 15)$, where each $\mathtt{IN}_l^{-1}$ contains the columns of $\mathtt{IN}^{-1}$ indexed from $8l$ to $8l+7$. Further, recall that each $\mathcal{L}\text{-}\mathtt{Ia}_i^{(1,j)}$ $(0 \leq i, j \leq 3)$ table implements $\mathtt{IN}_{4j+i}^{-1}$. As a result, the Type Ia tables are not suitable for reuse as this would imply that the external input encoding is non-bijective.

Type III. Recall that each set of four Type III tables, given by

$$\mathcal{S}^{(r,j)} = \{\mathcal{L}\text{-}\mathtt{III}_0^{(r,j)}, \mathcal{L}\text{-}\mathtt{III}_1^{(r,j)}, \mathcal{L}\text{-}\mathtt{III}_2^{(r,j)}, \mathcal{L}\text{-}\mathtt{III}_3^{(r,j)}\}$$

for $0 \leq j \leq 3$ and $1 \leq r \leq 9$, implements the 32×32 binary matrix $L^{(r+1,j)} \circ \left(R^{(r,j)}\right)^{-1}$ annihilating the output encodings of round r and introducing the input encodings of round $r+1$. By keeping $L^{(r+1,j)} \circ \left(R^{(r,j)}\right)^{-1}$ the same for $0 \leq j \leq 3$ and $1 \leq r \leq 9$, one needs only a single set of four Type III tables, i.e.,

$$\mathcal{S} = \{\mathcal{L}\text{-III}_0, \mathcal{L}\text{-III}_1, \mathcal{L}\text{-III}_2, \mathcal{L}\text{-III}_3\} \ .$$

As a consequence, the implementation size of the Type III tables is reduced from 144 kB to only 4 kB.

Type IV. The 2688 encoded nibble XOR tables $\mathcal{L}$-`IV` account for $\approx 45\%$ of the total implementation size. Therefore, these tables are particularly suitable for reuse since it can result in a significant reduction of the implementation size. In the most extreme case, all 2688 $\mathcal{L}$-`IV` tables are replaced by one single $\mathcal{L}$-`IV` table. As a result, the cost of implementing the XOR operations is reduced from 336 kB to only 128 bytes.

Taking into account the above suggestions with regard to the reuse of tables, the total implementation size of the white-box AES implementation of Chow et al. becomes ≈ 276 kB, or in other words only $\approx 37\%$ of its original size.

Next, let us consider the Xiao-Lai white-box AES implementation (for a detailed description, refer to Sect. 5.1). At first sight, this implementation is not eligible for the reuse of lookup tables since all tables are key-dependent. However, these tables can be made key-*in*dependent by including the key addition operation into the neighboring 128×128 binary matrices $M^{(r)}$ $(1 \leq r \leq 11)$, which then become affine instead of solely linear. The resulting key-*in*dependent lookup tables can then be replaced by two $\texttt{dTMC}_i$ $(i = 0, 1)$ tables, one for each `MixColumns` submatrix $\texttt{MC}_i$ $(i = 0, 1)$, and only one `dT` table. Taking into account these suggestions with regard to the reuse of tables, the total implementation size of the Xiao-Lai white-box AES implementation becomes ≈ 790 kB, or in other words only $\approx 4\%$ of its original size.

The risk of reusing lookup tables. Even though the reuse of lookup tables is a tempting technique to bring down the implementation cost of white-box implementations, the main disadvantage is that it is inherently associated with the reuse of the involved secret white-box encodings. Furthermore, the reuse of tables combined with the data-flow of the white-box implementation (i.e., the relation between tables) can reveal weak spots in the implementation; these weak spots can for example be repeating patterns (due to the reuse of white-box encodings) in components of the white-box implementation. The weak spots can then be used as starting points for attacks. Hence, the reuse of lookup tables (and of white-box encodings) can pose a real threat to the achieved level of white-box security since the entropy of the white-box implementation decreases. Therefore, it should always be the priority to first design a secure white-box implementation, and then investigate to what extent lookup tables can be reused without compromising the achieved level of white-box security.

The cost of white-box security. Recall from Chapter 2 (Sect. 2.5.3) that the standard black-box software AES implementation is vulnerable to the S-box blanking attack when employed in the white-box model and thus provides no white-box security. Hence, it is clear that the black-box implementation is not intended to be deployed in the white-box attack context. On the other hand, specifically designed white-box AES implementations, such as the three lookup-table-based implementations listed in Table 7.1, are assumed to offer a sufficient level of robustness against a white-box attacker. Observe that this typically comes at a significant cost: i.e., the white-box implementations show an increased implementation size and a decreased performance when compared with the standard black-box implementation. For example, in the case of the white-box AES implementation of Chow et al., the increase in size is about 188x whereas the decrease in performance is estimated at 55x (cf. Chow et al. [23]). There is one exception to this rule, and that is the Xiao-Lai white-box AES implementation that clearly shows a good performance; in fact, its performance even competes with the performance of the black-box implementation.

Depending on the requirements of a given application, one takes the most secure white-box implementation satisfying the requirements. As the available memory only increases over time, the increased implementation size of white-box implementations becomes less of an issue. From that perspective, the Xiao-Lai white-box AES implementation becomes a viable option since it has a good performance in spite of its large implementation size. Furthermore, as is discussed in the next section, it also has the highest level of white-box security of the white-box AES implementations considered in this thesis.

Below, some suggestions are highlighted in order to reduce the cost of lookup-table-based white-box implementations:

1. *Reuse key-independent lookup tables* as explained above in order to reduce the overall implementation size. Be aware that this possibly affects the white-box security.
2. *Use solely $\mathbb{F}_2$-linear or $\mathbb{F}_2$-affine secret white-box encodings* such that XOR operations can be executed on encoded data (e.g., the Xiao-Lai white-box AES implementation) and do not need to be represented as a network of encoded nibble XOR tables. Observe that for Chow et al.'s white-box AES

Table 7.2: The cryptanalytic results on the three lookup-table-based white-box AES implementations discussed in this thesis.

White-Box AES Impl.	**Cryptanalysis**	**Work Factor**
1. Chow, Eisen, Johnson and van Oorschot [23]	BGE attack [13]	2^{30}
2. Karroumi [53]	us [36, 63] (Chapter 4)	2^{22}
3. Xiao and Lai [107]	us [35] (Chapter 5)	2^{32}
- generic linear version	us (Chapter 5)	2^{38}
- affine/non-affine version	Michiels et al.'s attack [75]	at least 2^{49}

implementation, the implementation of all XOR operations by means of 2688 encoded nibble XOR tables accounts for $\approx 45\%$ of the total implementation size and for $\approx 89\%$ of the overall performance. Hence the use of linear or affine encodings can significantly improve the performance of white-box implementations.

3. *Increase the input size of lookup tables* such that more operations can be merged into tables. This results in a decrease in the number of table lookups, however, the downside is the exponential increase in implementation size. Take as an example the Xiao-Lai white-box AES implementation where the input size of all lookup tables is 16 bits.

Even though the three lookup-table-based white-box AES implementations in Table 7.1 were specifically designed to withstand secret key extraction, they all have been shown to be white-box *in*secure as is discussed in the next section.

7.1.2 Cryptanalytic Results

Table 7.2 summarizes the cryptanalytic results discussed in Chapters 3, 4 and 5. Observe that most presented attacks have a practical work factor, i.e., they efficiently extract the embedded secret AES key together with the external encodings, such that the corresponding white-box AES implementation is proven white-box insecure (in fact WBKR-insecure). The only lookup-table-based white-box AES implementation still offering some resistance against a white-box attacker is the one of Xiao and Lai, but with other type of encodings (i.e., affine or non-affine) instead of linear ones (Sect. 5.5). We only consider the affine variant, as this is the only variant that shows implementation size and performance comparable to the original Xiao-Lai white-box AES implementation. But recall from the conclusion of Chapter 5 that there are still two open questions (i.e., optimization and randomization) with regard to this affine variant in order to fully assess its white-box security.

7.2 Questions and Answers

This section discusses (as a series of questions and answers) some aspects of white-box AES implementations that have not yet been addressed in this thesis.

7.2.1 All white-box AES implementations in the academic literature have been proven insecure. Now what?

The primary goal when designing new white-box AES implementations is to ensure that the new design is resistant against all known white-box attacks. This involves (i) identifying the vulnerabilities of the various existing white-box AES implementations that have been exploited by the existing white-box attacks, and (ii) providing countermeasures that avoid the known vulnerabilities and thus preclude the attacks. For Chow et al.'s white-box AES implementation, the vulnerabilities exploited by the BGE attack have already been discussed in Sect. 3.5.2 on p 94. Now, with regard to all white-box AES implementations considered in this thesis, the main vulnerabilities exploited by all known white-box attacks are the following:

1. The intermediate results of the white-box AES implementations are encoded using *fixed* secret bijective white-box encodings. The term 'fixed' refers to the fact that the encodings are invariant for different executions of the white-box implementation with different input data.
2. The entire specification of AES is publicly known except for the secret key. As a result, specific properties of the AES S-box or the diffusion operator (i.e., `ShiftRows` and `MixColumns`) can be exploited.

Countermeasures to avoid the second vulnerability is to hide the algebraic structure of AES without actually altering the specification of AES. For examples, refer to the perturbation white-box technique of Bringer et al. [20] or to the application of dual AES ciphers by Karroumi [53]. Despite their effort of hiding the algebraic structure of AES through either introducing perturbations (i.e., well-defined faults) or keeping the dual transformations secret, we showed in

Chapters 4 and 6 that these attempts failed in designing secure white-box AES implementations. Below, a technique to avoid the first vulnerability is discussed.

Variable White-Box Encodings

In [74], Michiels and Gorissen introduced the concept of *variable white-box encodings*; it is a technique belonging to the class of countermeasures that focuses on avoiding the first vulnerability (i.e., the fixed encodings). The main idea is that the intermediate results of the white-box AES implementation are variably encoded in such a way that different executions of the implementation with different input data yield different encodings applied to the intermediate results. In the following, the concept of variable encodings is explained with regard to a white-box AES implementation as a result of the application of the generic white-box techniques of Chow et al. However, note that the technique of variable encodings is not restricted to white-box AES implementations. For detailed information, refer to Michiels and Gorissen [74].

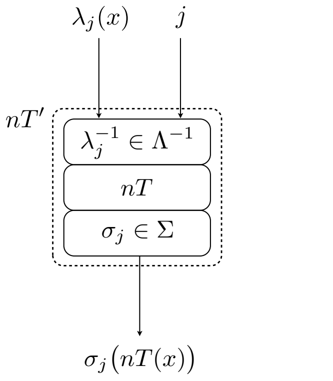
Recall that the generic white-box techniques of Chow et al. comprised two phases: the first phase merged several steps of the round functions in order to form lookup tables, after which the second phase applied *fixed* encodings to the input and output of these lookup tables. Further, recall that the `AddRoundKey` and `SubBytes` steps were typically merged into the so-called 8-bit bijective key-dependent *T-boxes*; a T-box was defined as $T(x) = S(x \oplus k)$, where S denotes the AES S-box and k denotes a secret round key byte. As such, the lookup tables occurring in a table-based white-box AES implementation can be categorized into the following two classes: *T-boxes* and *non-T-boxes*. Below, it is explained how variable encodings can be applied to both classes of lookup tables, where it is assumed that all lookup tables map 8 bits to 8 bits.

Variable encodings of non-T-boxes. Let x and y denote the input and output of a non-T-box nT, respectively. Then both x and y can be variably encoded by specifying a set of secret white-box encodings defined as

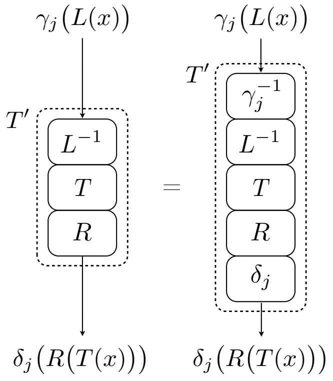
$$\Lambda = \{\lambda_j \mid 1 \leq j \leq n, \tau_j \text{ is a secret bijective affine mapping on } \mathbb{F}_2^8\} \quad \text{and}$$

$$\Sigma = \{\sigma_j \mid 1 \leq j \leq n, \mu_i \text{ is a secret bijective affine mapping on } \mathbb{F}_2^8\} \quad ,$$

associated with the input and output, respectively. Next, select an element indexed by j out of both sets and apply it to both x and y in order to obtain the variably encoded input $\lambda_j(x)$ and variably encoded output $\sigma_j(y)$. By making the index j dependent on intermediate results of AES, one obtains variable white-box encodings that are very likely to change between different executions



(a) Variable encodings of non-T-boxes.



(b) Variable encodings of T-boxes based on the affine self-equivalences.

Figure 7.1: Variable encodings as a countermeasure to avoid fixed encodings.

of the white-box implementation with different input data. For details about the handling of the index j of the variable encodings, refer to Michiels and Gorissen [74].

Now, a variably encoded non-T-box, denoted by nT', is defined as

$$nT' : \mathbb{F}_2^8 \times \{1, 2, \ldots, n\} \to \mathbb{F}_2^8 : nT'(z, j) = \sigma_j \circ nT \circ \lambda_j^{-1}(z) \enspace ,$$

where $\lambda_j \in \Lambda$ and $\sigma_j \in \Sigma$. The lookup table implementing such a variably encoded non-T-box takes as input the variably encoded input byte $\lambda_j(x)$ and the index j of the variable encoding λ_j applied to x, and outputs the variably encoded output byte $\sigma_j(y)$ (Fig. 7.1a). The index j needs to be given explicitly as input such that the corresponding variable input encoding can be annihilated and the corresponding variable output encoding can be applied. Therefore, for practical reasons concerning the storage requirement of lookup tables, the index j is typically kept small, e.g., 4 bits.

Variable encodings of T-boxes. In the case of T-boxes, the application of variable encodings is based on the affine self-equivalences of the AES S-box, as is explained in the following. Recall that a T-box T was defined as $T(x) = S(x \oplus k)$. Next, the input and output of the T-box are assumed to be encoded by secret 8-bit bijective affine mappings L and R, respectively, in order to prevent an attacker from extracting the embedded secret key information. This results in the *encoded T-box* T', defined as $T' = R \circ T \circ L^{-1}$ and implemented as a lookup table mapping 8 bits to 8 bits.

In [14], Biryukov et al. showed that the 8-bit bijective AES S-box S has exactly 2040 different affine self-equivalences; each affine self-equivalence is defined as a pair (α, β) of bijective affine mappings on $\mathbb{F}_2^8$ such that $S = \beta \circ S \circ \alpha^{-1}$. Now, based on the set of affine self-equivalences of S, given by $\Phi_S = \{(\alpha, \beta) \mid \beta \circ S \circ \alpha^{-1} = S\}$ with $|\Phi_S| = 2040$, a set of affine self-equivalences of T' can be derived, given by

$$\Phi_{T'} = \{(\gamma, \delta) = (L \circ \oplus_k \circ \alpha \circ \oplus_k \circ L^{-1}, R \circ \beta \circ R^{-1}) \mid (\alpha, \beta) \in \Phi_S\} \tag{7.1}$$

with $|\Phi_{T'}| = |\Phi_S| = 2040$, such that $\delta \circ T' \circ \gamma^{-1} = T'$, i.e.,

$$\begin{aligned}\delta \circ T' \circ \gamma^{-1} &= \delta \circ R \circ T \circ L^{-1} \circ \gamma^{-1} = \delta \circ R \circ S \circ \oplus_k \circ L^{-1} \circ \gamma^{-1} \\ &= R \circ \beta \circ R^{-1} \circ R \circ S \circ \oplus_k \circ L^{-1} \circ L \circ \oplus_k \circ \alpha^{-1} \circ \oplus_k \circ L^{-1} \\ &= R \circ \beta \circ S \circ \alpha^{-1} \circ \oplus_k \circ L^{-1} = R \circ S \circ \oplus_k \circ L^{-1} = T' \ .\end{aligned}$$

In [74], Michiels and Gorissen explain how this set $\Phi_{T'}$ of affine self-equivalences of T' can be used to apply variable encodings to the input and output of T'. Observe that the output of T' equals $\delta_j\big(R\big(T(x)\big)\big)$ if the input equals $\gamma_j\big(L(x)\big)$ for any affine self-equivalence pair $(\gamma_j, \delta_j) \in \Phi_{T'}$, i.e.,

$$T'\big(\gamma_j\big(L(x)\big)\big) = \delta_j\big(T'\big(\gamma_j^{-1}\big(\gamma_j\big(L(x)\big)\big)\big)\big) = \delta_j\big(T'\big(L(x)\big)\big) = \delta_j\big(R\big(T(x)\big)\big) \ ,$$

although the variable encodings γ_j and δ_j are not explicitly implemented at the input and output of T', respectively (Fig. 7.1b). As a result, a variable encoding can be applied to the input and output of T' using only one instance of T', i.e., it is not required to give the index j as input to the lookup table implementing T', which is in contrast to the lookup tables implementing the variably encoded non-T-boxes (Fig. 7.1). Hence, an encoded T-box also corresponds with a variably encoded T-box since the variable encodings (i.e., the affine self-equivalences) are handled implicitly through the encoded T-box. Since $|\Phi_{T'}| = 2040$, a total of 2040 different variable encodings can be applied. And, as explained above in the case of non-T-boxes, by making the index j dependent on intermediate results of AES, the applied variable encodings are very likely to change during different executions of the white-box AES implementation with different inputs.

Finally, as a comparison with solely fixed encodings, suppose that the input x to a T-box T is identical in two different executions of the same white-box implementation. Then in the scenario of fixed encodings, the encoded input $L(x)$ of the encoded T-box T' is identical as well. However, in the scenario of variable encodings, the encoded input $\gamma_j\big(L(x)\big)$ of T' is likely to be different since the index j depends on intermediate results which may have been different. In [74], Michiels and Gorissen explain how the use of variable encodings may prevent the BGE attack [13] and Michiels et al.'s attack [75], and hence how it may result in a secure white-box AES implementation.

7.2.2 All white-box implementations are fixed-key. What about dynamic-key?

All white-box AES implementations discussed so far are fixed-key. As Shamir and van Someren already pointed out back in 1999 [97], fixed-key variants are most likely to result in more robust solutions. In that scenario, the key can be handled as a constant, and can be baked inside the implementations in a much more efficient and optimized way. But what about dynamic-key implementations? Some applications in which white-box cryptography is deployed require the ability to update the cryptographic key by means of a *key-update message*. The naive approach would be to update the entire white-box implementation; that is, replace it with a new implementation instantiated with the updated key. The semi-naive approach would be to update only the key-dependent part (e.g., replacing the key-dependent lookup tables) of the white-box implementation (cf. Gorissen et al. [46]). However, in both approaches, the size of the key-update message (referred to as the *key-update size* in the following) is rather large; for example, in the case of Chow et al.'s white-box AES implementation, the key-update size is given by 752 kB (naive approach) or 208 kB (semi-naive). For some applications (e.g., pay-TV systems with many end-users), the distribution of a large key-update message is impractical due to restrictions on the available bandwidth. Hence, for those applications, it is the primary goal to keep the key-update size as small as possible.

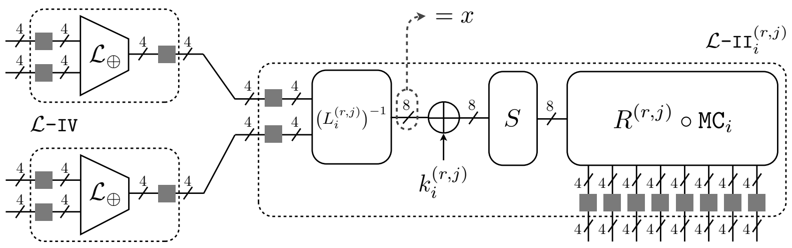
Optimal key-update size. So far, the white-box implementations of block ciphers published in the academic literature lack the key scheduling algorithm. As a result, in order to update the secret key of a white-box implementation, it is required to provide the *expanded key* comprising the set of all round keys. Therefore, the smallest key-update size is given by the size of the expanded key and is referred to as the *optimal key-update size*, where the term 'optimal' is used with respect to the lack of the key scheduling algorithm. For example, in the case of AES-128, AES-192 or AES-256, the optimal key-update sizes are 176 bytes, 208 bytes or 240 bytes, respectively.

Dynamic-Key White-Box Schemes with Optimal Key-Update Size

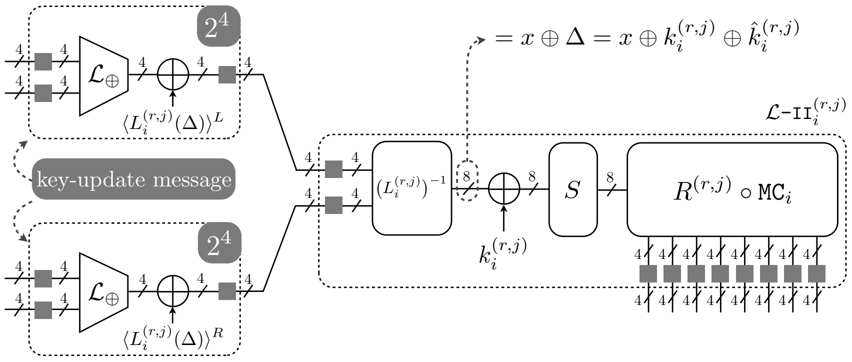
In [71, 72], Michiels proposed two schemes for dynamic-key white-box implementations with optimal key-update size are proposed. In both schemes, it is assumed that the standard dynamic-key white-box implementation is instantiated with the secret key k. In order to update the key, the offset Δ (or difference) between the original key k and the updated key $\hat{k}$, i.e., $\Delta = k \oplus \hat{k}$, needs to be given as input. Both schemes introduce this offset in different ways. In scheme A [71], the dynamic-key white-box implementation already includes all possible offsets such that the key-update message (a *selection string*) selects the correct offset by specifying the correct data-flow. The relation between the selection string and the selected offset may be any fixed secret bijective encoding. In scheme B [72], the key-update message comprises the encoded version of the key-offset, where the applied encoding is secret and fixed. For detailed information, refer to Michiels [71, 72]. Observe that it is also possible to make the standard dynamic-key white-box implementation key-less such that the key-update messages do not comprise key-offsets but rather comprise the actual (updated) key.

Figure 7.2 depicts the application of both schemes to the white-box AES implementation of Chow et al. [23] in order to obtain a dynamic-key version. Recall that the Type II and Ib tables are key-dependent. The examples in Fig. 7.2 focus on updating the round key bytes $k_i^{(r,j)}$ contained within the Type II tables $\mathcal{L}\text{-}\mathtt{II}_i^{(r,j)}$ $(0 \leq i, j \leq 3$ and $1 \leq r \leq 9)$. In scheme A (Fig. 7.2b), each round key byte offset is considered as the concatenation of its left and right nibble. For each nibble offset, all possible 2^4 values are included in a set of 2^4 *encoded nibble offset XOR tables* preceding the $\mathcal{L}\text{-}\mathtt{II}_i^{(r,j)}$ lookup table. For each set, the key-update message contains a 4-bit index selecting one of the 2^4 tables corresponding with the correct nibble offset. In scheme B (Fig. 7.2c), the key-update message comprises a fixed encoded byte offset for each round key byte $k_i^{(r,j)}$; the encoded offset is then XOR'ed with the fixed encoded input of the $\mathcal{L}\text{-}\mathtt{II}_i^{(r,j)}$ table. Each 8-bit XOR is implemented as the parallel execution of two encoded nibble XOR tables. For both schemes, observe that the key-update size equals the size of the expanded key, and hence is optimal.

Disadvantage. A disadvantage of both schemes discussed above is that the offset associated with each round key byte of the expanded key is encoded in a fixed way. Hence, the encodings remain invariant for different key updates such that the attacker can identify which key-offsets of specific round key bytes are preserved. Since each AES expanded key has an average number of 208 round key bytes and since each round key byte can take only 256 different values, the probability that at least one round key byte is preserved during a key update is given by $1 - (1 - 1/2^8)^{208} \approx 56\%$ and thus is high. The ability of identifying which round key bytes remained identical when performing a key update is an undesirable property since it reduces the entropy of the updated key if the attacker has compromised the AES expanded key that needed to be updated. To overcome this disadvantage, we presented a new dynamic-key white-box scheme [91] with a close to optimal key-update size based on variable encodings.

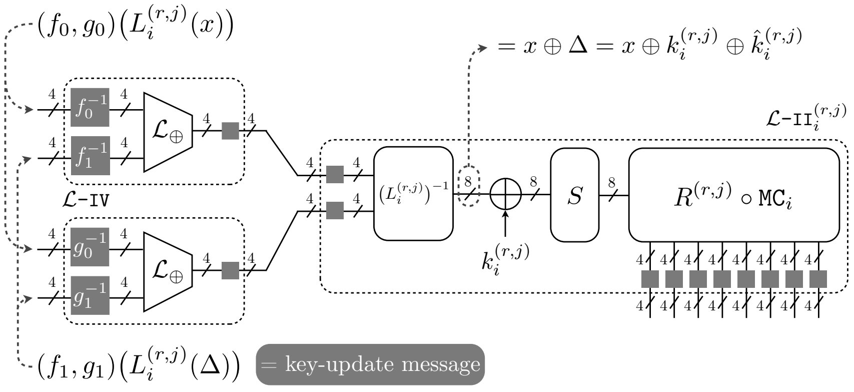


(a) Relevant part of the fixed-key white-box AES implementation of Chow et al. for updating the round key bytes of the type II tables.



2^4
$= x \oplus \Delta = x \oplus k_i^{(r,j)} \oplus \hat{k}_i^{(r,j)}$
$\mathcal{L}\text{-II}_i^{(r,j)}$
$\mathcal{L}_\oplus$
$\langle L_i^{(r,j)}(\Delta)\rangle^L$
key-update message
$\langle L_i^{(r,j)}(\Delta)\rangle^R$
$(L_i^{(r,j)})^{-1}$
S
$R^{(r,j)} \circ \texttt{MC}_i$
$k_i^{(r,j)}$

(b) Relevant part of the dynamic-key white-box AES implementation of Chow et al. for updating the round key bytes of the type II tables based on scheme A.



$(f_0, g_0)\big(L_i^{(r,j)}(x)\big)$
$= x \oplus \Delta = x \oplus k_i^{(r,j)} \oplus \hat{k}_i^{(r,j)}$
$\mathcal{L}\text{-II}_i^{(r,j)}$
f_0^{-1}
f_1^{-1}
g_0^{-1}
g_1^{-1}
$\mathcal{L}_\oplus$
$\mathcal{L}\text{-IV}$
$(L_i^{(r,j)})^{-1}$
S
$R^{(r,j)} \circ \texttt{MC}_i$
$k_i^{(r,j)}$
$(f_1, g_1)\big(L_i^{(r,j)}(\Delta)\big)$
= key-update message

(c) Relevant part of the dynamic-key white-box AES implementation of Chow et al. for updating the round key bytes of the type II tables based on scheme B.

Figure 7.2: Two dynamic-key white-box schemes with optimal key-update size applied to the white-box AES implementation of Chow et al.

Dynamic-Key White-Box Scheme based on Variable Encodings

In our solution [91], a set of secret white-box encodings is assigned to each round key byte of the expanded key. When updating the key, and hence computing the offset for each round key byte, an encoding is chosen out of the sets associated with each round key byte and the selected encoding is applied to the corresponding round key byte offset. If an offset remains identical for different key updates, different encodings need to be selected out of the set associated with that particular offset. This way, the attacker is unable to distinguish identical round key bytes between different updates. In this new dynamic-key white-box scheme, the key-update message comprises the variably encoded key-offsets and the corresponding indices of the applied variable encodings. For dynamic-key implementations of AES-128, AES-192 or AES-256 using 4-bit indices (i.e., sets of 2^4 secret encodings are associated with each round key byte), the key-update size becomes 264 bytes, 312 bytes or 360 bytes, respectively, which is close to optimal. Further, in order to provide sufficient security and not to undermine the effect of the variable encodings, our new solution assumes a white-box implementation in which all intermediate results are variably encoded as well, either by user-defined sets of encodings (for non T-boxes) or by the affine self-equivalences of a T-box (as explained in Sect. 7.2.1). For a complete description of our solution, refer to us [91].

Figure 7.3 depicts an example of the application of our new key updating scheme to a white-box AES implementation. The table T' represents an encoded T-box defined as $T' = R \circ T \circ L^{-1}$ with $T = S \circ \oplus_k$, where S and k denote the AES S-box and a round key byte, respectively; the function F computes the XOR of an intermediate data byte x and a round key byte offset $\Delta = k \oplus \hat{k}$ in an encoded way, where $\hat{k}$ denotes the updated round key byte. The four inputs to F are (i) the variably encoded key-offset $\mu_i(\Delta)$, (ii) the index i of the variable encoding μ_i applied to Δ, (iii) the variably encoded data byte $\tau_j(x)$, and (iv) the index j of the variable encoding τ_j applied to x. The variable encodings μ_i and τ_j are selected from the sets

$$\{\mu_i \mid 1 \leq i \leq n, \mu_i \text{ is a secret bijective affine mapping on } \mathbb{F}_2^8\} \quad \text{and}$$

$$\{\tau_j \mid 1 \leq j \leq n, \tau_j \text{ is a secret bijective affine mapping on } \mathbb{F}_2^8\} \quad ,$$

respectively. For practical reasons (storage restrictions), the cardinality n of the sets is typically set to 2^4 such that i and j are 4-bit indices. The two outputs of F are (i) the variably encoded input $\gamma_l\big(L(x \oplus \Delta)\big)$ to T', and (ii) the index l of the applied variable encoding γ_l. Recall from Sect. 7.2.1 that the pair (γ, δ) denotes the variable encodings of T' based on its affine self-equivalences (see (7.1)). The index l depends on both indices i and j, i.e., $l = g(i, j)$ for some secret function g. For details about the implementation of F, refer to us [91].

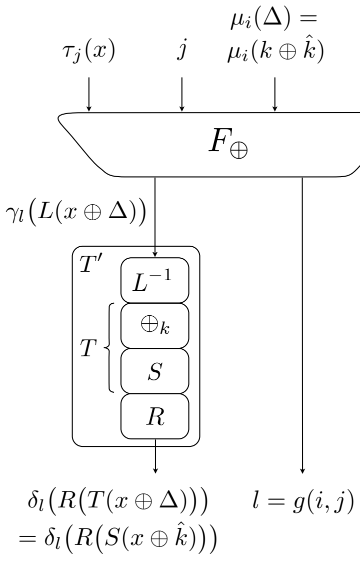

$\tau_j(x)$
j
$\mu_i(\Delta) = \mu_i(k \oplus \hat{k})$
i
$F_\oplus$
$\gamma_l\big(L(x \oplus \Delta)\big)$
T'
L^{-1}
T
$\oplus_k$
S
R
$\delta_l\big(R\big(T(x \oplus \Delta)\big)\big) = \delta_l\big(R\big(S(x \oplus \hat{k})\big)\big)$
$l = g(i, j)$

Figure 7.3: Dynamic-key white-box scheme based on variable encodings [91].

Possible Threat to Dynamic-Key White-Box Implementations

Recall that in fixed-key white-box implementations, the secret round keys are treated as constants and hence can be included in the white-box implementation in an optimized way. However, in dynamic-key white-box implementations, the secret round keys become variables and are most likely harder to hide. Furthermore, an additional interface (i.e., the input for the key-update message) is required to update the embedded secret key. In the white-box attack context, this interface is assumed to be accessible by the attacker who might try to exploit (partially) known-key attacks as explained in the following. Take for example the case where the same fixed encoding, or the same set of variable encodings, is used for all round key bytes. In that particular case, the attacker can for example modify the key-update message (given as input to the dynamic-key white-box implementation) in such a way that all round key bytes of the expanded key are identical, though still unknown because of the secret encoding. In this way, the attacker can reduce the entropy of the key of the white-box implementation to only 8 bits, which may lead to efficient white-box attacks extracting the single round key byte. By repeating the above technique for all bytes of one round key, the attacker can retrieve the secret key.

Also, if the white-box implementation has already been broken, then performing a key update will typically not provide a solution to correct the security breach since the attacker is most likely able to compromise the updated key as well. As an example, take both schemes depicted in Fig. 7.2 when applied to the white-box AES implementation of Chow et al.; both dynamic-key schemes remain vulnerable to the BGE attack.

7.3 Conclusion

This chapter presented an overview of the state-of-the-art of lookup-table-based white-box AES implementations published in the academic literature. By comparing the size and performance of AES implementations in the black-box and white-box settings, it is clear that the cost of white-box security typically corresponds with an increased implementation size and a performance slowdown. An exception to the rule of performance slowdown is given by the Xiao-Lai white-box AES implementation. Depending on the requirements of a given application, one takes the most secure white-box implementation satisfying the requirements. Over time, the increased implementation size of white-box implementations will become less of a problem since the available memory only increases. Nevertheless, in this chapter, we also presented some methods to reduce the white-box cost; take for example the reuse of lookup tables, although such reuse may weaken the white-box implementation since it is associated with the reuse of secret white-box encodings.

Despite the cost (i.e., large implementation size and performance slowdown) introduced by the various white-box techniques, the cryptanalytic results on all table-based white-box AES implementations published in the academic literature showed that none of them can be considered as white-box secure. In fact, most attacks have a practical work factor. The *best* white-box AES implementation according to the results obtained in this thesis is the variant of the Xiao-Lai white-box AES implementation with affine encodings instead of linear encodings. Here, the term 'best' refers to both the achieved level of white-box security as well as the performance of the implementation.

With respect to designing new secure white-box AES implementations, some indications were highlighted; the vulnerabilities of the existing implementations exploited by the currently known white-box attacks have been identified and possible countermeasures to avoid these vulnerabilities have been discussed. One promising technique is the use of so-called variable white-box encodings; these encodings are no longer fixed and vary depending on intermediate results of AES such that different executions of the white-box AES implementation with different input data results in different encodings.

Further, this chapter presented various techniques to obtain dynamic-key white-box implementations. In particular, two schemes based on fixed encodings with an optimal key-update size (i.e., equal to the size of the expanded key in the case of the lack of the key scheduling algorithm) are described. However, the use of fixed encodings allows the attacker to distinguish identical round key bytes between different key updates. Therefore, we presented a new dynamic-key scheme based on variable encodings with an almost optimal key-update size.

Part III

Conclusion

Chapter 8

Conclusions and Future Research

This doctoral thesis dealt with the design and analysis of white-box AES implementations. With regard to the analysis part, our main research contributions were covered by Chapters 4, 5 and 6. With regard to the design part, our contribution is presented in Chapter 7 (as part of Sect. 7.2.2).

8.1 Summary of Results and Conclusions

Chapters 1 and 2 showed that the attack models have evolved over time due to increasing demand for strong cryptographic algorithms implemented in software and executed on open platforms such as a PC, smartphone, tablet or set-top box. Since these platforms are owned and controlled by a possibly malicious party, there was a need for a new attack model as the conventional black-box model and even the grey-box model were no longer satisfied. In 2002, Chow et al. introduced the *white-box model* in which an attacker is assumed to have full access to the software implementation of a cryptographic algorithm (such as a block cipher) as well as full control over its execution environment. Chapter 1 showed that the white-box model is not only the strongest attack model (from the perspective of the attacker), but furthermore is also very realistic since it occurs in real-world applications such as Digital Rights Management. Further, Chapter 2 highlighted that widely used block ciphers that were designed for security in the black-box model are highly insecure in the white-box model. That is, strong black-box security properties does not prevent a white-box attacker from exploiting weak implementation properties. This was demonstrated by the fairly simple *entropy attack* (of Shamir and van Someren [97]) and the *S-box blanking attack* (of Kerins and Kursawe [55]) with regard to the black-box software implementations of block ciphers deployed in the white-box model.

Main conclusion for Part I (Chapters 1 and 2). *Nowadays, it is insufficient to solely focus on the black-box security of block ciphers. Secure implementations (either in hardware or software) of block ciphers are at least as important, and a secure black-box block cipher does not guarantee a secure implementation. Hence there is a need for white-box cryptography that provides techniques to construct secure software implementations of cryptographic algorithms specifically tailored to the white-box model.*

White-box cryptography was introduced in Chapter 3, the main part of which was dedicated to the design and analysis of the first published white-box AES implementation of Chow et al. (published in 2002 [23]). The negative result of Billet et al. [13], efficiently breaking this first published white-box AES implementation, triggered research to design new secure white-box AES implementations. The following three proposals appeared in the academic literature: the construction of Bringer et al. in 2006 [20] based on a new white-box technique (i.e., different from the technique of Chow et al.), of Xiao and Lai in 2009 [107] based on applying the white-box technique of Chow et al. in a specific way, and of Karroumi in 2010 [53] based on the application of dual AES ciphers. For all three proposals, their designers claimed that they were resistant against Billet et al.'s attack.

However, in this thesis, we showed that none of the above newly proposed white-box AES implementations provides a sufficient level of white-box security. We have done this by either presenting a new efficient attack, or by showing that the implementation remained vulnerable to Billet et al.'s attack.

In Chapter 4 we showed that Karroumi's white-box AES implementation belongs to the class of white-box AES implementations as originally specified by Chow et al. Consequently, Karroumi's implementation remains vulnerable to Billet et al.'s attack. With regard to Billet et al.'s attack itself, we presented several improvements in addition to the initial improvement proposed by Tolhuizen in 2012 [101]. The combination of the improvements significantly reduced the overall work factor of the attack. An interesting remark is that unlike the original attack of Billet et al., the use of non-affine white-box encodings and the (secret) randomization in the order of the bytes of the intermediate AES results in the white-box implementation have a negligible impact on the overall work factor of our improved version of the attack.

In Chapter 5, we presented a practical attack on the Xiao-Lai white-box AES implementation. The attack exploits specific properties of AES as well as leaked information (from the white-box implementation) about the secret encodings. The attack uses a modified variant of the linear equivalence algorithm proposed by Biryukov et al. [14] as a building block. Additionally, we considered design generalizations of the Xiao-Lai white-box AES implementation and investigated their impact on our attack. We showed that the Xiao-Lai white-box AES implementation with affine (instead of linear) encodings provides the highest level of white-box security with respect to all white-box AES implementations considered in this thesis.

In Chapter 6 we presented an efficient attack on the white-box AES implementation of Bringer et al. Instead of the original secret key, our attack extracts an *equivalent* key, i.e., both the original and the equivalent key yield functionally equivalent implementations.

In Chapter 7, all white-box AES implementations are compared with the conventional black-box software AES implementation (i.e., the lookup-table-based AES implementation proposed by Daemen and Rijmen [31] for processors with word lengths of 32-bits or more) with respect to their size and performance. This comparison showed that the cost of white-box security typically corresponds with an increased implementation size and a performance slowdown. However, the latter was less applicable to the Xiao-Lai white-box AES implementation with either linear or affine white-box encodings. We also presented some techniques to reduce the size and performance costs. They should be applied with caution as they may weaken the white-box implementation.

Further, Chapter 7 also focused on the design aspects of white-box AES implementations. First, a promising technique, introduced by Michiels and Gorissen [74], is described. This technique is based on the application of so-called variable white-box encodings, i.e., encodings that vary between different executions of the white-box implementation with different input data. Second, several techniques were discussed for dynamic-key white-box implementations. In particular, we proposed a new dynamic-key white-box scheme with a small key-update size based on variable encodings. In our solution, it is hard for the attacker to distinguish identical round key bytes between different key updates.

Main conclusions for Part II (Chapters 3 to 7). *We have shown that in early 2014, despite the typical 'significant' white-box cost, there does not exist a practical and secure white-box AES implementation published in the academic literature, even if AES is still considered to be a secure black-box block cipher. The best (i.e., providing the best white-box security and having the best performance) implementation according to our results obtained in Chapter 5* *is the Xiao-Lai white-box AES implementation with affine encodings. This naturally leads to the following two core questions within white-box cryptography:*

> *1) – Do we need new 'white-box friendly' block ciphers, i.e., design new block ciphers with the white-box model in mind?*

AES was designed with black-box security in mind. Michiels et al. [75] showed that some design principles that offer good protection against black-box attacks (e.g., the MDS property of the diffusion operator) can introduce weak spots for white-box attacks. Hence, it might be interesting to define new 'white-box friendly' design principles of block ciphers. As the white-box model includes the black-box model, these new design principles will also lead to secure black-box ciphers. Preliminary thoughts on new white-box design principles of block ciphers are listed in Sect. 8.2. However, the freedom to use a new block cipher depends on compatibility requirements with existing block ciphers at either the remote or client side.

First, if compatibility is not required, then one has the freedom to design a new block cipher with white-box security in mind. Moreover, if one is not required to rely on existing block ciphers, why still stick to block ciphers? As we discussed in Chapter 3, typically an encoded version of a block cipher was implemented. Hence, if the white-box security relies on the application of external encodings, what is then the added value of the block cipher itself? In the fixed-key scenario, why not just choose randomly a permutation on n bits that can be efficiently implemented and keep this permutation secret?

Second, if compatibility is required, then the following question becomes relevant.

> *2) – Do we need new white-box techniques to construct secure white-box implementations of existing secure black-box block ciphers?*

As mentioned above, if one is limited to using existing block ciphers, then either the existing white-box techniques need to be revised, or new white-box techniques need to be developed. The revision of the existing techniques is necessary since, even though their current applications failed in achieving secure white-box implementations (e.g., Chow et al.'s, Xiao-Lai and Bringer et al.'s white-box AES implementations), they might still provide a sufficient level of white-box security if applied in a specific way. In the case of Chow et al.'s technique, examples are to build larger lookup tables by merging more steps of the round function or to use only affine encodings since Tolhuizen's method showed that the use of non-affine encodings typically has a negligible impact on the overall work factor of a white-box attack. In line with this, we showed that Chow et al.'s technique still provides some white-box security for the Xiao-Lai white-box AES implementation with affine encodings (see above). With regard to developing new techniques, as mentioned before, a promising new white-box technique based on variable encodings has already been presented by Michiels and Gorissen in a patent application.

All white-box techniques published in the academic literature are solely focused on constructing fixed-key white-box implementations. However, there are many applications in the real world that may benefit from the ability to update the key. E.g., if the key of an application can be expected to leak from another part (i.e., not the white-box implementation) of the application, then a white-box implementation that allows to update the key can be used to resolve the security breach. Hence, when developing new white-box techniques, the primary focus should be on designing dynamic-key white-box implementations straightaway. But since dynamic-key will never be more secure than fixed-key, i.e., a fixed-key implementation can be derived from a dynamic-key implementation by fixing the key, the development of dynamic-key white-box techniques can be considered as a great challenge.

Further, it might also be beneficial to develop new white-box techniques that are capable of securely implementing existing block ciphers without the application of external encodings. This way, the designer is free to choose whether to include external encodings (either at the input or at the output or at both).

Main conclusion of this doctoral thesis. *It is clear that in early 2014, all white-box AES implementations published in the academic literature have been broken. Still, there are many proprietary white-box AES implementations offered by companies specialized in white-box cryptography (e.g., Irdeto [50], Nagra [80], whiteCryption [26], SafeNet [92], ...). However, as there is still no (published) breakthrough with regard to secure white-box techniques, the field of white-box cryptography remains in its infancy and there is still a lot to cover.*

8.2 Future Work

In this section, we suggest some directions for future research.

Investigate the white-box security of the Xiao-Lai white-box AES implementation with affine encodings. According to our results obtained in Chapter 5, the Xiao-Lai white-box AES implementation with affine (instead of linear) encodings offers not only good performance but also the highest level of white-box security with respect to all white-box AES implementations considered in this thesis. Naively applying the generic attack of Michiels et al. showed an estimated work factor of at least 2^{49}. However, since the attack is generic, it might be interesting to investigate to what extent the exploitation of specific properties of AES and the composition of the white-box implementation may affect its work factor. In other words, does there exist a non-generic attack with a reduced work factor? Additionally, since Michiels et al.'s attack assumes that the order of the bytes of the intermediate AES results in the Xiao-Lai implementation is known to the attacker, it might be interesting to investigate how a (secret) randomization of this order, which can be implemented 'for free', affects the work factor of the attack.

Develop secure white-box design principles of block ciphers and novel white-box techniques. As mentioned before, in order to obtain positive results within white-box cryptography, it might be interesting to develop either new 'white-box friendly' design principles of block ciphers or novel dynamic-key white-box techniques resulting in secure white-box implementations of existing or new block ciphers. For our discussion on this, refer to Sect. 8.1.

Based on the specific block cipher design properties exploited by the white-box attacks discussed in this thesis, we list the following suggestions towards new design principles that may preclude these attacks:

1. independent round keys or a non-invertible key scheduling algorithm: this prevents the attacker from recovering the secret key if he can extract a round key;
2. make more parts of the block cipher key-dependent, which can for example either be derived from the secret key or be specified separately: this prevents the attacker from exploiting specific properties of for example the diffusion operator;
3. use 'wide' non-sparse diffusion operators, where 'wide' refers to the block size of the cipher in order to increase the diffusion created in single round functions.

It should be noted however, that introducing such (and other) new design principles may interfere with the black-box security properties of the resulting block-cipher.

Part IV

Bibliography

Bibliography

[1] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. P. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. In J. Kilian, editor, *CRYPTO*, volume 2139 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2001.

[2] E. Barkan and E. Biham. The book of Rijndaels. *IACR Cryptology ePrint Archive*, 2002:158, 2002.

[3] E. Barkan and E. Biham. In how many ways can you write Rijndael? In Y. Zheng, editor, *ASIACRYPT*, volume 2501 of *Lecture Notes in Computer Science*, pages 160–175. Springer, 2002.

[4] BBC News. Apple to end music restrictions, 2009. `http://news.bbc.co.uk/2/hi/technology/7813527.stm`.

[5] R. Benadjila, O. Billet, and S. Francfort. DRM to counter side-channel attacks? In M. Yung, A. Kiayias, and A.-R. Sadeghi, editors, *Digital Rights Management Workshop*, pages 23–32. ACM, 2007.

[6] D. J. Bernstein. Cache-timing attacks on AES. Preprint, 2005. `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`.

[7] E. Biham, R. J. Anderson, and L. R. Knudsen. Serpent: A new block cipher proposal. In S. Vaudenay, editor, *FSE*, volume 1372 of *Lecture Notes in Computer Science*, pages 222–238. Springer, 1998.

[8] E. Biham and A. Shamir. Differential cryptanalysis of DES-like cryptosystems. In A. Menezes and S. A. Vanstone, editors, *CRYPTO*, volume 537 of *Lecture Notes in Computer Science*, pages 2–21. Springer, 1990.

[9] E. Biham and A. Shamir. Differential cryptanalysis of the full 16-round DES. In E. F. Brickell, editor, *CRYPTO*, volume 740 of *Lecture Notes in Computer Science*, pages 487–496. Springer, 1992.

[10] E. Biham and A. Shamir. Differential fault analysis of secret key cryptosystems. In B. S. J. Kaliski, editor, *CRYPTO*, volume 1294 of *Lecture Notes in Computer Science*, pages 513–525. Springer, 1997.

[11] E. Biham and A. Shamir. Power analysis of the key scheduling of the AES candidates. In *Proceedings of the Second Advanced Encryption Standard (AES) Candidate Conference*, pages 115–121, 1999.

[12] O. Billet and H. Gilbert. A traceable block cipher. In C.-S. Laih, editor, *ASIACRYPT*, volume 2894 of *Lecture Notes in Computer Science*, pages 331–346. Springer, 2003.

[13] O. Billet, H. Gilbert, and C. Ech-Chatbi. Cryptanalysis of a white box AES implementation. In H. Handschuh and M. A. Hasan, editors, *Selected Areas in Cryptography*, volume 3357 of *Lecture Notes in Computer Science*, pages 227–240. Springer, 2004.

[14] A. Biryukov, C. De Cannière, A. Braeken, and B. Preneel. A toolbox for cryptanalysis: Linear and affine equivalence algorithms. In E. Biham, editor, *EUROCRYPT*, volume 2656 of *Lecture Notes in Computer Science*, pages 33–50. Springer, 2003.

[15] A. Biryukov and A. Shamir. Structural cryptanalysis of SASAS. *J. Cryptology*, 23(4):505–518, 2010.

[16] J. Black. The ideal-cipher model, revisited: An uninstantiable blockcipher-based hash function. In M. J. B. Robshaw, editor, *FSE*, volume 4047 of *Lecture Notes in Computer Science*, pages 328–340. Springer, 2006.

[17] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe. PRESENT: An ultra-lightweight block cipher. In P. Paillier and I. Verbauwhede, editors, *CHES*, volume 4727 of *Lecture Notes in Computer Science*, pages 450–466. Springer, 2007.

[18] D. Boneh, R. A. DeMillo, and R. J. Lipton. On the importance of eliminating errors in cryptographic computations. *J. Cryptology*, 14(2):101–119, 2001.

[19] J. Bringer, H. Chabanne, and E. Dottax. Perturbing and protecting a traceable block cipher. In H. Leitold and E. P. Markatos, editors, *Communications and Multimedia Security*, volume 4237 of *Lecture Notes in Computer Science*, pages 109–119. Springer, 2006.

[20] J. Bringer, H. Chabanne, and E. Dottax. White box cryptography: Another attempt. Cryptology ePrint Archive, Report 2006/468, 2006. `http://eprint.iacr.org/2006/468.pdf`.

[21] S. Chari, C. Jutla, J. R. Rao, and P. Rohatgi. A cautionary note regarding evaluation of AES candidates on smart-cards. In *Proceedings of the Second Advanced Encryption Standard (AES) Candidate Conference*, pages 133–147, 1999.

[22] S. Chari, J. R. Rao, and P. Rohatgi. Template attacks. In B. S. J. Kaliski, Ç. K. Koç, and C. Paar, editors, *CHES*, volume 2523 of *Lecture Notes in Computer Science*, pages 13–28. Springer, 2002.

[23] S. Chow, P. A. Eisen, H. Johnson, and P. C. van Oorschot. White-box cryptography and an AES implementation. In K. Nyberg and H. M. Heys, editors, *Selected Areas in Cryptography*, volume 2595 of *Lecture Notes in Computer Science*, pages 250–270. Springer, 2002.

[24] S. Chow, P. A. Eisen, H. Johnson, and P. C. van Oorschot. A white-box DES implementation for DRM applications. In J. Feigenbaum, editor, *Digital Rights Management Workshop*, volume 2696 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2002.

[25] N. Courtois and J. Pieprzyk. Cryptanalysis of block ciphers with overdefined systems of equations. In Y. Zheng, editor, *ASIACRYPT*, volume 2501 of *Lecture Notes in Computer Science*, pages 267–287. Springer, 2002.

[26] Cryptanium Inc. Homepage (whiteCryption). `https://www.cryptanium.com/`.

[27] J. Daemen, L. R. Knudsen, and V. Rijmen. The block cipher Square. In E. Biham, editor, *FSE*, volume 1267 of *Lecture Notes in Computer Science*, pages 149–165. Springer, 1997.

[28] J. Daemen and V. Rijmen. AES proposal: Rijndael. In *First Advanced Encryption Standard (AES) Conference*, 1998.

[29] J. Daemen and V. Rijmen. Resistance against implementation attacks: A comparative study of the AES proposals. In *Proceedings of the Second Advanced Encryption Standard (AES) Candidate Conference*, pages 122–132, 1999.

[30] J. Daemen and V. Rijmen. The wide trail design strategy. In B. Honary, editor, *IMA Int. Conf.*, volume 2260 of *Lecture Notes in Computer Science*, pages 222–238. Springer, 2001.

[31] J. Daemen and V. Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard.* Information Security and Cryptography. Springer, 2002.

[32] J. Daemen and V. Rijmen. Plateau characteristics. *IET Information Security*, 1(1):11–17, 2007.

[33] C. De Cannière. *Analysis and Design of Symmetric Encryption Algorithms.* PhD thesis, Katholieke Universiteit Leuven, 2007. Bart Preneel (promotor).

[34] Y. De Mulder, G. Danezis, L. Batina, and B. Preneel. Identification via location-profiling in GSM networks. In V. Atluri and M. Winslett, editors, *WPES*, pages 23–32. ACM, 2008.

[35] Y. De Mulder, P. Roelse, and B. Preneel. Cryptanalysis of the Xiao-Lai white-box AES implementation. In L. R. Knudsen and H. Wu, editors, *Selected Areas in Cryptography*, volume 7707 of *Lecture Notes in Computer Science*, pages 34–49. Springer, 2012.

[36] Y. De Mulder, P. Roelse, and B. Preneel. Revisiting the BGE attack on a white-box AES implementation. Cryptology ePrint Archive, Report 2013/450, 2013. `http://eprint.iacr.org/2013/450.pdf`.

[37] Y. De Mulder, K. Wouters, and B. Preneel. A privacy-preserving ID-based group key agreement scheme applied in VPAN. In I. Cerná, T. Gyimóthy, J. Hromkovic, K. G. Jeffery, R. Královic, M. Vukolic, and S. Wolf, editors, *SOFSEM*, volume 6543 of *Lecture Notes in Computer Science*, pages 214–225. Springer, 2011.

[38] Y. De Mulder, B. Wyseur, and B. Preneel. Cryptanalysis of a perturbated white-box AES implementation. In G. Gong and K. C. Gupta, editors, *INDOCRYPT*, volume 6498 of *Lecture Notes in Computer Science*, pages 292–310. Springer, 2010.

[39] C. Delerablée, T. Lepoint, P. Paillier, and M. Rivain. White-box security notions for symmetric encryption schemes. In T. Lange, K. Lauter, and P. Lisonek, editors, *Selected Areas in Cryptography*, Lecture Notes in Computer Science. Springer, 2013.

[40] J. Ding. A new variant of the Matsumoto-Imai cryptosystem through perturbation. In F. Bao, R. H. Deng, and J. Zhou, editors, *Public Key Cryptography*, volume 2947 of *Lecture Notes in Computer Science*, pages 305–318. Springer, 2004.

[41] Electronic Frontier Foundation. *Cracking DES: Secrets of Encryption Research, Wiretap Politics & Chip Design.* O'Reilly Press, 1998.

[42] J.-C. Faugère and L. Perret. Polynomial equivalence problems: Algorithmic and theoretical aspects. In S. Vaudenay, editor,

EUROCRYPT, volume 4004 of *Lecture Notes in Computer Science*, pages 30–47. Springer, 2006.

[43] N. Ferguson, J. Kelsey, S. Lucks, B. Schneier, M. Stay, D. Wagner, and D. Whiting. Improved cryptanalysis of Rijndael. In B. Schneier, editor, *FSE*, volume 1978 of *Lecture Notes in Computer Science*, pages 213–230. Springer, 2000.

[44] N. Ferguson, R. Schroeppel, and D. Whiting. A simple algebraic representation of Rijndael. In S. Vaudenay and A. M. Youssef, editors, *Selected Areas in Cryptography*, volume 2259 of *Lecture Notes in Computer Science*, pages 103–111. Springer, 2001.

[45] K. Gandolfi, C. Mourtel, and F. Olivier. Electromagnetic analysis: Concrete results. In Çetin Kaya Koç, D. Naccache, and C. Paar, editors, *CHES*, volume 2162 of *Lecture Notes in Computer Science*, pages 251–261. Springer, 2001.

[46] P. Gorissen, W. Michiels, and M. Bijsterveld. Updating cryptographic key data. WO 2010142612, 2008.

[47] L. Goubin, J.-M. Masereel, and M. Quisquater. Cryptanalysis of white box DES implementations. In C. M. Adams, A. Miri, and M. J. Wiener, editors, *Selected Areas in Cryptography*, volume 4876 of *Lecture Notes in Computer Science*, pages 278–295. Springer, 2007.

[48] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: Cold boot attacks on encryption keys. In P. C. van Oorschot, editor, *USENIX Security Symposium*, pages 45–60. USENIX Association, 2008.

[49] M. E. Hellman. A cryptanalytic time-memory trade-off. *IEEE Transactions on Information Theory*, 26(4):401–406, 1980.

[50] Irdeto Inc. Homepage. `http://irdeto.com/`.

[51] M. Jacob, D. Boneh, and E. W. Felten. Attacking an obfuscated cipher by injecting faults. In J. Feigenbaum, editor, *Digital Rights Management Workshop*, volume 2696 of *Lecture Notes in Computer Science*, pages 16–31. Springer, 2002.

[52] M. Joye. On white-box cryptography. In A. Elçi, S. B. Örs, and B. Preneel, editors, *Security of Information and Networks (SIN 2007)*, pages 7–12. Trafford Publishing, 2008.

[53] M. Karroumi. Protecting white-box AES with dual ciphers. In K. H. Rhee and D. Nyang, editors, *ICISC*, volume 6829 of *Lecture Notes in Computer Science*, pages 278–291. Springer, 2010.

[54] A. Kerckhoffs. La cryptographie militaire ("military cryptography"). *Journal des Sciences Militaires*, IX:5–38, January 1883.

[55] T. Kerins and K. Kursawe. A cautionary note on weak implementations of block ciphers. In *1st Benelux Workshop on Information and System Security (WISSec 2006)*, page 12, Antwerp, BE, 2006.

[56] D. Klinec. White-box attack resistant cryptography. Master's thesis, Masaryk University, Faculty of Informatics, 2013. Petr Švenda (advisor).

[57] L. R. Knudsen and V. Rijmen. Known-key distinguishers for some block ciphers. In K. Kurosawa, editor, *ASIACRYPT*, volume 4833 of *Lecture Notes in Computer Science*, pages 315–324. Springer, 2007.

[58] L. R. Knudsen and M. J. B. Robshaw. *The block cipher companion.* Information security and cryptography. Springer-Verlag Berlin Heidelberg, 2011.

[59] P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In N. Koblitz, editor, *CRYPTO*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996.

[60] P. C. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In M. J. Wiener, editor, *CRYPTO*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.

[61] X. Lai, J. L. Massey, and S. Murphy. Markov ciphers and differentail cryptanalysis. In D. W. Davies, editor, *EUROCRYPT*, volume 547 of *Lecture Notes in Computer Science*, pages 17–38. Springer, 1991.

[62] T. Lepoint and M. Rivain. Another nail in the coffin of white-box AES implementations. Cryptology ePrint Archive, Report 2013/455, 2013. `http://eprint.iacr.org/2013/455.pdf`.

[63] T. Lepoint, M. Rivain, Y. De Mulder, P. Roelse, and B. Preneel. Two attacks on a white-box AES implementation. In T. Lange, K. Lauter, and P. Lisonek, editors, *Selected Areas in Cryptography*, Lecture Notes in Computer Science. Springer, 2013.

[64] H. E. Link and W. D. Neumann. Clarifying obfuscation: Improving the security of white-box DES. In *Information Technology: Coding and Computing (ITCC 2005), Volume 1*, pages 679–684. IEEE Computer Society, 2005.

[65] M. Matsui. Linear cryptoanalysis method for DES cipher. In T. Helleseth, editor, *EUROCRYPT*, volume 765 of *Lecture Notes in Computer Science*, pages 386–397. Springer, 1993.

[66] M. Matsui. The first experimental cryptanalysis of the Data Encryption Standard. In Y. Desmedt, editor, *CRYPTO*, volume 839 of *Lecture Notes in Computer Science*, pages 1–11. Springer, 1994.

[67] ECRYPT II. Yearly report on algorithms and keysizes (2011-2012). Deliverable D.SPA.20 Revision 1.0, September 2012. `http://www.ecrypt.eu.org/documents/D.SPA.20.pdf`.

[68] National Institute of Standards and Technology. Data Encryption Standard. Federal Information Processing Standard (FIPS), Publication 46, U.S. Department of Commerce, Washington D.C., January 1977.

[69] National Institute of Standards and Technology. Advanced Encryption Standard. Federal Information Processing Standard (FIPS), Publication 197, U.S. Department of Commerce, Washington D.C., November 2001. `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`.

[70] W. Michiels. Opportunities in white-box cryptography. *IEEE Security & Privacy*, 8(1):64–67, 2010.

[71] W. Michiels. White-box cryptographic system with configurable key using block selection. WO 2010146140, 2010.

[72] W. Michiels. White-box cryptographic system with configurable key using intermediate data modification. WO 2010146139, 2010.

[73] W. Michiels and P. Gorissen. Mechanism for software tamper resistance: an application of white-box cryptography. In M. Yung, A. Kiayias, and A.-R. Sadeghi, editors, *Digital Rights Management Workshop*, pages 82–89. ACM, 2007.

[74] W. Michiels and P. Gorissen. White-box cryptography system with input dependent encodings. WO 2010102960, 2010.

[75] W. Michiels, P. Gorissen, and H. D. L. Hollmann. Cryptanalysis of a generic class of white-box implementations. In R. M. Avanzi, L. Keliher, and F. Sica, editors, *Selected Areas in Cryptography*, volume 5381 of *Lecture Notes in Computer Science*, pages 414–428. Springer, 2008.

[76] Microsoft Windows. Windows Media Player DRM: frequently asked questions. `http://windows.microsoft.com/en-us/windows-vista/windows-media-player-drm-frequently-asked-questions`.

[77] F. Miller. Telegraphic code to insure privacy and secrecy in the transmission of telegrams. Charles M. Cornwell, New York, 1882.

[78] J. Muir. A tutorial on white-box AES. In E. Kranakis, editor, *Advances in Network Analysis and its Applications*, volume 18 of *Mathematics in Industry*, pages 209–229. Springer Berlin Heidelberg, 2013.

[79] S. Murphy and M. J. B. Robshaw. Essential algebraic structure within the AES. In M. Yung, editor, *CRYPTO*, volume 2442 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2002.

[80] Nagra Kudelski Group. Homepage. `http://www.nagra.com/cms/`.

[81] K. Nyberg. Linear approximation of block ciphers. In A. D. Santis, editor, *EUROCRYPT*, volume 950 of *Lecture Notes in Computer Science*, pages 439–444. Springer, 1994.

[82] OpenSSL. OpenSSL 1.0.1e, February 2013. `http://www.openssl.org`.

[83] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: The case of AES. In D. Pointcheval, editor, *CT-RSA*, volume 3860 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2006.

[84] J. Patarin. Hidden fields equations (HFE) and isomorphisms of polynomials (IP): Two new families of asymmetric algorithms. In U. M. Maurer, editor, *EUROCRYPT*, volume 1070 of *Lecture Notes in Computer Science*, pages 33–48. Springer, 1996.

[85] G. Piret and J.-J. Quisquater. A differential fault attack technique against SPN structures, with application to the AES and KHAZAD. In C. D. Walter, Çetin Kaya Koç, and C. Paar, editors, *CHES*, volume 2779 of *Lecture Notes in Computer Science*, pages 77–88. Springer, 2003.

[86] M. Plasmans. White-box cryptography for digital content protection. Master's thesis, Technische Universiteit Eindhoven, May 2005. `http://alexandria.tue.nl/extra2/afstversl/wsk-i/plasmans2005.pdf`.

[87] J.-J. Quisquater and D. Samyde. Electromagnetic analysis (EMA): Measures and counter-measures for smart cards. In I. Attali and T. P. Jensen, editors, *E-smart*, volume 2140 of *Lecture Notes in Computer Science*, pages 200–210. Springer, 2001.

[88] H. Raddum. More dual Rijndaels. In H. Dobbertin, V. Rijmen, and A. Sowa, editors, *AES Conference*, volume 3373 of *Lecture Notes in Computer Science*, pages 142–147. Springer, 2004.

[89] R. L. Rivest. Cryptography. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A)*, pages 717–755. 1990.

[90] R. L. Rivest, A. Shamir, and L. M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[91] P. Roelse and Y. De Mulder. Updating key information. WO 2013139380, 2013.

[92] SafeNet Inc. Homepage. `http://www.safenet-inc.com/`.

[93] T. Sander and C. F. Tschudin. Protecting mobile agents against malicious hosts. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*, pages 44–60. Springer, 1998.

[94] T. Sander and C. F. Tschudin. Towards mobile cryptography. In *IEEE Symposium on Security and Privacy*, pages 215–224. IEEE Computer Society, 1998.

[95] R. Schultz. The many facades of DRM. *MISC HS 5 magazine*, pages 58–64, April 2012. `http://www.whiteboxcrypto.com/files/2012_MISC_DRM.pdf`.

[96] SciEngines. Break DES in less than a single day, 2009. `http://www.sciengines.com/company/news-a-events/74-des-in-1-day.html`.

[97] A. Shamir and N. van Someren. Playing "hide and seek" with stored keys. In M. K. Franklin, editor, *Financial Cryptography*, volume 1648 of *Lecture Notes in Computer Science*, pages 118–124. Springer, 1999.

[98] C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27(4):379–423, 623–656, 1948.

[99] C. E. Shannon. Communication theory of secrecy systems. *Bell System Technical Journal*, 28(4):656–715, 1949.

[100] A. Tardy-Corfdir and H. Gilbert. A known plaintext attack of FEAL-4 and FEAL-6. In J. Feigenbaum, editor, *CRYPTO*, volume 576 of *Lecture Notes in Computer Science*, pages 172–181. Springer, 1991.

[101] L. Tolhuizen. Improved Cryptanalysis of an AES implementation. *33rd WIC Symposium on Information Theory in the Benelux*, 2012.

[102] G. S. Vernam. Cipher printing telegraph systems for secret wire and radio telegraphic communications. *Journal of the American Institute of Electrical Engineers*, 55:109–115, 1926.

[103] B. Wyseur. *White-Box Cryptography.* PhD thesis, Katholieke Universiteit Leuven, 2009. Bart Preneel (promotor).

[104] B. Wyseur. White-box cryptography: Hiding keys in software. *MISC HS 5 magazine*, pages 65–72, April 2012. `http://www.whiteboxcrypto.com/files/2012_misc.pdf`.

[105] B. Wyseur, W. Michiels, P. Gorissen, and B. Preneel. Cryptanalysis of white-box DES implementations with arbitrary external encodings. In C. M. Adams, A. Miri, and M. J. Wiener, editors, *Selected Areas in Cryptography*, volume 4876 of *Lecture Notes in Computer Science*, pages 264–277. Springer, 2007.

[106] J. Xiao and Y. Zhou. Generating large non-singular matrices over an arbitrary field with blocks of full rank. *IACR Cryptology ePrint Archive*, 2002:96, 2002.

[107] Y. Xiao and X. Lai. A secure implementation of white-box AES. In *2nd International Conference on Computer Science and its Applications (CSA 2009)*, pages 1–6. IEEE, 2009.

[108] H. Yamauchi, A. Monden, M. Nakamura, H. Tamada, Y. Kanzaki, and K.-i. Matsumoto. A goal-oriented approach to software obfuscation. *IJCSNS International Journal of Computer Science and Network Security*, 8(9):59–71, 2008.

List of Publications

International Articles

1. T. Lepoint, M. Rivain, Y. De Mulder, P. Roelse, and B. Preneel. Two attacks on a white-box AES implementation. In T. Lange, K. Lauter and P. Lisonek, editors, *Selected Areas in Cryptography, 20th Annual International Workshop, SAC 2013, Lecture Notes in Computer Science.* Springer, 2013, *to appear.* (merged paper)

2. Y. De Mulder, P. Roelse, and B. Preneel. Cryptanalysis of the Xiao-Lai white-box AES implementation. In L. R. Knudsen and H. Wu, editors, *Selected Areas in Cryptography, 19th Annual International Workshop, SAC 2012*, volume 7707 of *Lecture Notes in Computer Science*, pages 34–49. Springer, 2012.

3. Y. De Mulder, K. Wouters, and B. Preneel. A privacy-preserving ID-based group key agreement scheme applied in VPAN. In I. Cerná, T. Gyimóthy, J. Hromkovic, K. G. Jeffery, R. Královic, M. Vukolic, and S. Wolf, editors, *37th Conference on Current Trends in Theory and Practice of Informatics, SOFSEM 2011*, volume 6543 of *Lecture Notes in Computer Science*, pages 214–225. Springer, 2011.

4. Y. De Mulder, B. Wyseur, and B. Preneel. Cryptanalysis of a perturbated white-box AES implementation. In G. Gong and K. C. Gupta, editors, *INDOCRYPT 2010*, volume 6498 of *Lecture Notes in Computer Science*, pages 292–310. Springer, 2010.

5. Y. De Mulder, G. Danezis, L. Batina, and B. Preneel. Identification via location-profiling in GSM networks. In V. Atluri and M. Winslett, editors, *7th ACM Workshop on Privacy in the Electronic Society, WPES 2008*, pages 23–32. ACM, 2008.

National Articles

6. Y. De Mulder, J. Cappaert, N. Kisserli, N. Mavrogiannopoulos, and B. Preneel. Perturbated functions: a new approach to obfuscation and diversity. In *2010th Benelux Workshop on Information and System Security*, *WISSec 2010*, 11 pages, 2010.

7. Y. De Mulder, K. Wouters, and B. Preneel. Anonymous ID-based group key agreement scheme applied in Virtual Private Ad Hoc Networks. In *3rd Benelux Workshop on Information and System Security*, *WISSec 2008*, 14 pages, 2008.

Patent

8. P. Roelse and Y. De Mulder. Updating key information. WO 2013139380, 2013.

Others

9. Y. De Mulder, P. Roelse, and B. Preneel. Revisiting the BGE attack on a white-box AES implementation. Cryptology ePrint Archive, Report 2013/450, 2013. `http://eprint.iacr.org/2013/450.pdf`.

FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF ELECTRICAL ENGINEERING (ESAT)
COMPUTER SECURITY AND INDUSTRIAL CRYPTOGRAPHY (COSIC)
Kasteelpark Arenberg 10, box 2452
B-3001 Heverlee