On the practical security of white-box cryptography

Junwei Wang

UNIVERSITÉ DU LUXEMBOURG
PhD-FSTM-2020-30
The Faculty of Science,
Technology and Medicine

UNIVERSITÉ PARIS 8
VINCENNES-SAINT-DENIS

DISSERTATION

Defence held on 06/24/2020 in Paris
to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG
ET DE L'UNIVERSITÉ PARIS 8
EN INFORMATIQUE

by

Junwei WANG (王军委)
Born on 14 May 1990 in Weifang (China)

ON THE PRACTICAL SECURITY OF WHITE-BOX CRYPTOGRAPHY

Dissertation defence committee

Dr. Alex Biryukov
Professor, Université du Luxembourg

Dr. Jean-Sébastien Coron, dissertation supervisor
Professor, Université du Luxembourg

Dr. Pierre-Alain Fouque, reporter
Professor, Université Rennes 1

Dr. Sihem Mesnager, dissertation supervisor
McF, HDR, Université Paris 8

Dr. Bart Preneel, reporter
Professor, KU Leuven

Dr. Matthieu Rivain, industrial supervisor
CryptoExperts

Dr. Wolfgang Schmid, chairman
Professor, Université Paris 8

Dr. François-Xavier Standaert
Professor, Université catholique de Louvain

DOCTORAL THESIS

OF

UNIVERSITY OF LUXEMBOURG

AND OF

UNIVERSITY PARIS 8

IN COMPUTER SCIENCE

ON THE PRACTICAL SECURITY OF WHITE-BOX CRYPTOGRAPHY

---

presented by

Junwei WANG

王军委

supervised by

Pr. Jean-Sébastien CORON *and* Pr. Sihem MESNAGER

under the guidance of

Dr. Pascal PAILLIER *and* Dr. Matthieu RIVAIN

prepared at

CRYPTOEXPERTS

Acknowledgments

This thesis could not have been accomplished without the help of many people. I would like to take this opportunity to express my sincere gratitude to them.

First of all, I would like to thank my respectable supervisors Jean-Sébastien Coron, Sihem Mesnager, Pascal Paillier and Matthieu Rivain for their willingness to accept me as their student, and for offering me all their selfless support and help that I needed during my PhD in every way. I am particularly grateful to Jean-Sébastien for referring me to many places when I was looking for a PhD position and for introducing me to my advisors Pascal and Matthieu. I also would like to thank Pascal and Matthieu for providing me with a place at CryptoExperts where I can concentrate on scientific research. Special acknowledgments to Matthieu for his patient guidance and meticulous care. Most of the work in this article was completed absolutely due to Matthieu's excellent ideas and constructive opinions and comments, as well as to the constant affirmation and encouragement from Sihem and Pascal.

Secondly, I would like to thank all the members of the defense committee. I especially thank Pierre-Alain Fouque and Bart Preneel for reviewing my manuscript, providing valuable feedback, and granting me permission to defend it. I am thankful to Alex Biryukov and Francois-Xavier Standaert for their willingness to be jury members, to Wolfgang Schmid for organizing and chairing the defense.

Thanks to all my co-authors: Louis Goubin, Andrey Bogdanov, Pascal Paillier, Matthieu Rivain, Philip S. Vejre for their contribution to making this thesis possible and for many discussions with them. Special thanks to Louis, together with whom I spent two summers working on the WhibOx Competitions, and many hours on enlightening topics in white-box cryptography and related domains.

I thank my master advisors Qiuliang Xu and Johann Großschädl for leading me into the world of cryptography, and for motivating me to obtain a doctoral degree in this field even after two years of working.

I would like to thank all my colleagues and former colleagues at CryptoExperts: Thomas Baignères, Sonia Belaïd, Cécile Delerablée, Matthieu Finiasz, Louis Goubin, Dahmun Goudarzi, Antoine Joux, Pascal Paillier, Matthieu Rivain, Aleksei Udovenko. It's always pleasant and relaxing to work with them. Special thanks to Sonia for always being ready to offer comfort, help and encouragement in my hard times.

I would also like to thank Alex Biryukov and Aleksei Udovenko again for contributing such interesting and challenging implementations to the WhibOx contests. Many of the results in this paper were inspired by their work.

I would also like to thank Albert Spruyt for hosting my visit at Riscure.

This thesis is financially funded by the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement ITN ECRYPT-NET (No. 643161). I would like to thank the European Union, the

ECRYPT-NET project initiators, coordinators, contributors and all lovely ECRYPT-NET fellows.

This list of acknowledgments will never be complete. Lastly, I would like to express my warmest gratitude to my family, especially to my mother and to my wife, Lin Li, for everything.

Abstract

Cryptography studies how to secure communications and information. The security of a cryptosystem depends on the secrecy of the underlying key. White-box cryptography explores methods to hide a cryptographic key into some software deployed in the real world.

Classical cryptography only assumes that the adversary accesses the target cryptographic primitive in a black-box manner in which she can only observe or manipulate the input and output of the primitive, but cannot know or tamper with its internal details. The gray-box model further allows an adversary to exploit key-dependent sensitive information leaked from the execution of physical implementations. All sorts of side-channel attacks exploit some physical information leakage, such as the power consumption of the device. The white-box model considers the worst-case scenario in which the adversary has complete control over the software and its execution environment. The goal of white-box cryptography is to securely implement a cryptographic primitive against such a powerful adversary. Although the scientific community has proposed some candidate solutions to build white-box cryptography, all have proven ineffective. Consequently, this problem has remained open for almost two decades since the concept was introduced.

The continuous growth in market demand and the emerging potential applications have driven the industry to deploy secretly-designed proprietary solutions. Although this paradigm of achieving security through obscurity contradicts the widely accepted Kerckhoffs' principle in cryptography, this is currently the only option for white-box cryptography. Security experts have reported how gray-box attacks could be used to extract keys from several publicly available white-box implementations. In a gray-box attack, the adversary adapts side-channel analysis techniques to the white-box context, i.e., to target computation traces made of noise-free runtime information instead of the noisy physical leakage. Gray-box attacks are generic since they do not require any a priori knowledge of the implementation and hence avoid costly reverse engineering. Some non-publicly scrutinized industrial white-box schemes in the market are believed to be under the threat of gray-box attacks.

This thesis focuses on the analysis and improvement of gray-box attacks and the associated countermeasures for white-box cryptography. We first provide an in-depth analysis of why gray-box attacks are capable of breaking the classical white-box design which is based on table encodings. Next, we introduce a new gray-box attack named linear decoding analysis and show that linearly encoding sensitive information is insufficient to protect the cryptographic software. Afterward, we describe how to combine state-of-the-art countermeasures to resist gray-box attacks and comprehensively elaborate on the (in)effectiveness of these combined countermeasures in terms of computation complexity. Finally, we introduce a new attack technique that exploits the data-dependency of the targeted implementation to substantially

lower the complexity of the existing gray-box attacks on white-box cryptography. In addition to the theoretical analyses and new attack techniques introduced in this thesis, we report some attack experiments against practical white-box implementations. In particular, we could break the winning implementations of two consecutive editions of the well-known WhibOx white-box cryptography competition.

Résumé

La cryptographie est la science de la protection des communications et des données. La sécurité d'un cryptosystème dépend du secret de la clé sous-jacente. La cryptographie en boîte blanche explore les méthodes permettant de cacher une clé dans un logiciel cryptographique déployé dans le monde réel.

La cryptographie classique suppose que l'adversaire accède à la primitive cryptographique ciblée en boîte noire. Cela signifie qu'elle ne peut qu'observer et manipuler les entrées et les sorties de la primitive, mais ne peut pas connaître ou altérer son état interne. Le modèle en boîte grise permet en outre à un adversaire d'observer des informations sensibles qui sont divulguées lors de l'exécution d'une implémentation de la primitive. Toutes sortes d'attaques par canaux auxiliaires exploitent certaines fuites d'informations physiques, telles que la consommation électrique de l'appareil. Le modèle en boîte blanche considère le scénario le plus défavorable dans lequel l'adversaire a un contrôle total sur le logiciel cryptographique et sur son environnement d'exécution. Le rôle de la cryptographie en boîte blanche est d'implémenter une primitive cryptographique de façon à protéger la clé secrète contre un tel adversaire. Bien que la communauté scientifique ait tenté à plusieurs reprises de solutionner ce problème, ces tentatives se sont toutes révélées inadéquates. Par conséquent, la construction d'une solution de cryptographie en boîte blanche est resté un problème ouvert depuis l'introduction du concept il y a deux décennies.

L'émergence d'applications avec de fortes contraintes de sécurité logicielle a poussé l'industrie à développer des solutions propriétaires dont la sécurité repose (en partie) sur des secrets de conception. Bien que ce paradigme de sécurité par l'obscurité contredise le principe de Kerckhoffs largement admis en cryptographie, c'est actuellement la seule option s'offrant à l'industrie pour répondre au besoin de cryptographie en boîte blanche. Des experts en sécurité ont récemment démontré comment certaines attaques en boîtes grise pouvaient être utilisées pour extraire les clés de plusieurs implémentations en boîte blanche accessibles publiquement. Dans une attaque en boîte grise, l'adversaire adapte des techniques d'analyse par canaux auxiliaires au contexte boîte blanche, en replaçant la fuite physique par des traces de calcul faites des valeurs intermédiaires non-bruitées observées lors de l'exécution. Les attaques en boîte grise sont génériques car elles ne nécessitent aucune connaissance a priori de l'implémentation et évitent ainsi la nécessité pour l'attaquant de recourir à une rétro-ingénierie coûteuse. Il semble que certaines solutions de cryptographie en boîte blanche actuellement déployée et n'ayant pas fait l'objet d'un examen public soient menacées par ce type d'attaques en boîte grise.

Cette thèse se concentre sur l'analyse et l'amélioration des attaques en boîte grise et des contre-mesures associées pour la cryptographie en boîte blanche. Nous

présentons tout d'abord une analyse approfondie des raisons pour lesquelles les attaques en boîte grise basiques sont capables de casser la technique classique de cryptographie en boîte blanche basée sur les encodages de tables. Nous proposons également de nouvelles techniques d'attaque en boîte grise significativement plus efficace contre ce type d'encodages. Nous introduisons ensuite une nouvelle attaque en boîte grise appelée analyse par décodage linéaire qui permet de déjouer toute méthode de protection basée sur un encodage linéaire des variables internes au calcul. Par la suite, nous étudions la combinaison de différentes contre-mesures pour résister aux attaques en boîte grise et analysons en détail la complexité d'attaques avancées contre ces contre-mesures combinées. Nous introduisons enfin une nouvelle technique d'attaque qui exploite le graphe de calcul de l'implémentation ciblée pour réduire considérablement la complexité des attaques en boîte grise sur la cryptographie en boîte blanche. Outre les analyses théoriques et nouvelles techniques d'attaque introduites dans cette thèse, nous rapportons plusieurs expériences d'attaque pratique contre divers implémentations en boîte blanche. Nous démontrons notamment comment nous avons pu casser les implémentations gagnantes des deux éditions consécutives de la compétition WhibOx.

摘 要

密码学是一门研究保护数据和通信安全的科学。一个密码系统的安全性取决于其密钥的保密性。白盒密码学探索在现实世界中部署的软件中隐藏密钥的方法。

经典密码学仅假定敌手以黑盒的方式访问被研究的密码学原语，在这种方式中，她只能观察或操作原语的输入和输出，而无法知道或篡改其内部细节。灰盒模型进一步允许敌手利用物理实现执行过程中泄露的与密钥相关的敏感信息。各种侧信道攻击都是利用一些物理信息泄露，比如设备的功耗。白盒模型考虑的是最坏的情况，即敌手完全控制了软件及其执行环境。白盒密码学的目标即是在敌手拥有上述强大能力的情况下，给出密码学原语的安全软件实现。尽管学术界提出了一些构建白盒密码学的候选方案，但事实证明这些方案都无效。因此，自概念提出以来，这个开放问题持续存在了近二十年。

白盒密码市场需求的持续增长和新兴的潜在应用促使工业界采用秘密设计的专有解决方案。尽管这种通过模糊来实现安全性的范例与密码学中被广泛接受的 Kerckhoffs 原则相悖，但这是在当前困境中的无奈之举。安全专家报道了如何使用灰盒攻击从几种公开的白盒实现中提取密钥。在灰盒攻击中，敌手将侧信道分析技术应用到白盒密码学中。她的研究对象是从软件运行时收集不带任何噪音计算数据而非带有噪声的物理泄漏。灰盒攻击是一种通用攻击，因为它们不需要关于攻击对象的任何先验知识，从而避免了代价高昂的逆向工程。市场上一些未经公开审阅的工业白盒计划被认为正受到灰盒攻击的威胁。

本论文主要研究针对白盒密码学进行的灰盒攻击的分析与改进以及相关对策。首先，我们深入分析了为什么灰盒攻击能够打破经典的基于编码表的白盒设计。接下来，我们介绍了一种新的称为线性解码分析的灰盒攻击，并阐释了仅通过对敏感信息进行线性编码的方式来保护软件中的密钥是不足够的。随后，我们将描述如何通过组合最新的对策来抵抗灰盒攻击，并从计算复杂性的角度全面阐述这些组合对策的有效性和无效性。最后，我们介绍了一种新的攻击技术，该技术利用目标实现的数据依赖性来显著降低现有灰盒攻击应用在白盒密码学的复杂性。本论文除了介绍的理论分析和新的攻击技术外，还报告了一些针对实际白盒实现的攻击实验。特别的是，我们连续两届打破了著名的 WhibOx 白盒密码学竞赛中的获胜实现。

Contents

Chapter 1

Introduction (en français)

1.1 Cryptographie en boîte blanche

1.1.1 Nuances de gris

Dans le cadre cryptanalytique classique, l'adversaire est confronté au défi de casser la sécurité, p. ex., d'un algorithme de chiffrement tout en ne pouvant considérer l'algorithme que comme une boîte noire ; il peut interroger la boîte avec des entrées et recevoir les sorties correspondantes. Bien que la conception de l'algorithme soit connue, l'adversaire ne peut pas observer l'état interne de l'algorithme, ni altérer l'exécution de l'algorithme.

En pratique, un algorithme cryptographique doit être implémenté, en matériel ou logiciel, afin d'être utilisable. Selon le contexte d'utilisation, l'adversaire peut alors potentiellement interagir physiquement avec le dispositif cryptographique. Dans ce cas, l'adversaire a accès aux informations des canaux auxiliaires (*side-channel information*) spécifiques à l'implémentation (Kocher, 1996; Kocher et al., 1999; Coron, 1999). Si une implémentation n'est pas suffisamment protégée, des informations telles que le temps d'exécution ou la consommation d'énergie peuvent être utilisées pour extraire des informations secrètes, p. ex., des clés de chiffrement. L'utilisation répandue et le succès des attaques par canaux auxiliaires montrent qu'un cryptographe doit être très prudent lorsqu'il opère dans ce modèle en *boîte grise*.

Pour les implémentations matérielles, le modèle en boîte grise est souvent la limite de ce qu'un adversaire peut réaliser. Ce n'est pas le cas pour les implémentations logicielles exécutées dans des environnements non fiables. Si un adversaire a un accès complet à l'environnement d'exécution du logiciel cryptographique, il peut facilement observer et manipuler l'exécution de la primitive, instanciée avec une clé secrète. Ce scénario, introduit par Chow et al. dans (Chow, Eisen, Johnson, and van Oorschot, 2003), est appelé le modèle en boîte blanche.

Ainsi, les mises en œuvre logicielles des algorithmes cryptographiques dans le monde réel sont souvent sensibles à des menaces plus importantes que celles envisagées lors de leur conception. En plus des attaques bien connues de l'analyse par canaux auxiliaires (*side-channel analysis*, SCA), un adversaire en boîte blanche obtient un accès complet à une implémentation logicielle d'un algorithme cryptographique. Il peut alors tenter d'extraire la clé secrète sous-jacente par toutes sortes de moyens. En général, il peut effectuer une analyse statique ou dynamique du binaire contrôlé, p. ex., en étudiant la structure de contrôle de l'implémentation (graphe de dépendance des opérations); il peut choisir arbitrairement les entrées pour le logiciel et observer toutes les informations d'exécution, telles que les adresses et les valeurs de la mémoire accédée ; il peut également altérer les implémentations, par exemple en modifiant la structure de contrôle ou en provoquant de fautes lors de l'exécution comme dans l'analyse (différentielle) par injection de faute (Boneh et al., 1997; Biham and Shamir, 1997).

Récemment, les applications critiques pour la sécurité, telles que les systèmes de gestion des droits numériques (*digital right management*, DRM) et les services de

paiement mobile, ont connu un développement rapide et un large déploiement sur les appareils électroniques grand public. Des algorithmes cryptographiques sont généralement utilisés dans ces contextes pour assurer la confidentialité, l'intégrité et l'authenticité des données. Comme ces applications sont généralement hébergées dans des environnements non fiables et / ou que les utilisateurs eux-mêmes peuvent représenter des attaquants potentiels, la menace en boîte blanche doit alors être prise en compte par les concepteurs et les évaluateurs de la sécurité. Si la clé intégrée dans une implémentation sous-jacente était extraite par l'attaquant (et que plusieurs fonctionnalités de sécurité étaient compromises en même temps), non seulement l'objectif de sécurité poursuivi pourrait être perdu, mais l'activité commerciale associée pourrait également être menacée. Par exemple, un attaquant pourrait faire des profits illégaux en vendant la clé révélée dans une application DRM sur le marché noir à un prix beaucoup moins élevé. Il est donc crucial d'étudier la capacité d'un adversaire en boîte blanche ainsi que les contre-mesures susceptibles d'empêcher l'exposition de la clé.

1.1.2 L'histoire: un jeu du chat et de la souris

"Quand l'attaquant peut obtenir de l'information sur l'état interne d'une implémentation cryptographique, le choix de l'implémentation constitue la seule ligne de défense (Chow, Eisen, Johnson, and van Oorschot, 2003). Les deux articles fondateurs de la cryptographie en boîte blanche (white-box cryptography, WBC), présentés par Chow et al. en 2002 (Chow, Eisen, Johnson, and Oorschot, 2002) and 2003 (Chow, Eisen, Johnson, and van Oorschot, 2003) respectivement, visent à protéger les implémentations logicielles de chiffrement par blocs standard (DES et AES) utilisées dans les applications DRM contre ce type de menaces. L'idée générale qui sous-tend leurs constructions est d'implémenter un chiffrement sous la forme d'un réseau de tables de correspondance (look-up tables) précalculées et encodées aléatoirement, de telle sorte qu'un adversaire ne puisse exploiter les variables intermédiaires ainsi randomisées. Les fonctions tirées aléatoirement et appliquées aux tables de correspondance sont appelées encodages. Elles peuvent être divisés en deux catégories : les encodages *internes* et les encodages *externes*. En particulier, les encodages externes sont des bijections appliquées sur l'entrée ou la sortie du chiffrement. Cependant, l'application de encodages externes modifie la spécification du chiffrement d'origine, ce qui est prohibitif pour de nombreux cas d'utilisation de la cryptographie en boîte blanche basée sur des algorithmes cryptographiques (standard) prédéfinis.

Bientôt, ces techniques ont été cassées par Billet et al. avec la cryptanalyse structurelle (Billet et al., 2004). L'attaque fonctionne selon le principe que chacune des tables de correspondance ne laisse échapper aucune information sensible dépendant de la clé lorsqu'elle est considérée séparément, mais lorsqu'elles sont combinées ensemble, elles révèlent des informations sur les encodages utilisés. Cette observation permet aux auteurs de simplifier les encodages et de récupérer entièrement la clé secrète dans l'implémentation. Depuis lors, la concurrence entre les concepteurs

d'implémentations boîte blanche et les attaquants est devenue un *jeu du chat et de la souris*. La communauté des chercheurs a observé de nombreuses constructions candidates différentes d'implémentation boîte blanche pour AES (Link and Neumann, 2005; Bringer, Hervé Chabanne, et al., 2006; Bringer, Herve Chabanne, et al., 2006; Xiao and Lai, 2009; Karroumi, 2011), ainsi que leur cassage ultérieure par l'analyse structurelle peu après, voire des années plus tard (Goubin, Masereel, et al., 2007; Wyseur et al., 2007; Michiels et al., 2009; De Mulder, Wyseur, et al., 2010; Tolhuizen, 2012; De Mulder, Roelse, et al., 2013b; De Mulder, Roelse, et al., 2013a; Lepoint and Rivain, 2013; Lepoint, Rivain, et al., 2014).

1.1.3 Progrès théoriques

En tant que l'un des composants clés de la protection des applications du monde réel, la cryptographie en boîte blanche (*white-box cryptography*, WBC) cherche une solution pour transformer un algorithme cryptographique avec une clé donnée en une implémentation obscurcie. Idéalement, l'adversaire en boîte blanche qui contrôle l'implémentation ne devrait pas avoir d'avantage significatif par rapport à la situation dans laquelle il ne pourrait accéder qu'à un oracle répondant à des requêtes de chiffrement (sous la même clé). En particulier, la cryptographie en boîte blanche vise à rendre l'*extraction de clé* difficile - voire impossible - pour toute partie malveillante qui obtiendrait un accès complet au programme et / ou à l'environnement d'exécution.

Jusqu'à présent, peu de travaux ont été conduits sur la formalisation de la cryptographie en boîte blanche. Deux premiers travaux (Saxena et al., 2009; Delerablée et al., 2014) ont introduit quelques notions formelles de sécurité en boîte blanche. Plus précisément, Saxena et al. (Saxena et al., 2009) montrent comment adapter les notions de sécurité du modèle en boîte noire (Barak et al., 2001) aux notions de sécurité du modèle en boîte blanche ; tandis que (Delerablée et al., 2014) formalisent la propriété de base de l'*incassabilité* et plusieurs autres notions utiles : *sens unique*, *incompressibilité*, et *traçabilité* pour les chiffrements symétriques.

En particulier, comme expliqué dans (Delerablée et al., 2014), l'exigence minimale pour la cryptographie en boîte blanche est la résistance à l'extraction de la clé (formalisée comme *incassabilité*). Cependant, cette propriété seule ne *suffit* pas en pratique. L'adversaire pourrait éviter d'extraire la clé secrète d'une implémentation, mais plutôt utiliser l'implémentation en boîte noire (c.-à-d., comme un oracle) de l'algorithme sous-jacent. Une telle attaque est appelée "*code lifting*" et d'autres dispositifs de sécurité devraient être prévus pour empêcher cette menace. On pourrait rendre l'implémentation large et *incompressible* pour durcir son extraction, la rendre à *sens unique* pour empêcher soit le chiffrement soit le déchiffrement, ou la rendre *traçable* pour contrer le partage non autorisé. Il a également été suggéré, par exemple, de s'appuyer sur des *encodages externes* (Chow, Eisen, Johnson, and van Oorschot, 2003) et / ou de lier l'implémentation à un mot de passe choisi par l'utilisateur, ou

à certaines données biométriques, ou à la propriété physique du matériel (Alpirez Bock et al., 2020).

Malgré son intérêt pratique, aucune implémentation boîte blanche dont la sécurité soit prouvée n'a été produite après 20 ans d'exploration. Néanmoins, de nombreux travaux ont été réalisés dans le domaine connexe de l'offuscation cryptographique. L'offuscation des logiciels peut être définie comme le problème de la création d'un programme fonctionnellement équivalent à un programme cible, mais dont le code source (ou binaire) est inintelligible. Malheureusement, l'offuscation générale au sens cryptographique (*virtual black-box obfuscation*), dans lequel l'adversaire ne peut rien apprendre du programme obscurci à l'exception de ses comportements d'entrée et de sortie, s'est avérée impossible (Barak et al., 2001). Un assouplissement de la notion originale, appelée $i\mathcal{O}$ (*indistinguishability obfuscation*), semble pouvoir être atteint et la con conception d'un schéma d'offuscation atteignant cette notion est devenu un problème ouvert majeur de la recherche au cours des dernières années (Garg, Gentry, Halevi, et al., 2013; Garg, Gentry, and Halevi, 2013; Sahai and Waters, 2014; Lin, 2016; Lin, 2017; Lin and Tessaro, 2017). Littéralement, un offuscateur $i\mathcal{O}$ embrouille l'attaquant en le rendant incapable de déterminer l'origine du programme obscurci entre deux programmes sources fonctionnellement équivalents. Un aperçu complet des problèmes liés à $i\mathcal{O}$ est donné dans (Horváth, 2015). Cependant, la propriété $i\mathcal{O}$ n'implique pas directement la cryptographie en boîte blanche dans le sens où l'application d'un compilateur $i\mathcal{O}$ à un programme de chiffrement ne garantit pas que l'extraction de la clé du programme résultant soit difficile. En outre, les constructions actuelles de $i\mathcal{O}$ sont encore peu pratiques. D'autre part, la question de savoir si l'incassabilité boîte blanche peut être obtenue à partir de la propriété $i\mathcal{O}$ est aujourd'hui encore ouverte.

Une autre branche de la recherche théorique est consacrée à la construction de nouvelles primitives cryptographiques incompressibles par conception (Biryukov, Bouillaguet, et al., 2014; Bogdanov and Isobe, 2015; Fouque et al., 2016; Bock, Amadori, et al., 2019). Biryukov et Perrin décrivent un cadre unifié pour la complexité d'une primitive vis-à-vis de plusieurs ressources : vitesse, taille du code et mémoire (Biryukov and Perrin, 2017). Cette notion de *complexité en ressources* est très utile en pratique, car l'inefficacité peut être une caractéristique souhaitable dans certaines situations. Par exemple, la complexité en mémoire est utilisée dans le hachage des mots de passe pour ralentir la vitesse de recherche de l'attaquant "brute force". À certains égards, l'incompressibilité –et plus généralement la complexité en ressources– présente des similitudes avec la cryptographie en boîte blanche classique qui fait l'objet de la présente thèse, mais elles sont de nature très différente. La différence la plus évidente étant que la première propose principalement de nouvelles primitives atteignant certains niveaux de complexité en ressources par construction, tandis que la seconde vise à protéger les implémentations logicielles des algorithmes cryptographiques standards existants (et souvent imposés par le contexte).

1.2 La cryptographie en boîte blanche en pratique

1.2.1 L'obscurité en tant que solution

Comme mentionné ci-dessus, toutes les implémentations boîte blanche présentées dans la littérature sont peu sûres et aucune solution dont la sécurité est prouvée n'a été rapportée dans la littérature. Pourtant, il existe un besoin industriel croissant de protection des implémentations cryptographiques exécutées dans des environnements non fiables, comme par exemple dans le cas de l'utilisation de la gestion traditionnelle des droits numériques (*digital rights management*, DRM) et des applications de paiement mobile fonctionnant sur des appareils intelligents.

Dans cette situation, l'industrie est contrainte de développer des solutions *artisanales*, pour répondre aux besoins croissants de logiciels cryptographiques sécurisés. La sécurité des solutions propriétaires déployées repose principalement sur l'*obscurité*, c.-à-d., le secret de leur conception. Leur conception n'est donc pas examinée publiquement, ce qui est en contradiction avec le principe de Kerckhoffs largement admis en cryptographie. Néanmoins, le paradigme actuel vise à fournir une *sécurité pratique*, en ce sens que l'implémentation est suffisamment difficile à attaquer pour qu'un adversaire soit contraint à investir plus de ressources (puissance de calcul et/ou expertise humaine). Les attaques susmentionnées contre les implémentations boîte blanche *publiques* exploitent certaines failles dans les schémas en boîte blanche sous-jacents. Cependant, certaines variantes de schémas connus modifiant quelques paramètres ou combinant différentes techniques peuvent contrecarrer une simple application de ces attaques, à condition que la variante exacte soit gardée secrète. Contre une telle implémentation obscure, l'adversaire est contraint de procéder à une rétro-ingénierie, ce qui peut s'avérer très coûteux en temps, en puissance calcul, et/ou en ressources humaines si plusieurs couches d'offuscation ont été appliquées.

Les solutions industrielles de cryptographie en boîte blanche actuelles constituent ainsi un barrière de sécurité qui est souvent complétée par d'autres mesures. Dans ce paradigme, la cryptographie en boîte blanche est basée sur des mises à jour de sécurité régulières et/ou une rotation fréquente des clés secrètes exposées.

1.2.2 Attaques en boîte grise dans le contexte en boîte blanche

Tel que mentionné ci-dessus, les implémentations obscures peuvent être peu sûres contre un adversaire bien informé, mais la sécurité de leurs conceptions peut toutefois les rendre difficiles à casser dans un scénario réaliste. En effet, les attaques structurelles connues ne s'appliquent pas contre de telles implémentations, et une rétro-ingénierie fastidieuse doit être effectuée par des ingénieurs qualifiés.

Dans la littérature, deux principes d'attaques génériques en boîte grise ont été utilisés pour casser ces implémentations boîte blanche *obscures*. D'une part, à l'instar de l'analyse différentielle de la consommation (*differential power analysis*,

DPA) (Kocher et al., 1999), l'analyse différentielle du calcul (*differential computation analysis*, DCA) (Bos et al., 2016; Bock, Bos, et al., 2019) recherche la corrélation entre des variables sensibles dépendantes de la clé et des *traces de calcul* composées de valeurs traitées lors de l'exécution de l'implémentation ; d'autre part, puisqu'un chiffrement par blocs standard, p. ex., AES, est généralement intrinsèquement vulnérable à l'analyse différentielle par fautes (*differential fault analysis*, DFA) (Dusart et al., 2003), cette dernière peut également être appliquée pour casser une majorité des implémentations disponibles publiquement (Jacob et al., 2002; Sanfelix et al., 2015).

Notamment, l'adversaire en boîte grise est extrêmement puissant car il est *agnostique* à l'implémentation et n'a donc pas besoin d'exercer une rétro-ingénierie coûteuse. La DCA et la DFA ont cassé certaines solutions industrielles mélangeant les premières techniques en boîte blanche à de l'offuscation de code. La DFA peut être rendue difficile dans le paradigme de sécurité par l'obscurité, conformément à nos observations sur les implémentations gagnantes des deux compétitions de cryptographie en boîte blanche WhibOx (qui seront présentés dans la section 2.3). Dans cette thèse, nous nous concentrons sur les attaques *passives* en boîte grise qui correspondent à un adversaire plus faible et qui devraient donc être traitées en priorité par les concepteurs d'implémentations boîte blanche.

Analyse différentielle du calcul

L'analyse différentielle du calcul (*differential computation analysis*, DCA) est une méthode permettant d'attaquer les implémentations boîte blanche à la manière en boîte grise. Elle a été introduite indépendamment par deux équipes : Bos et al. à CHES 2016 et Sanfelix et al. à Black Hat Europe 2015. Les deux équipes ont démontré comment cette technique est capable de récupérer la clé de plusieurs implémentations boîte blanche existantes de l'AES.

En principe, la DCA est principalement une adaptation de l'analyse differentielle de la consommation (*differential power analysis*, DPA) (Kocher et al., 1999) au contexte de la cryptographie en boîte blanche. Elle exploite le fait que les variables apparaissant dans le calcul, bien que sous une forme encodée inconnue, peuvent avoir une forte corrélation linéaire avec les valeurs d'origine. Elle fonctionne en collectant d'abord quelques *traces de calcul*, qui sont composées des valeurs calculées pendant plusieurs exécutions grâce à un outil d'instrumentation dynamique, tel que Valgrind (Nethercote and Seward, 2007) ou Intel PIN (Luk et al., 2005). Elle fait ensuite une supposition de clé et prédit la valeur de la variable intermédiaire cible dépendant de la clé. Enfin, la corrélation entre cette prédiction et les traces de calcul est calculée. L'hypothèse de clé avec le pic le plus élevé dans la trace de corrélation obtenue est retenue. Cette approche s'est avérée particulièrement efficace pour extraire les clés de nombreuses implémentations boîte blanche (obscures) accessibles publiquement (Bos et al., 2016; Bock, Bos, et al., 2019).

Le puissance de la DCA réside dans le fait qu'elle ne nécessite pas une connaissance complète de l'implémentation boîte blanche ciblée. L'adversaire a uniquement

besoin d'observer les adresses et les valeurs de mémoire auxquelles on accède pendant l'exécution de l'implémentation. Il n'a pas besoin de raisonner sur les détails de l'implémentation, ni de modifier la fonctionnalité du code de quelque manière que ce soit – tâches qui pourraient nécessiter des efforts considérables. Ainsi, la DCA est une attaque *passive* et en *boîte grise*, qui peut casser de nombreuses implémentations pratiques en boîte blanche avec une faible complexité (Bos et al., 2016). Suite à ces observations, les implémentations boîte blanche actuelles ne sont même pas sécurisées dans un contexte d'attaque plus faible que celui pour lequel elles ont été conçues. Étant donné le modèle d'attaque faible de DCA et son efficacité contre les implémentations boîte blanche practiques, il est important de concevoir une implémentation boîte blanche dont la sécurité contre cette attaque peut être prouvée.

1.2.3 Contre-mesures pratiques

Pour empêcher les attaques passives en boîte grise de type DCA, il est naturel de prendre en compte les contre-mesures classiques par canaux auxiliaires, c.-à-d., le *masquage linéaire* et le *shuffling*. Le masquage linéaire (masquage booléen) divise une variable intermédiaire sensible en plusieurs parts, les *shares*, dont la somme permet de retrouver la variable initiale. Les shares sont traités indépendamment de manière à garantir l'exactitude du calcul tout en empêchant dans une certaine mesure les fuites d'informations sensibles. Le principe du shuffling est de permuter aléatoirement l'ordre de plusieurs opérations indépendantes (incluant éventuellement des opérations bidons) pour augmenter le bruit dans la fuite instantanée sur une variable sensible.

Nous montrerons qu'une implémentation protégée uniquement par un masquage linéaire est vulnérable à une analyse par décodage linéaire (*linear decoding analysis*, LDA) qui peut récupérer les emplacements de manipulation des shares en résolvant un système linéaire. À Asiacrypt 2018, Biryukov et Udovenko ont introduit la notion de masquage non linéaire à *sécurité algébrique* pour protéger les implémentations en boîte blanche contre la LDA (Biryukov and Udovenko, 2018). Le masquage non linéaire garantit que l'application de toute fonction linéaire aux variables intermédiaires de l'implémentation protégée ne devrait pas résulter en une variable *prédictible* avec une probabilité (proche de) 1. Cependant, le masquage non linéaire seul pourrait être vulnérable à l'attaque DCA standard. Il a alors été suggéré dans (Biryukov and Udovenko, 2018) de combiner le masquage linéaire et non linéaire pour pouvoir contrer les attaques DCA et LDA en même temps. L'intuition est double : d'une part, un masquage non linéaire à sécurité algébrique mélangé à un masquage linéaire ne devrait pas diminuer le degré algébrique pour construire une valeur prédictible ; d'autre part, les variables sont linéairement masquées ce qui assure la sécurité contre la DCA standard (d'ordre supérieur).

Une implémentation boîte blanche à l'état de l'art met typiquement en œuvre les contre-mesures mentionnées ci-dessus, ainsi qu'une couche d'offuscation. La

sécurité des implémentations repose sur les propriétés de sécurité (faibles) obtenues par les contre-mesures employées, ainsi que sur l'obscurité de la conception globale (c.-à-d., le niveau d'offuscation). L'objectif principal de ces implémentations est de contrecarrer les attaques automatiques en boîte grise, contraignant ainsi les adversaires potentiels à investir dans une rétro-ingénierie coûteuse et incertaine et à utiliser des techniques d'attaque plus complexes et spécialisées. Ces efforts peuvent prendre beaucoup de ressources et devenir prohibitif, notamment lorsque ces protections sont combinés à une stratégie de cible mouvante.

La génération d'aléa joue un rôle important dans l'implémentation de toutes les contre-mesures mentionnées. Il est bien connu qu'un adversaire en boîte blanche pourrait altérer le canal de commutation entre une implémentation boîte blanche et son monde extérieur, y compris un générateur aléatoire externe. Par conséquent, l'aléa utilisé dans une implémentation boîte blanche ne peut qu'être pseudo-aléatoire et dérivé de l'entrée. Dans cette thèse, chaque fois que nous faisons référence à l'aléa dans les contre-mesures décrites, nous parlons de pseudo-aléa dérivé de l'entrée.

1.3 Aperçu de la thèse

Ce manuscrit réorganise principalement mes quatre articles publiés dans des revues scientifiques à comités de relecture et les présentations que j'ai faites durant ma thèse. Cette section donne un aperçu de chacun des chapitres suivants. Pour éviter les répétitions, mes publications et présentations ne sont répertoriées que dans l'introduction en anglais de cette thèse (précisément, à la Section 2.4.1).

Chapitre 3: Contexte technique (Technical Background)

Dans ce chapitre, nous présentons d'abord tous les concepts et définitions mathématiques qui seront utilisés tout le long de cette thèse. Ensuite, nous formalisons un *modèle d'adversaire passif en boîte grise* pour la cryptographie en boîte blanche et reformulons l'analyse différentielle du calcul (*differential computation analysis*, DCA) dans ce modèle. Par la suite, nous passons en revue le masquage linéaire, le masquage non linéaire et les contre-mesures de shuffling. Enfin, nous discutons en détail de la source d'aléa nécessaire à l'implémentation de ces contre-mesures dans un contexte boîte blanche.

Chapitre 4: Attaques en boîte grise contre les encodages internes (Gray-Box Attacks against Internal Encodings)

L'encodage interne est la toute première technique en boîte blanche, encore couramment utilisée, pour protéger les implémentations de chiffrement par blocs. Il consiste à représenter une implémentation comme un réseau de tables de correspondance qui sont ensuite encodées à l'aide de bijections générées aléatoirement (les encodages

internes). L'implémentation protégée est vulnérable à l'analyse de calcul différentiel (DCA) lorsque les encodages utilisés s'appliquent à des *nibbles* (c.-à-d., des encodages d'une largeur de 4 bits). Pour contrecarrer la DCA, il a été suggéré d'utiliser des encodages plus larges, et en particulier des encodages d'octets, au moins pour protéger les tours externes du chiffrement par blocs qui sont les cibles principales de la DCA.

Dans ce chapitre, nous analysons en profondeur quand et pourquoi la DCA fonctionne. Nous identifions les propriétés des variables cibles et les encodages qui rendent l'attaque (in)faisable. En particulier, nous montrons que la DCA peut casser des encodages plus larges que 4 bits, tels que les encodages d'octets. De plus, nous proposons de nouvelles attaques de type DCA inspirées des techniques d'analyse par canaux auxiliaires. Plus précisément, nous décrivons une attaque par collision particulièrement efficace contre la contre-mesure d'encodage interne. Nous étudions également l'analyse par information mutuelle (mutual information analysis, MIA) qui s'applique naturellement dans ce contexte. Par rapport à la DCA originale, ces attaques sont également passives et elles nécessitent une connaissance très limitée de l'implémentation attaquée, mais elles permettent d'obtenir des améliorations significatives en termes de complexité de trace. Toutes les analyses de notre travail sont étayées expérimentalement par différents résultats de simulation d'attaques.

Chapitre 5: Analyse par décodage linéaire et extension aux degrés supérieurs (Linear Decoding Analysis and Higher-Degree Extension)

Les implémentations modernes en boîte blanche utilisent des techniques algébriques pour masquer les variables sensibles dépendantes de la clé pendant son exécution. L'encodage linéaire est l'une des nombreuses contre-mesures envisagées dans les solutions industrielles, ainsi que dans les implémentations boîte blanche disponibles publiquement, p. ex., le gagnant de la compétition WhibOx 2017 - `Adoring Poitras`. La DCA est inefficace contre les encodages linéaires, à moins que les encodages ne remplissent certaines conditions nécessaires. Dans ce chapitre, nous décrivons formellement l'attaque par analyse de décodage linéaire (*linear decoding analysis*, LDA) pour extraire la clé des implémentations boîte blanche, dans lesquelles les variables cibles sont encodées linéairement par un ensemble de variables intermédiaires dans l'implémentation. Nous expliquons ensuite comment cette attaque peut être étendue pour casser les implémentations protégées par des encodages de degré supérieur.

Chapitre 6: DCA d'ordre supérieur contre le masquage et le shuffling (Higher-Order DCA against Masking and Shuffling)

L'adversaire DCA est passif et n'exploite donc pas toute la puissance en boîte blanche, ce qui implique que de nombreux schémas en boîte blanche ne sont pas sûrs, même

dans un cadre plus faible que celui pour lequel ils ont été conçus. Il est donc important de développer des implémentations qui résistent à cette attaque. Une approche naturelle pour tenter d'atténuer la menace des attaques DCA consiste à appliquer les contre-mesures connues dans la littérature sur les canaux auxiliaires. Cependant, la question de l'application de ces contre-mesures au contexte de la cryptographie boîte blanche et le niveau de sécurité pouvant être atteint par cette approche restent à étudier.

Dans ce chapitre, nous examinons l'approche consistant à appliquer des contre-mesures standards aux attaques par canaux auxiliaires, telles que le masquage et le shuffling, pour contrecarrer les attaques DCA. Nous montrons que sous certaines conditions sur la source d'aléa, cette approche est suffisante pour atteindre la sécurité contre la DCA (de 1er ordre). D'autre part, nous introduisons la DCA d'ordre supérieur, ainsi qu'une version améliorée "multivariée", et nous analysons la sécurité des contre-mesures contre ces attaques. Nous dérivons des expressions analytiques de la complexité des attaques - étayées par des expériences d'attaque approfondies - permettant à un concepteur de quantifier le niveau de sécurité d'une implémentation protégée par du masquage et du shuffling vis-à-vis de la DCA (d'ordre supérieur).

Chapitre 7: Attaques en boîte grise par analyse de la dépendance des données (Data Dependency Gray-Box Attacks)

Le "bitslicing" est une technique courante pour obtenir une implémentation logicielle efficace d'un chiffrement, qui a également été habilitée à concevoir des implémentations ende boîte blanche avec un bon niveau de résistance en pratique. L'état de l'art des implémentations boîte blanche utilise le masquage linéaire, le masquage non linéaire, le shuffling, ainsi que l'application d'une ou plusieurs couches d'offuscation. La sécurité des implémentations repose sur les propriétés des contre-mesures utilisées, ainsi que sur l'obscurité de la conception globale (c.-à-d., le niveau d'offuscation). L'objectif principal de ces implémentations est de contrecarrer les attaques automatiques en boîte grise, ce qui contraint les adversaires potentiels à investir d'avantage d'efforts et de ressources dans des attaques de rétro-ingénierie plus coûteuses et incertaines par nature.

Dans ce chapitre, nous considérons une implémentation boîte blanche dans le paradigme d'un circuit booléen randomisé et compilé en un programme "bitslice". Nous rappelons d'abord ces contre-mesures et discutons des moyens possibles de les combiner dans des implémentations "bitsliced" en boîte blanche. Ensuite, nous analysons les différents chemins d'attaque en boîte grise et étudions leurs performances en terme de traces requises et de temps de calcul. Par la suite, nous proposons une attaque boîte grise avancée contre la cryptographie en boîte blanche qui

exploite la dépendance des données de l'implémentation cible. Cette nouvelle attaque résulte en des améliorations de complexité significatives dans différents scénario d'attaque en localisant précisément les shares ciblés dans une trace de calcul et en évitant l'explosion combinatoire standard. Nous montrons que notre approche peut efficacement casser plusieurs combinaisons de masquage linéaire et non linéaire en présence de shuffling et d'offuscation et démontrons que notre approche donne lieu à des améliorations substantielles en terme de complexité par rapport aux attaques existantes.

Chapitre 8: Attaques pratiques (Practical Attacks)

Dans ce chapitre, nous vérifions la praticabilité de nos analyses théoriques sur les techniques d'attaque exposées dans cette thèse en cassant plusieurs implémentations d'AES en boîte blanche accessibles publiquement. Plus précisément, nous effectuons d'abord les trois attaques en boîte grise analysées et présentées dans le chapitre 4 sur deux implémentations boîte blanche différentes protégées par des encodages interne qui sont censées être protégées contre les attaques DCA ; nous extrayons ensuite progressivement la clé de l'implémentation gagnante de la compétition WhibOx 2017 en utilisant une rétro-ingénierie complexe et une analyse par décodage linéaire, formalisée dans le chapitre 5 ; nous démontrons également comment notre nouvelle attaque par analyse de la dépendance des données présentée dans le chapitre 7 peut être utilisée pour casser les trois implémentations gagnantes de la compétition WhibOx 2019. De plus, nous présentons une méthodologie générale d'attaque contre les implémentations obscures en boîte blanche, qui a été suivie pour casser les implémentations gagnantes des deux compétitions WhibOx.

À notre connaissance, nous avons été la seule ou la première équipe à produire des rapports techniques sur la possibilité de casser ces implémentations et analysant les attaques en question d'un point de vue théorique. Afin de faciliter la reproduction de nos résultats, nos outils d'attaque ont été partiellement publiés.

Chapter 2

Introduction

2.1 White-Box Cryptography

2.1.1 Shades of Gray

In the classical cryptanalytic setting, the adversary faces the challenge of breaking the security of e.g., an encryption algorithm while only being able to consider the algorithm as a *black box*; she can query the box with inputs and receive the corresponding outputs. While the design of the algorithm is known, the adversary cannot observe the internal state of the algorithm, or tamper with the execution of the algorithm.

In practice, a cryptographic algorithm has to be implemented to be useful, i.e., in hardware or software. Thus, the adversary has the option of physically interacting with the encryption device. In this case, the adversary has access to the implementation-specific *side-channel information* (Kocher, 1996; Kocher et al., 1999; Coron, 1999). If an implementation is not sufficiently protected, leaking information such as the execution time or the power consumption can be used to extract secret information, e.g., encryption keys. The widespread use and success of side-channel attacks show that a cryptographer has to be very careful when operating in this *gray-box* model.

For hardware implementations, the gray-box model is often the limit of what an adversary can achieve. This is not the case for software implementations that are executed in untrusted environments. If an adversary is given full access to the execution environment of the cryptographic software, she can easily observe and manipulate the execution of the primitive, instantiated with some secret key. This setting, introduced by Chow et al. in (Chow, Eisen, Johnson, and van Oorschot, 2003), is called the *white-box* model.

Put differently, software implementations of cryptographic algorithms in the real world suffer more severe challenges than expected in their design model. In addition to the well-known *side-channel analysis* (SCA) attacks, a white-box adversary gains full access to a software implementation of a cryptographic algorithm. She could then try to extract the underlying secret key by all kinds of means. Generally, she could perform static or dynamic analysis of the controlled binary, e.g., study the logic flow of the implementation; she could arbitrarily pick the inputs for the software and observe all the runtime information, such as the addresses and values of accessed memory; she could also tamper with the implementations, e.g., by altering the control flows and injecting faults, or by interfering in the execution and exploiting leakage from erroneous outputs as in *(differential) fault analysis* (Boneh et al., 1997; Biham and Shamir, 1997).

Recently, security-critical applications, such as digital right management (DRM) systems and mobile payment services, have known a fast development and wide deployment on consumer electronic devices. Cryptographic algorithms are usually involved in these contexts to assure the confidentiality, integrity, and authenticity in several aspects. Since these applications are usually hosted on untrusted

environments and/or the users themselves might represent potential attackers, the white-box threat must then be considered by security designers and analysts. If the key embedded in an underlying implementation was extracted by the attacker (and several security features were compromised at the same time), not only could the pursued security goal be lost, but the associated business also could be threatened. For instance, an attacker could make illegal profits by selling the revealed key in a DRM application to some purchaser in the black market for a much cheaper price. It is therefore crucial to investigate the capability of a white-box adversary and the countermeasures to prevent key exposure.

2.1.2 History: A Cat-And-Mouse Game

Indeed, *"when the attacker has internal information about a cryptographic implementation, choice of implementation is the sole remaining line of defense"* (Chow, Eisen, Johnson, and van Oorschot, 2003). The two seminal papers on *white-box cryptography* (WBC), introduced by Chow et al. in 2002 (Chow, Eisen, Johnson, and Oorschot, 2002) and 2003 (Chow, Eisen, Johnson, and van Oorschot, 2003) respectively, intend to protect software implementations of standard block ciphers (DES and AES) used in DRM application against these kinds of threats. The rough idea behind their constructions is to implement a cipher as a network of precomputed and randomly encoded lookup tables, such that an adversary is confused by seemingly useless intermediate values in the memory. The randomly chosen functions applied on the lookup tables are called encodings, which can be divided into two categories: *internal* and *external* encodings. In particular, external encodings are bijections applied on the input or output of the cipher. However, the application of external encodings changes the specification of the original cipher, which is prohibitive for many use cases of white-box cryptography based on predefined (standard) cryptographic algorithms.

Soon, these techniques were broken by Billet et al. with structural cryptanalysis (Billet et al., 2004). The attack works in the principle that each of the lookup tables does not leak any key-dependent sensitive information when considered separately, but when combined together, they will reveal information on the used encodings. This observation enables the authors to simplify the encodings and entirely recover the secret key in the implementation. Since then, the competition between white-box designers and attackers has become a cat-and-mouse race. The research community has observed many different candidate constructions of white-box implementations for AES (Link and Neumann, 2005; Bringer, Hervé Chabanne, et al., 2006; Bringer, Herve Chabanne, et al., 2006; Xiao and Lai, 2009; Karroumi, 2011), as well as their subsequent break by structural analysis shortly after or even years later (Goubin, Masereel, et al., 2007; Wyseur et al., 2007; Michiels et al., 2009; De Mulder, Wyseur, et al., 2010; Tolhuizen, 2012; De Mulder, Roelse, et al., 2013b; De Mulder, Roelse, et al., 2013a; Lepoint and Rivain, 2013; Lepoint, Rivain, et al., 2014). The failure of internal encodings in white-box cryptography is essentially due to the impossibility to protect ASA, ASASA or SASAS constructions, as shown in (Biryukov and Shamir,

2001; Biryukov and Shamir, 2010; Minaud et al., 2015; Minaud et al., 2018; Derbez et al., 2018). However, the early white-box community was small and not always versatile in cryptanalysis, so they were rediscovering it slowly in ad-hoc manner for many years.

2.1.3 Theoretical Progress

As one of the key components in the protection of real-world applications, *white-box cryptography* (WBC) seeks a solution to transform a cryptographic algorithm with a given key into an obfuscated implementation. Ideally, the white-box adversary who controls the implementation should not have a significant advantage compared to the situation in which she could only access an oracle answering encryption queries (under the same key). In particular, it aims to render *key extraction* difficult –if not infeasible– to any malicious party that would gain full access to the program and/or to the execution environment.

To date, not much formalization of white-box cryptography has been put forward. Two initial works (Saxena et al., 2009; Delerablée et al., 2014) have introduced some formal white-box security notions. Specifically, Saxena et al. (Saxena et al., 2009) demonstrate how to adapt security notions in the black-box model (Barak et al., 2001) into security notions in the white-box model; while (Delerablée et al., 2014) formalizes the basic *unbreakability* property and several other useful notions: *one-wayness*, *incompressibility*, and *traceability* for symmetric ciphers.

In particular, as explained in (Delerablée et al., 2014) the minimal requirement for white-box cryptography is the resistance to key-extraction (formalized as *unbreakability*). However, this property alone is not *enough* in practice. The adversary could avoid extracting the secret key from an implementation, but instead use the implementation as a black-block (i.e., an oracle) of the underlying algorithm. Such an attack is called *code lifting* and further security features should be provided to prevent the adversary from *code lifting*. One could make the implementation large and *incompressible* to harden its extraction, make it *one-way* to prevent either encryption or decryption, or make it *traceable* to counteract unauthorized sharing. It has also for instance been suggested to rely on *external encodings* (Chow, Eisen, Johnson, and van Oorschot, 2003) and/or to bind the implementation with a user-chosen password, or some biometric data, or physical property of hardware (Alpirez Bock et al., 2020).

Despite its practical interest, no provably secure white-box implementation can be found in the literature after almost 20 years of exploration. Nevertheless, a lot of work has been done on the related area of obfuscation. Software obfuscation can be stated as the problem of creating functionally equivalent, but unintelligible programs. Unfortunately, general *virtual black-box obfuscation* has been proved impossible, in which the adversary can learn nothing from the obfuscated program except its inputs and outputs behaviors (Barak et al., 2001). A relaxation of the original notion – *indistinguishability obfuscation* ($i\mathcal{O}$) – may still be possible to achieve and has become a major open problem attracting extensive research (Garg, Gentry, Halevi,

et al., 2013; Garg, Gentry, and Halevi, 2013; Sahai and Waters, 2014; Lin, 2016; Lin, 2017; Lin and Tessaro, 2017). Literally, an $i\mathcal{O}$ obfuscator confuses the attacker by making her unable to determine the origin of the obfuscated program when there exist more than one functionally-equivalent source programs. A comprehensive overview of the problems connected to $i\mathcal{O}$ is given in (Horváth, 2015). However, $i\mathcal{O}$ does not straightly imply white-box cryptography in the sense that applying an $i\mathcal{O}$ compiler to an encryption program does not guarantee that extracting the key from the resulting program is difficult. Additionally, the current constructions of $i\mathcal{O}$ are still impractical. As a matter of fact, the question of whether unbreakability could be obtained from $i\mathcal{O}$ is still open up to now.

Another branch of theoretical research is dedicated to building new cryptographic primitives that implement incompressibility by design (Biryukov, Bouillaguet, et al., 2014; Bogdanov and Isobe, 2015; Fouque et al., 2016; Bock, Amadori, et al., 2019). The incompressibility here is also called *space-hard cryptography* or *weak white-box cryptography* in the literature. Biryukov and Perrin describe a unified framework for the hardness of a primitive in terms of any of the main efficiency metrics: speed, code size, and memory (Biryukov and Perrin, 2017). This *resource-hardness* notion is quite useful in practice since inefficiency can be a desirable feature in certain situations. For instance, memory-hardness is used in password hashing to slow down the speed of brute-force search by the attacker. In some respects, the so-called weak white-box cryptography shares some similarities with the classical white-box cryptography subject of this thesis, but they are very different in nature. Perhaps the most obvious difference is that the former primarily proposes new primitives that meet inherently specific resource-hardness, while the latter aims to protect the key in software implementations of existing standardized cryptographic algorithms.

2.2 Practical White-Box Cryptography

2.2.1 Obscurity as a Solution

As mentioned above, all current white-box implementations presented in the literature are insecure and no provably secure solution has been reported in the literature. Still, there exists an increasing industrial need for the protection of cryptographic implementations executed in untrusted environments, such as, for instance in the traditional *digital rights management* use case and the mobile payment applications running on smart devices.

In this situation, the industry is constrained to develop *home-made* solutions, to meet the growing needs for secure cryptographic software. The security of deployed proprietary solutions mainly relies on *obscurity*, i.e., the secrecy of their design. Their design is hence not publicly scrutinized, which contradicts the widely admitted Kerckhoffs's principle in cryptography. Nevertheless, the current paradigm aims to provide *practical security*, in the sense that the implementation is difficult enough to

attack so that an adversary is forced to attempt other attack vectors. The aforementioned attacks against *public* white-box implementations exploit some flaws in the underlying white-box schemes. However, some variant of known designs changing a few parameters or combining different techniques can thwart a simple application of these attacks, provided that the exact variant is kept secret. Against such an obscure implementation, the adversary would have to perform reverse engineering, which can take considerable time and effort if various layers of obfuscation have been applied.

However, the industry can only rely on their current white-box solutions (combined with other protections) from a short-term security perspective, and on other security barriers such as frequent key rotation. In this paradigm, white-box cryptography is based on regular security updates and/or short-term key and it is considered as a building block of wider security solutions.

2.2.2 Gray-Box Attacks in White-Box Context

As mentioned, obscure implementations might be insecure against a well-informed adversary, but the security of their designs can still make them practically hard to break. This is because, for instance, known structural attacks do not apply as is, and tedious reverse engineering has to be performed by skilled engineers.

In the literature, two generic gray-box attack principles have been used to break such *obscure* white-box implementations. On one hand, similarly to differential power analysis (DPA) (Kocher et al., 1999), *differential computation analysis* (DCA) (Bos et al., 2016; Bock, Bos, et al., 2019) looks for correlation between key-dependent sensitive variables and *computation traces* composed of values processed in the execution of the implementation; on the other hand, since a standard block cipher, e.g., AES, is usually inherently vulnerable to *differential fault analysis* (DFA) (Dusart et al., 2003), it can also be applied to break a majority of the public implementations (Jacob et al., 2002; Sanfelix et al., 2015).

Notably, the gray-box adversary is extremely powerful as she is *implementation agnostic* and therefore she does not need to exert expensive reverse engineering. DCA and DFA have broken some industrial solutions based on early white-box techniques mixed with code obfuscation. DFA can be slightly mitigated in the paradigm of security through obscurity, which is in line with our observations in breaking the winning implementations from the two WhibOx white-box cryptography competitions (which will be introduced later). In this thesis, we focus on the *passive* gray-box attacks which correspond to a weaker adversary and should hence be primarily addressed by white-box designers.

Differential Computational Analysis

Differential computation analysis (DCA) is a method to attack white-box implementations in a gray-box fashion. It was independently introduced by two teams: Bos et

al. at CHES 2016 and Sanfelix et al. at Black Hat Europe 2015. The two teams have demonstrated how this technique is able to recover the encryption key of several existing white-box implementations of the AES.

In principle, DCA is mainly an adaptation of *differential power analysis* (DPA) (Kocher et al., 1999) to the white-box context. It exploits the fact that the variables appearing in the computation, although in some unknown encoded form, might have a strong linear correlation with the original plain values. It works by first collecting some *computation traces*, which are composed of the runtime computed values over several executions through a dynamic instrumentation tool, such as Valgrind (Nethercote and Seward, 2007) and Intel PIN (Luk et al., 2005). One then makes a key guess and predicts the value of target key-dependent intermediate variable. Finally, the correlation between this prediction and each sample of the computation trace is calculated. The key guess with the highest peak in the obtained correlation trace is selected as the key candidate. This approach has been shown especially effective to break many publicly available (obscure) white-box implementations (Bos et al., 2016; Bock, Bos, et al., 2019).

The power of DCA is that it does not require full knowledge of the target white-box implementation. The adversary only needs to be able to observe the addresses and values of memory being accessed during the execution of the implementation. She does not need to reason about the implementation details, or modify the functionality of the code in any way – tasks that could require considerable effort. Thus, DCA is a *passive* and *gray-box* attack, which can break many practical white-box implementations with low complexity (Bos et al., 2016). Following these observations, the current white-box implementations are not even secure in a weaker attack context than the one they were designed for. Given the weak attack model of DCA and its effectiveness against practical white-box implementations, it is of importance to design (provably) secure white-box implementation against this attack.

2.2.3 Practical Countermeasures

To prevent DCA-like passive gray-box attacks, it is natural to consider classical side-channel countermeasures, i.e., *linear masking* and *shuffling*. Roughly speaking, linear masking (a.k.a. Boolean masking) splits a sensitive intermediate variable into multiple linear shares and processes them in a way that ensures the correctness of the computation while preventing sensitive information leakage to some extent. The principle of shuffling is to randomly permute the order of several independent operations (possibly including dummy operations) to increase the noise in the instantaneous leakage on a sensitive variable.

We will show that an implementation solely protected with linear masking is vulnerable to a *linear decoding analysis* (LDA) which can recover the locations of shares by solving a linear system. At Asiacrypt 2018, Biryukov and Udovenko introduced

the notion of *algebraically-secure* non-linear masking to protect white-box implementations against LDA (Biryukov and Udovenko, 2018). Non-linear masking ensures that applying any linear function to the intermediate variables of the protected implementation should not result in a *predictable* variable with probability (close to) 1. However, the non-linear masking alone might be vulnerable to the standard DCA attack. It was then suggested in (Biryukov and Udovenko, 2018) to combine linear and non-linear masking, which was conjectured to be able to counter DCA and LDA attacks at the same time. The intuition behind is two-fold: on the one hand, an algebraically-secure non-linear masking mixed with linear masking should not decrease the algebraic degree to construct a predictable value; on the other hand, the biased non-linear shares are further linearly masked and standard DCA is not able to break such a linear masking.

The state-of-the-art of white-box implementation puts to use all countermeasures mentioned above, as well as mixed with a layer of obfuscation. The security of the implementations relies on the (weak) security properties achieved by the employed countermeasures as well as on the obscurity of the overall design (including obfuscation). The main purpose of these implementations is to thwart automatic gray-box attacks, hence constraining the potential adversaries to invest costly and uncertain reverse engineering and to employ more complicated and dedicated attack techniques. Those efforts might take a long time to develop and apply, which is beneficial when combined with a moving target strategy.

Randomness plays an important role in implementing all mentioned countermeasures. It is well-known that a white-box adversary could tamper with the commutation channel between a white-box implementation and its external world, including external random sources. Hence, the effective randomness in a white-box implementation is pseudorandomness derived from the input.[1] In this thesis, each time we refer to randomness in the described countermeasures we mean pseudorandomness derived from the input.

2.3 WhibOx Competitions

2.3.1 WhibOx 2017

In this context of security through obscurity, needless to say, plenty of home-made solutions sold in the market, which are claimed to be secure based on the confidentiality of related technologies and tools, would be fragile in front of a motivated attacker. Therefore, the ECRYPT CSA project organized the WhibOx workshop (WhibOx, 2016) to fulfill public cognition of the academic progress and industrial experiences on white-box cryptography and obfuscation in 2016. At this occasion, it was suggested to organize a contest on white-box cryptography to give a playground for

[1] Pseudorandomness can not be distinguished from the uniform distribution with significant advantage any "efficient" adversary, but is reproducible by its deterministic generator and the seed used to generate it.

"researchers and practitioners to confront their (secretly designed) white-box implementations to state-of-the-art attackers" (WhibOx, 2017). One year later, the so-called WhibOx competition was launched by ECRYPT CSA as the catch the flag challenge of CHES 2017.

In a nutshell, the participants of this contest were divided into two categories:

- the *designers* who were invited to submit the source codes of their white-box implementations of AES-128 (Daemen and Rijmen, 2013) with freely chosen key, and
- the *breakers* who were challenged to reveal the hidden keys in the submitted implementations.

The participants could remain anonymous (based on a pseudonymity submission system) and they were not expected to reveal the designing or attacking techniques. The score system worked as follows: a white-box submission can accumulate $n(n+1)/2$ *strawberry* points if it survives for n days, and once it is broken, the strawberry points will decrease symmetrically down to 0. A designer gets as her final strawberry score the maximal peaking strawberries among all the challenges submitted. Similarly, a breaker gets as *banana* points the number of strawberry points of a challenge at breaking time. And she gets her final banana score as the highest banana score among all her breaks.

In order to submit a valid challenge, the implementation must fulfill several requirements, recalled in Table 2.1, which are less restrictive than that in a practical scenario. As a result, the contest successfully attracted 194 players with 94 submitted implementations which were all broken in the end for a total of 877 individual breaks. Only 13 implementations survived for more than 1 day. In particular, the winning implementation (named `Adoring Poitras`) was submitted by Biryukov and Udovenko. It survived 28 days and got broke only once by Goubin, Paillier, Rivain and myself. These results once again demonstrate that the attackers prevail in the current cat-and-mouse game. Nevertheless, many interesting designs were submitted that are worth further discussion and investigation.

Table 2.1: Requirements for a valid implementation on a reference environment mentioned in (WhibOx, 2017).

C source code	$\leq$ 50MB
compilation time	$\leq$ 100s
executable binary	$\leq$ 20MB
running memory	$\leq$ 20MB
execution time	$\leq$ 1s

2.3.2 WhibOx 2019

In light of the fact that the WhibOx 2017 improved our understanding of white-box cryptography, WhibOx 2019 was organized to further promote the public knowledge

in this paradigm of white-box cryptography. WhibOx 2019 also made new rules in order to encourage designers to submit "smaller" and "faster" implementations due to a factor of performance. The *performance score* is a parameter derived from the code size, RAM consumption and execution time, and it weighs the points score by an implementation (while being unbroken) over time. The submission server measured the following fractions for each submission at posting time:

- *Execution time*: Denote t be the average CPU time in seconds consumed by the challenge in one encryption block, the fraction for execution time is defined as $f_{\text{time}} = \frac{t}{1}$;
- *Code size*: Denote s be the size in MB of the executable after compilation, the fraction for code size is defined as $f_{\text{size}} = \frac{s}{20}$;
- *RAM usage*: Denote m be the average RAM consumption of the executable in MB, the fraction for RAM usage is defined as $f_{\text{memory}} = \frac{\text{m}}{20}$;

The performance score of the challenge is then evaluated as

$$\log_2 \left(\frac{2}{f_{\text{time}} \cdot f_{\text{size}} \cdot f_{\text{memory}}} \right).$$

Note that the denominator in each fraction is the allowed maximal limit w.r.t the measurement unit and the performance score of a challenge that meets the allowed maximal limits is equal to 1. Challenges that are either smaller, faster or less memory-consuming get a higher performance score and challenges with a higher performance score collect *strawberry* points faster.

The contest had attracted 63 players and received 27 implementations in the end. The resistance of several submissions in terms of living time was significantly improved over the first edition of the competition, which shows evidence of refinement of the underlying white-box techniques. Three implementations (all due to Biryukov and Udovenko) were still alive at the deadline of the contest.[2] Goubin, Rivain, and I succeeded in breaking one of three surviving challenges after the contest, and two others also got broken by two different teams, including a team of Rivain and myself, about one month after the competition.

2.4 Thesis Outline

This thesis mainly reorganizes my four peer-reviewed articles published in journals and conference proceedings and the presentations that I have given during my PhD. This section first lists my publications and presentations, then overviews each of the following chapters.

[2] See https://www.cryptolux.org/index.php/Whitebox_cryptography#WhibOx_2019_Competition .

2.4.1 Publications and Presentations

Publications

Andrey Bogdanov, Matthieu Rivain, Philip S. Vejre, and Junwei Wang (Apr. 2019). "Higher-Order DCA against Standard Side-Channel Countermeasures". In: *COSADE 2019: 10th International Workshop on Constructive Side-Channel Analysis and Secure Design*. Ed. by Ilia Polian and Marc Stöttinger. Vol. 11421. Lecture Notes in Computer Science. Darmstadt, Germany: Springer, Heidelberg, Germany, pp. 118–141. DOI: 10.1007/978-3-030-16350-1_8.

Louis Goubin, Pascal Paillier, Matthieu Rivain, and Junwei Wang (Apr. 2020). "How to reveal the secrets of an obscure white-box implementation". In: *Journal of Cryptographic Engineering* 10.1, pp. 49–66. DOI: 10.1007/s13389-019-00207-5.

Louis Goubin, Matthieu Rivain, and Junwei Wang (2020). "Defeating State-of-the-Art White-Box Countermeasures". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2020.3. https://tches.iacr.org/index.php/TCHES/article/view/8597, pp. 454–482. ISSN: 2569-2925. DOI: 10.13154/tches.v2020.i3.454-482.

Matthieu Rivain and Junwei Wang (2019). "Analysis and Improvement of Differential Computation Attacks against Internally-Encoded White-Box Implementations". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2019.2. https://tches.iacr.org/index.php/TCHES/article/view/7391, pp. 225–255. ISSN: 2569-2925. DOI: 10.13154/tches.v2019.i2.225-255.

Presentations

Analysis and Improvement of Differential Computation Attacks against Internally-Encoded White-Box Implementations (Aug. 26, 2019). Conference on Cryptographic Hardware and Embedded Systems 2019, Atlanta. URL: https://ches.iacr.org/2019/program.shtml (visited on 04/05/2020).

Call for Contribution: A New White-Box Analytic Tool (May 19, 2019). WhibOx 2019, Darmstadt. URL: https://www.cryptoexperts.com/whibox2019/ (visited on 04/05/2020).

Differential Computation Analysis against Internally-Encoded White-Box Implementations (May 18, 2019). WhibOx 2019, Darmstadt. URL: https://www.cryptoexperts.com/whibox2019/ (visited on 04/05/2020).

Higher-Order DCA against Standard Side-Channel Countermeasures (Apr. 4, 2019). Workshop on Constructive Side-Channel Analysis and Secure Design 2019, Darmstadt. URL: https://www.cosade.org/cosade19/program.html (visited on 04/05/2020).

How to Reveal the Secrets of an Obscure White-Box Implementation (Jan. 12, 2018). Real World Crypto 2018, Zurich. URL: https://rwc.iacr.org/2018/program.html (visited on 04/05/2020).

Recent Progress on White-Box Attacks (Dec. 13, 2018). Journée "Protection du Code et des Données", 2018, Paris Saclay. URL: http://sebastien.bardin.free.fr/2018-obfuscation-day.html (visited on 04/05/2020).

Reveal Secrets in Adoring Poitras - A generic attack on white-box cryptography (Oct. 11, 2017). ECRYPT-NET School on Correct and Secure Implementation, Crete. URL: https://hyperelliptic.org/tanja/ECRYPT-NET/schedule.html (visited on 04/05/2020).

Reveal Secrets in Adoring Poitras - A victory of reverse engineering and cryptanalysis over challenge 777 (Sept. 26, 2017). CHES 2017 (Rump Session), Taipei. URL: https://ches.2017.rump.cr.yp.to/ (visited on 04/05/2020).

White-Box Cryptogrpaphy (Oct. 16, 2018). Workshop on Physical Attacks and Design Attestation 2018, Beijing.

2.4.2 Chapter Organization

Chapter 3: Technical Background

In this chapter, we first introduce all the mathematical concepts and definitions that will be used throughout this thesis. Then, we formalize a *passive gray-box adversary model* in the white-box setting and reformulate *differential computation analysis* (DCA) in this model. Afterward, we review the linear masking, the non-linear masking and shuffling countermeasures. Last but not least, we discuss in detail the source of randomness necessary to implement these countermeasures and its design criteria.

Chapter 4: Gray-Box Attacks against Internal Encodings

Internal encoding is the very first and still commonly used white-box technique to protect block cipher implementations. It consists in representing an implementation as a network of lookup tables which are then *encoded* using randomly generated bijections (the internal encodings). The protected implementation is vulnerable to *differential computation analysis* (DCA) when this approach is implemented based on *nibble* (i.e., 4-bit wide) encodings. To thwart DCA, it has then been suggested to use wider encodings, and in particular *byte* encodings, at least to protect the outer rounds of the block cipher which are the prime targets of DCA.

In this chapter, we provide an in-depth analysis of when and why DCA works. We pinpoint the properties of the target variables and the encodings that make the attack (in)feasible. In particular, we show that DCA can break encodings wider than 4-bit, such as *byte* encodings. Additionally, we propose new DCA-like attacks inspired by side-channel analysis techniques. Specifically, we describe a *collision attack* particularly effective against the internal encoding countermeasure. We also investigate *mutual information analysis* (MIA) which naturally applies in this context. Compared to the original DCA, these attacks are also passive and they require very limited

knowledge of the attacked implementation, but they achieve significant improvements in terms of trace complexity. All the analyses of our work are experimentally backed up with various attack simulation results.

Chapter 5: Linear Decoding Analysis and Higher-Degree Extension

Modern white-box implementations employ algebraic techniques to hide the key-dependent sensitive variables during its execution. Linear encoding is one of the many considered countermeasures in industrial solutions, as well as open white-box challenges, e.g., the winning challenge WhibOx 2017 contest – `Adoring Poitras`. DCA is ineffective against linear (table) encodings unless the encodings satisfy certain necessary conditions. In the chapter, we formally describe a *linear decoding analysis* (LDA) attack to extract the key from white-box implementations, in which the target key-dependent variables are linearly encoded by a set of intermediate variables in the implementation. Then we explain how this attack can be extended to break implementations protected with higher-degree encodings.

Chapter 6: Higher-Order DCA against Masking and Shuffling

The *DCA adversary* is *passive*, and so does not exploit the full power of the white-box setting, implying that many white-box schemes are insecure even in a weaker setting than the one they were designed for. It is therefore important to develop implementations that are resistant to this attack. A natural approach when attempting to mitigate the threat of DCA attacks is to apply known countermeasures from the side-channel literature. However, it is not clear how well these countermeasures carry over to the white-box context and what level of security can be achieved by such countermeasures against a DCA adversary.

In this chapter, we investigate the approach of applying standard side-channel countermeasures such as *masking* and *shuffling* to introduce noise in the DCA traces. We show that if the source of randomness used in the implementation satisfies some necessary conditions, this approach is sufficient to achieve security against the (standard 1^{st}-order) DCA. Furthermore, we introduce *higher-order DCA*, along with an enhanced *multivariate* version, and analyze the security of the countermeasures against these attacks. We derive analytic expressions for the complexity of the attacks – backed up through extensive attack experiments – enabling a designer to quantify the security level of a masked and shuffled implementation in the (higher-order) DCA setting.

Chapter 7: Data Dependency Gray-Box Attacks

Bitslicing is a common technique to derive efficient software implementation of a cipher, which enables the design of white-box implementations with a good level of resistance in practice. The state-of-the-art of white-box implementations puts to use linear masking, non-linear masking, shuffling, and combines this with a layer of

obfuscation. The security of the implementations relies on the (weak) security properties achieved by the employed countermeasures, as well as on the obscurity of the overall design (including obfuscation). The main purpose of these implementations is to thwart automatic gray-box attacks, hence constraining the potential adversaries to invest costly and uncertain reverse engineering efforts.

In this chapter, we consider a state-of-the-art white-box implementation in the paradigm of a randomized Boolean circuit with hardcoded key represented in software as a bitsliced program. We first revisit these state-of-the-art countermeasures and discuss possible ways to combine them in bitsliced-type white-box implementations. Then we analyze the different gray-box attack paths and study their performance in terms of required traces and computation time. Afterward, we propose an advanced gray-box attack against white-box cryptography which exploits the data-dependency of the target implementation. The new data-dependency based attack achieves significant complexity improvements in several attack scenarios by precisely locating the target shares within a computation trace and avoiding the standard combinatorial explosion. We show that our approach can efficiently break several combinations of linear and non-linear masking in the presence of shuffling and obfuscation and demonstrate that our approach provides substantial complexity improvements over the existing attacks.

Chapter 8: Practical Attacks

In this chapter, we verify the practicability of the theoretical analyses and attack techniques exhibited in this thesis by recovering keys in several publicly available white-box AES implementations. Specifically, we first perform the three gray-box attacks analyzed and introduced in Chapter 4 on two different internally encoded white-box implementations which are believed to be protected against DCA-like attacks; we then gradually extract the key from the winning implementation of the WhibOx 2017 by using intricate reverse engineering and linear decoding analysis, formalized in Chapter 5; we also demonstrate how our novel data-dependency attack presented in Chapter 7 can be used to break the three winning implementations of WhibOx 2019. Additionally, we summarize a general attack methodology against obscure white-box implementations, which has been followed to analyze the winning challenges from both WhibOx contests.

To best of our knowledge, we were either the *only* or the *first* team to produce technical reports on the possibility to break these implementations by applying relevant attacks supported by theoretical analysis. To facilitate the reproduction of our results, our attack tools have been partially open-sourced.

Chapter 3

Technical Background

Organization. We first introduce all the mathematical concepts and definitions used in this thesis in Section 3.1. Then, we formalize a passive gray-box adversary model in the white-box setting and reformulate differential computation analysis by using the well-established theory of Boolean functions in this model in Section 3.2. Finally, in Section 3.3, we review linear masking, non-linear masking, and shuffling countermeasures, and we discuss in detail the source and the criteria of the randomness necessary to implement these countermeasures.

3.1 Mathematical Preliminaries

Notation. Throughout this thesis, we use the following notation. The *random variables* are denoted by uppercase Latin letters, e.g., X, while the lowercase letter x denotes a particular realization of X. We further denote *vectors* by bold symbols, e.g., $\boldsymbol{x}$. Latin letters in calligraphic format, e.g., $\mathcal{X}$ are used to denote *random distributions* and *finite sets*. $X \sim \mathcal{X}$ means the probability distribution of random variable X is $\mathcal{X}$. The *size* or the *cardinality* of a set $\mathcal{X}$ is denoted by $|\mathcal{X}|$.

For a random variable $X \sim \mathcal{X}$, $\mathrm{E}(X)$ is the *expectation* of X, and σ_X is the *standard deviation* of X, which is the square root of the *variance* of X, denoted by $\mathrm{Var}(X)$. Φ_X or $\Phi_{\mathcal{X}}$ denotes the *cumulative distribution function* (CDF) of $\mathcal{X}$ and $\Pr_X(x)$ or $\Pr_{\mathcal{X}}(x)$ denotes the *probability mass function* (PMF) of $\mathcal{X}$ evaluated on x. For two random variables X and Y, $\mathrm{Cov}(X, Y)$ is the *covariance* between X and Y.

We denote $\mathbb{N}$ the set of non-negative integers $\{0, 1, 2, \cdots\}$, and let $\mathbb{N}^+ = \mathbb{N} \setminus \{0\}$. We denote $[n]$ the set of positive integers not greater than n, i.e., $\{1, 2, \cdots, n\}$.

3.1.1 Probability and Statistics

Hypergeometric Distribution

The *hypergeometric distribution* $\mathcal{HG}(\alpha, \beta, \tau)$ is a discrete distribution describing the probability of the number of successes in τ draws without replacement where the sample population has size β and contains exactly α successes. Then, the PMF of $\mathcal{HG}(\alpha, \beta, \tau)$ is

$$\Pr_{\mathcal{HG}(\alpha,\beta,\tau)}(t) = \frac{\binom{\alpha}{t}\binom{\beta-\alpha}{\tau-t}}{\binom{\beta}{\tau}}$$

and its variance is

$$\mathrm{Var}\big(\mathcal{HG}(\alpha, \beta, \tau)\big) = \tau\frac{\alpha}{\beta}\Big(1 - \frac{\alpha}{\beta}\Big)\Big(\frac{\beta - \tau}{\beta - 1}\Big) .$$

In this thesis, we will consider a special case of hypergeometric distribution, denoted $\widetilde{\mathcal{HG}}(n)$, which is defined as

$$\widetilde{\mathcal{HG}}(n) = \mathcal{HG}\big(2^{n-1}, 2^n, 2^{n-1}\big) ,$$

for some $n \in \mathbb{N}^+$. The variance of this distribution satisfies

$$\mathrm{Var}\left(\widetilde{\mathcal{HG}}(n)\right) = \frac{2^{2n-4}}{2^n - 1} .$$

Pearson's Correlation Coefficient

Pearson's correlation coefficient is a measure of the *linear* correlation between two random variables X and Y. It is defined by the following equation

$$\mathrm{Cor}(X,\ Y) = \frac{\mathrm{Cov}(X,\ Y)}{\sigma_X \cdot \sigma_Y} = \frac{\mathrm{E}(XY) - \mathrm{E}(X)\,\mathrm{E}(Y)}{\sqrt{\mathrm{E}(X^2) - (\mathrm{E}(X))^2}\sqrt{\mathrm{E}(Y^2) - (\mathrm{E}(Y))^2}} .$$

The correlation coefficient satisfies $-1 \leq \mathrm{Cor}(X,\ Y) \leq 1$, where the lower bound is reached when X and Y are negatively linearly correlated, and the upper bound is achieved when X and Y are positively linearly correlated. A zero correlation is obtained if X and Y are *linearly* independent (which doesn't imply that X and Y are independent).

Sampling Distribution of Pearson's Correlation Coefficient

Different samples used to calculate Pearson's correlation coefficient will result in different values. The sampling distribution of a Pearson's correlation coefficient ρ can be measured by its Fisher transformation

$$z = \frac{1}{2} \cdot \ln\left(\frac{1+\rho}{1-\rho}\right) ,$$

which is approximately a normal distribution with $\mu_z = \rho$ and $\delta_z = \frac{1}{\sqrt{N-3}}$, where N denotes the number of measurements (which might be small).

To achieve a (high) success rate in a DPA/DCA attack where the peak correlation value is ρ, the number of necessary samples N can be approximated by

$$N \approx \eta_0 \cdot \left(\frac{1}{\ln\left(\frac{1+\rho}{1-\rho}\right)}\right)^2 \approx \eta \cdot \left(\frac{1}{\rho}\right)^2 , \tag{3.1}$$

where η_0 and η are small constant factors (which depend on the success rate and the key space size $|\mathcal{K}|$). The first approximation is due to several previous works on DPA/DCA (Mangard, 2004; F.-X. Standaert et al., 2006) and the second is a Taylor approximation which is sound as long as ρ is small enough (which holds in our case). Empirically, η is around 10 if the success rate is 0.9 and $|\mathcal{K}| = 256$.

3.1.2 Boolean Functions

Let $\mathbb{F}_2$ denote the field with 2 elements and let $n \in \mathbb{N}^+$. A *Boolean function* f with n variables is a function from $\mathbb{F}_2^n$ to $\mathbb{F}_2$.

The *weight* of a Boolean function f, denoted by $\mathrm{wt}(f)$, is the number of 1s in its value table, i.e., $\mathrm{wt}(f) = \big|\{x \in \mathbb{F}_2^n : f(x) = 1\}\big|$. A Boolean function f is *balanced* if $\mathrm{wt}(f) = 2^{n-1}$ (i.e., if it has as many 0 outputs as 1 outputs). The set of balanced n-variable Boolean functions is denoted by $\mathcal{B}(n)$ in this thesis. The *bias* (or *imbalance*) of a Boolean function f is defined as

$$\mathrm{B}(f) = \sum_{x \in \mathbb{F}_2^n} (-1)^{f(x)} = 2^n - 2 \cdot \mathrm{wt}(f) \,. \tag{3.2}$$

We have $\mathrm{B}(1+f) = -\mathrm{B}(f)$ and $\mathrm{B}(f) = 0$ iff $f \in \mathcal{B}(n)$.

A Boolean function f has a unique *algebraic normal form* (ANF), which is given by a set of coefficients $a_u \in \mathbb{F}_2$, $u \in \{0,1\}^n$ as

$$f(x_1, x_2, \cdots, x_n) = \sum_{u \in \{0,1\}^n} a_u x^u \,,$$

where $x^u = \prod_{i=1}^n x_i^{u_i}$.

Boolean Correlation

Let f, g be two n-variable Boolean functions and $b, b_1, b_2 \in \mathbb{F}_2$, define

$$N_b^f = \Big|\big\{x \in \mathbb{F}_2^n : f(x) = b\big\}\Big| \,,$$
$$N_{b_1 b_2}^{f,g} = \Big|\big\{x \in \mathbb{F}_2^n : f(x) = b_1, g(x) = b_2\big\}\Big| \,.$$

Then the Pearson's correlation between $f(X)$ and $g(X)$ for a uniform random input X over $\mathbb{F}_2^n$, simply denoted by $\mathrm{Cor}(f,g)$ for the sake of clarity, satisfies

$$\mathrm{Cor}(f,g) = \frac{N_{11}^{f,g} N_{00}^{f,g} - N_{10}^{f,g} N_{01}^{f,g}}{\sqrt{N_1^f N_0^f N_1^g N_0^g}} \,.$$

If $f, g \in \mathcal{B}(n)$, the above can be simplified to

$$\mathrm{Cor}(f,g) = \frac{1}{2^n} \sum_{x \in \mathbb{F}_2^n} (-1)^{f(x)+g(x)} = \frac{1}{2^n} \mathrm{B}(f+g) \,. \tag{3.3}$$

Vectorial Boolean Functions

Let $n, m \in \mathbb{N}^+$, an (n,m)-*vectorial Boolean function* (VBF) f is a function from $\mathbb{F}_2^n$ to $\mathbb{F}_2^m$. A VBF is balanced if the cardinality of $\{x \in \mathbb{F}_2^n : f(x) = y\}$ equals 2^{n-m} for every $y \in \mathbb{F}_2^m$. Given an (n,m)-VBF f, the n-variable Boolean functions $f_1, f_2, \cdots, f_m$ such that

$$f(x) = \Big(f_1(x), f_2(x), \cdots, f_m(x)\Big) \,,$$

are called the *coordinate functions* of f. If an (n,m)-VBF is balanced, then any non-zero linear combination of its coordinate functions is balanced.

3.1.3 Information Theory

Entropy

The Shannon *entropy* of a discrete random variable $X \in \mathcal{X}$ measures the *uncertainty* of X. It is defined by the following equation

$$\mathrm{H}(X) = -\sum_{x \in \mathcal{X}} \Pr(X = x) \cdot \log_2 \Pr(X = x).$$

The *joint* entropy of two discrete random variables X and Y expresses the uncertainty of the combination of variables:

$$\mathrm{H}(X, Y) = -\sum_{x \in \mathcal{X}, y \in \mathcal{Y}} \Pr(X = x, Y = y) \cdot \log_2 \Pr(X = x, Y = y).$$

The joint entropy satisfies $\mathrm{H}(X, Y) = \mathrm{H}(Y, X)$ and

$$\max\big(\mathrm{H}(X), \mathrm{H}(Y)\big) \leq \mathrm{H}(X, Y) \leq \mathrm{H}(X) + \mathrm{H}(Y), \tag{3.4}$$

where the left equality is reached if and only if (iff) Y is a deterministic function of X, and the right equality occurs iff X and Y are independent. The *conditional* entropy of a random variable X given by another variable Y expresses the uncertainty on X if Y is known:

$$\mathrm{H}(X|Y) = -\sum_{x \in \mathcal{X}, y \in \mathcal{Y}} \big(\Pr(X = x, Y = y) \cdot \log_2 \Pr(X = x|Y = y)\big),$$

which satisfies

$$0 \leq \mathrm{H}(X|Y) \leq \mathrm{H}(X)\,,$$

where both the left equality and the right equality are reached with the same conditions of Equation 3.4.

Mutual Information

The *mutual information* (MI) of two discrete random variables X and Y expresses the dependence between them. It measures the quantity of information one has obtained on X by observing Y. It is defined as

$$\mathrm{I}(X; Y) \;=\; -\sum_{x \in \mathcal{X}, y \in \mathcal{Y}} \Pr(X = x, Y = y) \cdot \log_2 \left(\frac{\Pr(X = x, Y = y)}{\Pr(X = x) \cdot \Pr(Y = y)} \right).$$

It can also be computed by the Shannon entropy

$$\begin{aligned} \mathrm{I}(X; Y) &= \mathrm{H}(X) - \mathrm{H}(X|Y) \\ &= \mathrm{H}(X) + \mathrm{H}(Y) - \mathrm{H}(X, Y) \\ &= \mathrm{H}(X, Y) + \mathrm{H}(X|Y) - \mathrm{H}(Y|X). \end{aligned}$$

The mutual information satisfies $\mathrm{I}(X;Y) = \mathrm{I}(Y;X)$ and

$$0 \leq \mathrm{I}(X;Y) \leq \min\big(\mathrm{H}(X), \mathrm{H}(Y)\big) .$$

3.2 Passive Gray-Box Adversary Model

Throughout this thesis, we consider a *passive gray-box adversary* who is capable of querying a software implementation of a cryptographic primitive with arbitrary input to obtain a *computation trace* of the execution with the help of *dynamic binary instrumentation* (DBI) tools, such as Valgrind (Nethercote and Seward, 2007) and Intel PIN (Luk et al., 2005).

The computation trace consists of any value calculated, written, or read by the implementation, and the address of any memory location read from or written to during execution. For instance, table lookups (if there exists any) are included in the computation traces. Each data point in the computation trace is further annotated with the time it occurred in the execution. Formally, each computation trace $\boldsymbol{v}$ is composed of t samples in chronological order, i.e.,

$$\boldsymbol{v} = (v_1,\ v_2,\ \cdots,\ v_t),$$

where $v_j \in \mathcal{V}$, for every $j \in [t]$ for some set $\mathcal{V}$. In the following, we consider an adversary who collects N computation traces $\big(\boldsymbol{v}^{(1)},\ \boldsymbol{v}^{(2)},\ \cdots,\ \boldsymbol{v}^{(N)}\big)$ corresponding to N inputs $\big(x^{(1)},\ x^{(2)},\ \cdots,\ x^{(N)}\big)$ of the target variable. Alternatively, the N traces can be interpreted as an $N \times t$ matrix, where a row is a computation trace $\boldsymbol{v}^{(i)}$, and a column is composed of N instances of the same intermediate variable over different computations.

In some attack scenarios, the adversary may first preprocess the traces before launching her analysis. For instance, she can remove all the constant or duplicate samples in the traces; she can also split each multi-bit sample into a tuple of bits to get a *binary* trace in which $\mathcal{V} = \mathbb{F}_2$.

When a number of traces have been collected, the adversary attempts to build a *distinguisher* D, mapping the inputs $\big(x^{(i)}\big)_i$ and the corresponding computation traces $\big(\boldsymbol{v}^{(i)}\big)_i$ to a *score vector*:

$$(\gamma_k)_{k\in\mathcal{K}} = \mathsf{D}\Big(\big(x^{(1)},\cdots,x^{(N)}\big),\ \big(\boldsymbol{v}^{(1)},\cdots,\boldsymbol{v}^{(N)}\big)\Big) .$$

A distinguisher can be built with outputs similarly. Without loss of generality, we only consider the distinguisher built with inputs in our theoretical analyses. The adversary then selects the key guess $k \in \mathcal{K}$ with the highest score γ_k as the candidate for the correct key value k^*, where $\mathcal{K}$ is the key space. Hence, the success probability

of the attack is defined as

$$p_{\text{succ}} = \Pr\left(k^* = \underset{k \in \mathcal{K}}{\operatorname{argmax}}\, \gamma_k\right).$$

where this probability is taken over any randomness supplied to the implementation (including the randomness of the inputs).

In a typical scenario, the adversary exploits that the (software) implementation leaks some information about intermediate variables involved in the execution of the cryptographic algorithm. Some of these intermediate variables depend on the plaintext and (part of) the secret key, and knowledge of such variables can, therefore, reveal the key. We denote such a key-dependent *sensitive variable* by $s = \varphi_k(x)$, where φ is a deterministic selection function, x is a public value, e.g., (part of) the plaintext, and $k \in \mathcal{K}$ is a (secret) subkey over some subkey space $\mathcal{K}$. For instance, k could be a byte of the secret key and $\mathcal{K}$ would then be $\{0,1\}^8$. Then for each key guess $k \in \mathcal{K}$, she computes a prediction of the target sensitive variable $\varphi_k(x)$.

The score for a key guess $k \in \mathcal{K}$ is calculated by measuring some form of dependency between the predictions $\varphi_k\left(x^{(i)}\right)$ and the computation traces $v^{(i)}$. Throughout this thesis, we denote respectively by k a key guess, k^* the correct key guess, and $k^\times$ a wrong key guess. In some case, for clarity, we abuse notation by skipping the parameter k in the selection function by letting $\varphi = \varphi_k$, $\varphi^* = \varphi_{k^*}$ and $\varphi^\times = \varphi_{k^\times}$. We then denote φ_i the i^{th} coordinate of φ and v_j the j^{th} sample in a trace.

3.2.1 Differential Computation Analysis Distinguisher

Here, we briefly describe the (standard first-order) DCA distinguisher used in (Bos et al., 2016) to effectively smash a range of publicly available AES and DES white-box implementations. This attack builds on the same principles as *differential power analysis* (DPA) in the classical side-channel context but uses computation traces consisting of plain values computed by the implementation during execution.

Specifically, the DCA distinguisher for a key guess k is calculated as the maximal absolute value of the correlation between the i^{th} bit of hypothesized sensitive variable $\varphi_i(X)$, for some $i \in [m]$, and each trace sample V_j, that is

$$\gamma_k^{\text{dca}} = \max_{j \in [T]} \left| \operatorname{Cor}\left(\varphi_i(X), V_j\right) \right|.$$

for some correlation measurement Cor such as Pearson's correlation coefficient. If there exists a statistical correlation between the secret variable $\varphi_i(X)$ and the values of the computation trace V_j for some index $1 \leq j \leq t$, we would expect a large absolute value of $\operatorname{Cor}(\varphi_i(X), V_j)$ for the correct prediction of the secret variables $\varphi_i^*(X)$. On the other hand, if $k \neq k^*$, we expect a low correlation between all $\varphi_i^\times(X)$ and any point in the computation trace.

In practice, the adversary does not compute the exact value of the above correlation, but an estimation of it based on N sampled computation traces

$$\left(\boldsymbol{v}^{(1)}, \boldsymbol{v}^{(2)}, \cdots, \boldsymbol{v}^{(N)}\right)$$

corresponding to random plaintexts $\left(x^{(1)}, x^{(2)}, \cdots, x^{(N)}\right)$. In the words, the score γ_k in a practical attack is defined as the following maximum correlation

$$\gamma_k = \max_j \left| \mathrm{Cor}\left(\left(v_j^{(1)}, \ldots, v_j^{(N)}\right),\ \left(\varphi_i(x^{(1)}), \ldots, \varphi_1(x^{(N)})\right) \right) \right| .$$

Moreover, she could try several different selection functions φ and any $i \in [m]$ until the correlation for one key guess can be distinguished from the others.

Although DCA can work whatever the definition space $\mathcal{V}$ of the samples in the computation trace, we consider hereafter that all the samples have been previously split into bits before computing the correlation scores in a trace preprocessing stage (i.e., we have $\mathcal{V} = \mathbb{F}_2$).

3.3 Countermeasures against Gray-Box Attacks

In this section, we revisit linear masking, non-linear masking, and shuffling countermeasures, and we discuss in detail the source and the criteria of the randomness necessary to implement these countermeasures.

3.3.1 Linear Masking

Linear masking, a.k.a. *Boolean masking* is a widely-deployed countermeasure against DPA attacks of hardware implementations (Chari, Jutla, et al., 1999; Goubin and Patarin, 1999; Rivain and Prouff, 2010). Since the DCA attack relies on the same ideas as DPA, the prospect of applying masking to secure a software implementation against DCA is promising.

To linearly mask a secret variable, it is split into several linear parts that are then processed independently. Specifically, a linear masking scheme of order $n-1$ splits each sensitive variable x occurring in a cryptographic computation into n shares satisfying

$$x = x_1 \oplus x_2 \oplus \cdots \oplus x_n \tag{3.5}$$

(where $\oplus$ denote the bitwise addition). A simple way to achieve this is by picking $x_1, \ldots, x_{n-1}$ uniformly at random (the *masks*), and setting $x_n = x \oplus x_1 \oplus \ldots \oplus x_{n-1}$ (the *masked variable*). Then, the computation must be handled on these n shares in a way that ensures the correctness of the computation while achieving some security property. Roughly speaking, one must ensure that any subset of fewer than n shares does not reveal any information about the x, i.e., is statistically independent of x. The notion is formalized in a circuit computation model as the *probing security*: an n^{th}-order probing secure circuit ensures that any observation of n wires (so-called *probes*)

can be perfectly simulated without knowledge of the sensitive variables (Ishai et al., 2003).

To compute any $\mathbb{F}_2$-linear function on a linearly masked variable, we simply compute the function on each linear share separately. Thus, the calculation of the linear components in a block cipher can be easily implemented on the masked state. Computing the non-linear components (i.e., typically the s-boxes which perform substitution in symmetric-key ciphers) is more involved but several linear masking schemes exist that achieve n^{th}-order security (see for instance (Rivain and Prouff, 2010; Coron, Prouff, et al., 2014; Coron, 2014)). A modular approach in probing security is first to design n^{th}-order probing secure *gadgets* which compute elementary operations, then to compose these gadgets in a way that preserves the n^{th}-order security of the full circuit (Barthe et al., 2016; Belad et al., 2018).

ISW Gadgets

Without loss of generality, (Ishai et al., 2003) only describes n^{th}-order masking gadgets for Boolean `NOT` and `AND` gates, which can be composed to defeat $\lfloor \frac{n}{2} \rfloor^{\text{th}}$-order probing attacks. However, they can be composed to achieve n^{th}-order probing secure circuit by carefully placing some refresh gadgets, as shown in the application on AES (Rivain and Prouff, 2010; Coron, Prouff, et al., 2014). We recall hereafter the secure `AND` gadget for linear masking in Algorithm 3.1. Note that fresh randomness is taken to ensure the security goal of a secure `AND` gadget.

Algorithm 3.1 `AND` gadget for linear masking

Input: linear sharing $(x_i)_{1\leq i\leq n}$ s.t. $\bigoplus_{1\leq i\leq n} x_i = x$, linear sharing $(y_i)_{1\leq i\leq n}$ s.t. $\bigoplus_{1\leq i\leq n} y_i = y$, randomness $(r_{i,j})_{1\leq i<j\leq n}$

Output: $(z_i)_{1\leq i\leq n}$ satisfying $\bigoplus_{1\leq i\leq n} z_i = xy$

1: **for** $i \leftarrow 1, \cdots, n$ **do**
2: **for** $j \leftarrow i+1, \cdots, n$ **do**
3: $r_{j,i} \leftarrow r_{i,j} \oplus x_i y_j \oplus x_j y_i$
4: **end for**
5: **end for**
6: **for** $i \leftarrow 1, \cdots, n$ **do**
7: $z_i \leftarrow x_i y_i$
8: **for** $j \leftarrow 1, \cdots, n$ **do**
9: **if** $j \neq i$ **then**
10: $z_i \leftarrow z_i \oplus r_{i,j}$
11: **end if**
12: **end for**
13: **end for**

A secure `NOT` gadget merely puts a `NOT` gate on the output wire of one share, and a secure `XOR` gate simply applies an `XOR` gate to each share one by one. `Refresh` gadget can be achieved by applying a secure `AND` gadget between the shares to be refreshed and $(1, 0, \cdots, 0)$.

3.3.2 Non-Linear Masking

At Asiacrypt 2018, Biryukov and Udovenko (Biryukov and Udovenko, 2018) introduced the notion of *algebraically-secure* non-linear masking to protect white-box implementations against the *linear decoding analysis* (LDA) introduced in Chapter 5 of this thesis, also independently introduced in (Biryukov and Udovenko, 2018) as *algebraic attacks*.

Roughly, a d^{th} degree algebraically-secure non-linear masking ensures that applying any function of up to degree d to the intermediate variables of the protected implementation should not compute a "predictable" variable with probability (close to) 1. By ensuring such a property, one guarantees that LDA cannot be applied to a first-degree secure implementation since any linear function of the intermediate variables is not "predictable".

Non-linear masking alone being vulnerable to higher-order DCA *per se*, Biryukov and Udovenko also suggested using a combination of non-linear masking and classic (higher-order) linear masking to resist both categories of attacks.

First-Degree Secure Non-Linear Masking

(Biryukov and Udovenko, 2018) introduces non-linear masking satisfying first-degree algebraic security. Their scheme is based on the minimalist 3-share encoding (a, b, c) for a sensitive variable x such that

$$x = ab \oplus c \,, \tag{3.6}$$

where a and b are uniform random bits and c is computed as $c = x \oplus ab$. This encoding ensures immunity against LDA by its non-linearity. To perform computation on encoded variables, the authors further define an `XOR` gadget and an `AND` gadget. Those gadgets are depicted in Algorithm 3.2 and Algorithm 3.3 respectively. For both of them, the input encodings must be *refreshed* which is performed by applying a `Refresh` gadget described in Algorithm 3.4.

3.3.3 Shuffling

In addition to the LDA attack, we will show in Section 6.2 that the higher-order DCA adversary can easily recover the key if masking is the only countermeasure. Indeed, the strength of a masked implementation is directly related to how noisy the adversary's observation of the shares is. Several approaches for introducing and

Algorithm 3.2 `XOR` gadget for minimalist quadratic masking

Input: a, b, c, d, e, f satisfying $ab \oplus c = x$ and $de \oplus f = y$, and randomness $r_a, r_b, r_c, r_d, r_e, r_f$

Output: h, i, j satisfying $hi \oplus j = x \oplus y$

1: $a, b, c \leftarrow \texttt{Refresh}(a, b, c, r_a, r_b, r_c)$
2: $d, e, f \leftarrow \texttt{Refresh}(d, e, f, r_d, r_e, r_f)$
3: $h \leftarrow a \oplus d$
4: $i \leftarrow b \oplus e$
5: $j \leftarrow c \oplus f \oplus ae \oplus bd$

Algorithm 3.3 `AND` gadget for minimalist quadratic masking

Input: a, b, c, d, e, f satisfying $ab \oplus c = x$ and $de \oplus f = y$, and randomness $r_a, r_b, r_c, r_d, r_e, r_f$

Output: h, i, j satisfying $hi \oplus j = xy$

1: $a, b, c \leftarrow \texttt{Refresh}(a, b, c, r_a, r_b, r_c)$
2: $d, e, f \leftarrow \texttt{Refresh}(d, e, f, r_d, r_e, r_f)$
3: $m_a \leftarrow bf \oplus r_c e$
4: $m_d \leftarrow ce \oplus r_f b$
5: $h \leftarrow ae \oplus r_f$
6: $i \leftarrow bd \oplus r_c$
7: $j \leftarrow am_a \oplus dm_d \oplus r_c r_f \oplus cf$

Algorithm 3.4 `Refresh` gadget for minimalist quadratic masking

Input: a, b, c satisfying $ab \oplus c = x$ and randomness r_a, r_b, r_c

Output: a', b', c' satisfying $a'b' \oplus c' = x$

1: $m_a = r_a(b \oplus r_c)$
2: $m_b = r_b(a \oplus r_c)$
3: $r'_c = m_a \oplus m_b \oplus (r_a \oplus r_c)(r_b \oplus r_c) \oplus r_c$
4: $a' = a \oplus r_a$
5: $b' = b \oplus r_b$
6: $c' = c \oplus r'_c$

increasing noise in masked implementations have been proposed and analyzed, e.g., in (Veyrat-Charvillon et al., 2012; Strobel and Paar, 2012; Coron and Kizhvatov, 2010; Rivain, Prouff, and Doget, 2009). One such approach is shuffling: instead of processing the calculations of the cipher in some fixed order, the order of execution is randomly chosen for each run of the implementation based on the value of the input (e.g., the plaintext). The situation is slightly more complicated in the white-box setting. Here, the adversary can make observations in two dimensions, namely *time* and *memory*. Even if the order of execution is shuffled in time, an adversary can choose to order the traces by the memory addresses accessed. Thus, we need to shuffle in both the time and memory dimension.

Memory Shuffle

In a masked implementation, we will typically have some state in which each element is shared as described in Section 3.3.1. The idea of the memory shuffle is to randomly rearrange the shares of the state in memory. Consider a state consisting of c elements. We assume that the shares $s_{i,j}$, $1 \leq i < c$, $1 \leq j \leq n$, are stored in an array, initially in order. That is, the implementation uses the array $(s_{1,1}, s_{2,1}, \ldots, s_{c-1,n}, s_{c,n})$. Then, we randomly pick a permutation

$$P : [1, c] \times [1, n] \to [1, c] \times [1, n] \,,$$

based on the value of the input. Note that this can be done efficiently using the Fisher-Yates shuffle (Fisher, Yates, et al., 1938). Now, instead of using the in-order array, we rearrange the array such that the implementation uses the array

$$(s_{P(1,1)}, s_{P(2,1)}, \ldots, s_{P(c-1,n)}, s_{P(c,n)}).$$

The situation is shown in Figure 3.1. Whenever the implementation needs to access share $s_{i,j}$, it simply looks up the element in position $P^{-1}(i, j)$ of the array. A similar randomization is performed for any key shares.

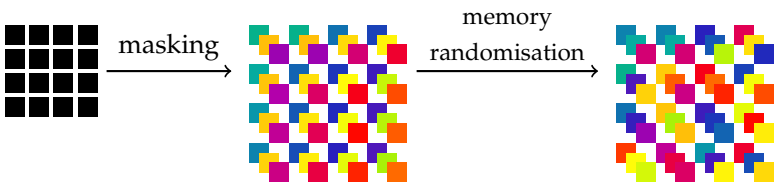

Figure 3.1: An illustration of memory shuffling applied to a second-order masked implementation. The location of each share in memory is randomized for each execution.

Time Shuffle

In a typical SPN, there will be several steps that operate on each element of the state in each round. The order of these operations is typically suggested by the

cipher designers. As an example, consider the case where we want to apply a linear operation A to each element of the state separately. Since the operation is linear, we can apply it to each share of the masked elements individually. This will normally be done in some "natural" order, e.g.,

$$A(s_{1,1}), A(s_{1,2}), A(s_{1,3}), \ldots, A(s_{c,n}) .$$

However, the exact order of execution does not matter. Thus, we can shuffle in the time dimension by randomly ordering these $c \cdot n$ operations. In general, if a set of λ independent operations exists, we can freely shuffle the order in which we process the $\lambda \cdot n$ shares. Formally, we randomly pick a permutation $Q : [1, \lambda] \times [1, n] \rightarrow [1, \lambda] \times [1, n]$. Then, when we normally would have processed share $s_{i,j}$, we instead process share $s_{Q(i,j)}$. Thus, the probability that a specific share is processed in a given step is $1/(\lambda \cdot n)$. The situation is depicted in Figure 3.2. We will denote the size of the smallest maximal set of independent operations the *shuffling degree*.

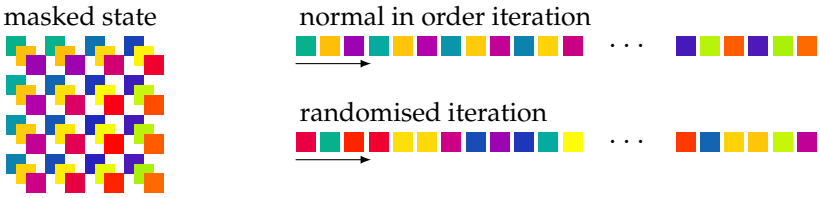

Figure 3.2: An illustration of time shuffling applied to a second-order masked implementation. The order of iteration is randomized for each execution.

Bitslicing and Shuffling

Bitslicing is a common technique to derive efficient software implementation of a cipher from its Boolean circuit representation (Biham, 1997; Rebeiro et al., 2006). The main idea is to manipulate several data slots in parallel by making the most of bitwise and/or SIMD instructions on modern CPUs. Bitslicing has been in particular applied as a strategy to design efficient implementations in the presence of linear masking (Goudarzi and Rivain, 2017; Journault and F.-X. Standaert, 2017; Goudarzi, Jean, et al., 2019; Bellizia et al., 2019). In the context of white-box cryptography, this approach has also been empowered (with additional layers of obfuscation and virtualization) to design implementations with a good level of resistance in practice. In particular, the winning implementations of the two editions of the WhibOx competition, due to Biryukov and Udovenko, were based on this principle (WhibOx, 2017; WhibOx, 2019). In a bitslice program, shuffling can be implemented both horizontally and vertically.

Horizontal vs. Vertical Shuffling. In horizontal shuffling, the data slots in a bitsliced computation are shuffled. By default, horizontal shuffling only randomizes

the computation in the dimension of memory (since all the slots are processed at the same time).

In vertical shuffling, several computation instances are done sequentially, the good one is randomly shuffled among the instances and retrieved afterward through a selection process. Vertical shuffling implements both time and memory shuffling. As an illustration in Figure 3.3, a sequential circuit $\mathcal{C}$ compromises λ sub-circuits $(\mathcal{C}_i)_{1\leq i\leq\lambda}$ that share the same inputs from a preparation stage, and from which one output will be selected as the good one after a merge process. Note that these λ sub-

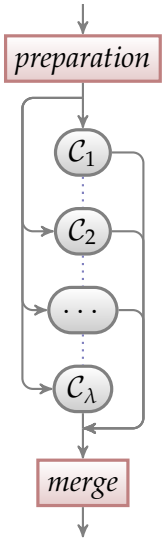

Figure 3.3: Illustration for vertical shuffling.

circuits $(\mathcal{C}_i)_{1\leq i\leq\lambda}$ might be interleaved and share some computation to increase the difficulty in analyzing the overall structure.

Horizontal and vertical shuffling can be combined in many ways. As a natural example, whenever the number of bitsliced slots is larger than the word-length of the target architecture, a full bitslicing consists of several copies of the word-length bitslicing. In this case, the desired computations manipulated in different copies are vertically shuffled both in time and memory dimensions.

Nature of the Dummy Computation. For both kinds of shuffling, the values computed in a dummy slot/instance can be of different nature:

- they are pseudorandom variables derived from the plaintext and acting like noise,
- they are genuine intermediate values corresponding to the input plaintext but for some dummy key,
- they are genuine intermediate values corresponding to right input round state but for some dummy key,

- they are redundant intermediate values of another slot (either dummy slot or right slot).

For the first three cases, the effect is to add noise in the computation trace. The second and third cases provide additional protection against DCA-type attacks by introducing dummy key candidates in the attack results. The last case might be used to defeat fault attacks but it also reduces the noise if some redundancy is used for the right slot.

3.3.4 On the Source of Randomness

A potential issue while applying linear masking, and non-linear masking, and shuffling countermeasures to the white-box context is randomness generation. In classic gray-box model, fresh randomness used in these countermeasures throughout the execution of the protected implementation which is usually provided by an external random number generator (RNG). However, such an external RNG can be easily detected and disabled, since a white-box adversary has full control over the execution environment. As a consequence, shuffled operations could be re-synchronized (e.g., using memory addresses, program counter, etc.), and/or masks could be canceled (if masked variables and corresponding masks are easily identified). Therefore, the randomness used by a white-box implementation must be pseudorandomly generated from the single available source of variation: the input plaintext. In other words, the white-box implementation should embed some kind of pseudorandom number generator (PRNG) seeded by the input plaintext.

We now (informally) state a few security properties that should be fulfilled by such a PRNG in the white-box setting:

1. *Pseudorandomness:* The output stream of the PRNG should be hard to distinguish from true randomness.

2. *Obscurity:* The design of the PRNG should be kept secret.

3. *Obfuscation:* The PRNG should be mixed with the white-box implementation so that its output stream is hard to distinguish from other intermediate variables.

The pseudorandomness property is required to ensure that the PRNG does not introduce a statistical flaw in the implemented countermeasures. It is well known that a flawed RNG can be a disaster for the security of masking, see for instance (Mangard et al., 2007). The pseudorandomness property further implies that the generated randomness is *unpredictable* provided that the obscurity property also holds. The unpredictability of the generated randomness is necessary to get DCA resistance since otherwise the countermeasures could be made ineffective. Indeed, an adversary able to predict the masks and/or the shuffling of operations could easily annihilate the effect of these countermeasures. If the PRNG design was known to

the adversary, then she could predict all the generated randomness from the plaintext. Therefore to provide unpredictability, some parts of the design must be secret, even if the obscurity concept clashes with the adage of Kerckhoffs's Principle (Kerckhoffs, 1883). Nevertheless, it seems almost impossible to provide any security if the full design is known, and we stress that this does not imply that one should forego all good cryptographic engineering practices. One could use a keyed PRNG (or PRNG with a secret initial state), but even then if the design was known to the adversary she could mount a DCA attack to recover the PRNG key and we would then face a chicken and egg problem. Alternatively, an implementation could use a known strong PRNG with some sound changes to design parameters, in order to have some confidence in its security. Another approach, which aligns with the moving target strategy, would be to have a set of different PRNG designs that are often changed. Finally, the obfuscation property is required to prevent easy detection of the PRNG output which could facilitate a DCA attack. It is for instance described in Section 8.4.3 how the generated randomness can be easily detected by switching the values of intermediate variables and checking whether this affects the final result. Such detection is an *active* attack that tampers with the execution (in the same way as *fault attacks*) and is hence out of the scope of the DCA adversary. However, it should be made difficult (in the same way as fault attacks should be made difficult) to achieve some level of resistance in practice.

In the following, we shall consider that the above security properties are satisfied by the used PRNG so that the passive gray-box adversary cannot easily remove or predict the generated randomness. We will then analyze which level of security is achievable by using linear masking, non-linear masking, and shuffling in this context.

Chapter 4

Gray-Box Attacks against Internal Encodings

Part of the results presented in this chapter have been published in (Rivain and Wang, 2019).

4.1 Introduction

The *table encoding* principle was first put forward by Chow et al. in the seminal white-box paper (Chow, Eisen, Johnson, and van Oorschot, 2003) and is still commonly used as a *countermeasure* to provide some levels of protection in a white-box context. The main principle is first to turn the implementation with some key into a sequence of connected lookup tables, then a bijection and its inverse are applied to each pair of connected tables to hide their content. Encodings can be divided into two categories: *internal* and *external* encodings. In particular, external encodings are bijections applied on the input or output of the cipher. However, the application of external encodings changes the specification of the original cipher, which is prohibitive for many use cases of white-box cryptography based on predefined (standard) cryptographic algorithms. For this reason, we only focus on internal encodings in the present thesis.

At CHES 2016, Bos et al. proposed to use *differential computation analysis* (DCA) to attack white-box implementations (Bos et al., 2016). DCA is mainly an adaptation of the *differential power analysis* (DPA) techniques (Kocher et al., 1999) to the white-box context. It exploits the fact that the variables appearing in the computation in some unknown encoded form might have a strong *linear* correlation with the original plain values.

This approach has been shown especially effective to break many publicly available white-box implementations (Bos et al., 2016) (many of which are protected with internal encodings), and it was extensively used as a white-box cryptanalytic technique in the WhibOx 2017 (WhibOx, 2017; Bock and Treff, 2020).

Although impressive, the effectiveness of DCA to break implementations protected by internal encodings was left without formal explanation in (Bos et al., 2016). This gap was addressed recently by Bock et al. who provide in (Bock, Brzuska, et al., 2018; Bock, Bos, et al., 2019) a first formal explanation of the DCA success. However their analysis is partly experimental (in particular for wrong key guesses) and it is limited to the case of linear and/or nibble encodings, which appeals for a more formal and more general analysis.

This chapter has three main contributions:

1. **Analysis and improvement of DCA.** We provide an in-depth analysis of the attack that pinpoints when and why DCA works against encoded implementations in Section 4.3. Our results include close formulas for the DCA success probability with respect to different parameters (and in particular the encoding width). This allows us to validate several formal and informal claims of (Bock, Brzuska, et al., 2018). Moreover, we show that DCA can actually break byte (and wider) encodings by targeting variables beyond the first round of the cipher.

2. **New DCA-like collision attack.** We propose a new kind of *collision attack* in the passive white-box setting where an adversary observes a computation trace

with only limited knowledge of the underlying implementation in Section 4.4. Our attack is as generic as DCA but it can defeat internal encodings with a significantly lower trace complexity. For instance, as shown in Section 8.3.4, we could break a publicly available implementation protected with byte encodings in about 60 traces with our collision attack whereas DCA requires about 1800 traces. We give some theoretical analysis of our collision attack and show that its success can be formulated as a *balls-and-containers* game.

3. **Application of *mutual information analysis* (MIA).** We suggest applying MIA to the passive white-box attack setting in Section 4.5. In particular, we propose a more efficient variant of the MIA attack against internal encodings. We also analyze the deep connection between this improved MIA and our collision attack.

All our analyses are backed up with attack simulations. Chapter 8 also includes practical attack experiments validating our analyses and illustrating the power of our new attacks. Additionally, we compare the three attacks in Section 4.6.

4.2 Internal Encodings

The principle of internal encodings was proposed by Chow et al. at SAC 2002 (Chow, Eisen, Johnson, and van Oorschot, 2003) to protect block ciphers from key extraction in the white-box context. The fundamental idea is first to turn the implementation into a network of lookup tables with a hard-coded key. Then these tables are encoded by randomly sampled bijections, called *internal encodings* in the literature.[1] More specifically, for any pair of connected tables, an invertible transformation T is applied to the output of the first table, and then the inverse of T is applied to the input of the subsequent table. A comprehensive tutorial of the internally-encoded AES implementation of Chow et al. can be found in (Muir, 2013).

For a given encoded implementation of some block cipher, any intermediate variable can be expressed as the output of an internal encoding $\varepsilon : \mathbb{F}_2^m \mapsto \mathbb{F}_2^m$ applied to some variable s. The latter can be expressed through a key-dependent function $\varphi_k : \mathbb{F}_2^n \mapsto \mathbb{F}_2^m$ of a public variable x (part of the plaintext or the ciphertext) for some subkey $k \in \mathcal{K}$. This formalism is depicted in Figure 4.1. In this chapter, we consider that φ_k is a balanced *vectorial Boolean function* (VBF) which shall be the case for a vast majority of attack scenarios. In practice, the bit-size m of the internal encodings is usually small in consideration of the code size (since storage is exponential in m). For instance, the AES implementation of Chow et al. is based on nibble encodings (i.e., $m = 4$).

Whenever the target variable is such that $n = m$, namely if the number of plaintext bits in its expression equals the bit-size of the encoding, and assuming that ε is

[1] As aforementioned, the application of external encodings is out of the scope of this work.

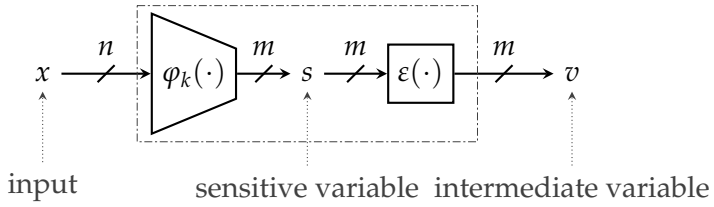

Figure 4.1: An illustration of how a sensitive intermediate variable is encoded.

a uniformly sampled bijection, then $\varepsilon \circ \varphi_k$ is a uniform random bijection which is independent of the secret key k. Consequently, there is no leakage of the underlying key k from the table itself, nor from the encoded variable $\varepsilon(s)$. This makes appear a fundamental requirement common to all kinds of DCA-like attack against encoded implementations: the target intermediate variable must be a key-dependent *non-injection*, i.e., φ_k must be such that $n > m$.

Note that although the above requirement is mandatory for a DCA-like attack to work, it does not represent a strong constraint for the attacker since such key-dependent non-injective intermediate variables naturally exist in standard block cipher designs. Indeed, for security reasons, a block cipher should have a good *diffusion*, which means that each bit of the internal state should depend on all the bits of the plaintext after a few rounds. For most block ciphers, such key-dependent non-injective variables can be found in the first couple of rounds, e.g., in the 1^{st} round of AES, a nibble of an s-box output or a byte of the MixColumn output. The former is the typical target of DCA attacks in the literature (Bos et al., 2016; Bock, Brzuska, et al., 2018), while the latter will be the case study in our experiments. Note that any byte of the AES state in the second (or a later) round is also a key-dependent non-injective variable.

4.3 Differential Computation Analysis

Differential computation analysis, proposed at CHES 2016, breaks a massive amount of publicly accessible white-box implementations (Bos et al., 2016). DCA simply consists of applying *differential power analysis* (DPA) techniques to computation traces. In this section, we provide an in-depth analysis of DCA (formally defined in Section 3.2.1) against implementations protected by internal encodings in Section 4.3.1. Specifically, we pinpoint when and why DCA works by using some properties of Boolean functions. In particular, we show that DCA can break encodings wider than 4 bits (such as byte encodings) by targeting variables deeper in the cipher than in the first (or last) round. We validate our theoretical analysis through several simulations in Section 4.3.2. In the end, several specific discussions, including a comparison with (Bock, Brzuska, et al., 2018), are conducted in Section 4.3.3.

4.3.1 Analysis of DCA against Encoded Implementations

In the following, we first introduce the formal (idealized) model which we use for our theoretical analysis. We then exhibit the distributions of the underlying correlation scores for different key guesses, and we analyze the success rate and trace complexity of DCA in this model.

Idealized Model

We perform our analysis in an idealized model in which the functions $(\varphi_k)_{k\in\mathcal{K}}$ are modeled as independent random balanced (n,m)-VBF. Using such an ideal assumption is common in symmetric cryptanalysis and it is justified in practice since the s-boxes are usually chosen in such a way that for two different $k_1, k_2 \in \mathcal{K}$, φ_{k_1} and φ_{k_2} are highly uncorrelated (as independent random functions would be). To get a formal model for the full computation trace, we further ideally assume that except the m coordinates of $\varepsilon \circ \varphi_{k^*}(X)$, the samples can be expressed as $V_j = f_j(X)$ where the f_j's are uniform random functions of $\mathcal{B}(n)$.

Note that this idealized model is used for our theoretical analysis which is then challenged and validated using attack simulations and practical attack experiments.

Distributions of Correlation Scores

Hereafter, we characterize the correlation score $\mathrm{Cor}\left(\varphi_i(X), V_j\right)$ when V is the target encoded variable i.e., $V = \varepsilon \circ \varphi^*(X)$ for a uniformly distributed plaintext variable X and a random m-bit encoding ε. According to our model, we have $\varphi = \varphi^*$ if the key guess is correct (i.e., $k = k^*$); φ and φ^* are mutually independent otherwise. We have:

$$\mathrm{Cor}\left(\varphi_i(X), V_j\right) = \mathrm{Cor}(\varphi_i, \varepsilon_j \circ \varphi^*) = \frac{1}{2^n} \cdot \mathrm{B}(\varepsilon_j \circ \varphi^* + \varphi_i).$$

Our analysis is based on the following key lemma.

Lemma 4.1. *Let $g \in \mathcal{B}(n)$. Let f be a random function uniformly sampled in $\mathcal{B}(n)$ independently of g. Then we have*

$$B(f+g) = 4 \cdot N_{00}^{f,g} - 2^n \quad \textit{with} \quad N_{00}^{f,g} \sim \widetilde{\mathcal{HG}}(n)\ ,$$

where $\widetilde{\mathcal{HG}}(n)$ is the hypergeometric distribution with parameters $(2^{n-1}, 2^n, 2^{n-1})$.

Proof. By Equation 3.2, we have $\mathrm{B}(f+g) = 2^n - 2 \cdot \mathrm{wt}(f+g)$. Since both f and g are balanced, we have $\mathrm{wt}(f+g) = 2^n - 2 \cdot N_{00}^{f,g}$ (see definition of $N_{00}^{f,g}$ in Section 3.1.2), which implies $\mathrm{B}(f+g) = 4 \cdot N_{00}^{f,g} - 2^n$. Since $N_{00}^{f,g}$ is the number of inputs x for which $f(x) = 0$ among the 2^{n-1} inputs satisfying $g(x) = 0$, we directly get $N_{00}^{f,g} \sim \widetilde{\mathcal{HG}}(n)$ by definition of the hypergeometric distribution and the uniformity of f. □

For the sake of clarity, we let Y_k be the bias $\mathrm{B}(\varepsilon_j \circ \varphi^* + \varphi_i)$ for a key guess $k \in \mathcal{K}$, and only consider Y_k in our analysis. For the correct key guess k^*, we have

$$Y_{k^*} = \mathrm{B}(\varepsilon_j \circ \varphi^* + \varphi_i^*) = 2^{n-m} \cdot \mathrm{B}(\varepsilon_j + l_i) \,,$$

where $l_i(x) = x_i$ (the i^{th} coordinate of x). Since ε is an m-bit random permutation, ε_j is randomly distributed over $\mathcal{B}(m)$. According to Lemma 4.1, we then get

$$Y_{k^*} = 2^{n-m+2} \cdot N_{00}^{\varepsilon_j, l_i} - 2^n \quad \text{with} \quad N_{00}^{\varepsilon_j, l_i} \sim \widetilde{\mathcal{HG}}(m) \,.$$

On the other hand, for an incorrect key guess $k^\times \in \mathcal{K} \setminus \{k^*\}$, we have –according to our ideal assumption– that $\varepsilon_j \circ \varphi^*$ and $\varphi_j^\times$ are randomly and independently distributed over $\mathcal{B}(n)$, which implies

$$Y_{k^\times} = \mathrm{B}(\varepsilon_j \circ \varphi^* + \varphi_i^\times) = 4 \cdot N_{00}^{\varepsilon_j \circ \varphi, \varphi_i^\times} - 2^n \quad \text{with} \quad N_{00}^{\varepsilon_j \circ \varphi, \varphi_i^\times} \sim \widetilde{\mathcal{HG}}(n) \,. \tag{4.1}$$

The mean of Y_{k^*} and $Y_{k^\times}$ are both 0, but recalling that $\mathrm{Var}\big(\widetilde{\mathcal{HG}}(\ell)\big) = \frac{2^{2\ell-4}}{2^\ell - 1}$ (which equally holds for $\ell = n$ or m), their variances satisfy

$$\mathrm{Var}(Y_{k^*}) = \frac{2^{2n}}{2^m - 1} \quad \text{and} \quad \mathrm{Var}(Y_{k^\times}) = \frac{2^{2n}}{2^n - 1} \,. \tag{4.2}$$

This makes the distributions of Y_{k^*} and $Y_{k^\times}$ easily distinguishable for practical parameters m and n (with $n > m$) as illustrated hereafter.

DCA Success Probability

For DCA to succeed, the encoding ε of the target variable must be such that the max absolute correlation over the coordinates $j \in [m]$ for the right key guess k^* is greater than the max absolute correlation over the coordinates $j \in [m]$ and the wrong key guesses $k^\times \in \mathcal{K} \setminus \{k^*\}$. We denote such an event $\mathsf{Succ}_\varepsilon$ in the following, that is

$$\mathsf{Succ}_\varepsilon \;:\; \max_j \big|\mathrm{Cor}(\varphi_i^*, \varepsilon_j \circ \varphi^*)\big| > \max_{j,k^\times} \big|\mathrm{Cor}(\varphi_i^\times, \varepsilon_j \circ \varphi^*)\big| \,.$$

Note that the probability that the above event occurs only depends on the random generation of ε, which happens during the *compilation* process of the white-box implementation. If this condition is satisfied, then the attack will succeed provided that the number of computation traces is sufficient to get enough accuracy in the correlation estimation. We first analyze the occurrence probability of $\mathsf{Succ}_\varepsilon$ and then address the trace complexity.

Simpler Case. Let us first look at the simpler case of a single j, namely the probability to get

$$\big|\mathrm{Cor}(\varphi_i^*, \varepsilon_j \circ \varphi^*)\big| > \max_{k^\times} \big|\mathrm{Cor}(\varphi_i^\times, \varepsilon_j \circ \varphi^*)\big| \,.$$

In our idealized model, this amounts to get $\left|Y_{k^*}\right| > \max_{k^\times} \left|Y_{k^\times}\right|$ where Y_{k^*} and $Y_{k^\times}$ are as defined above.

Proposition 4.1. *Under our idealized model, we have*

$$\Pr\left(|Y_{k^*}| > \max_{k^\times} |Y_{k^\times}|\right) = 2 \sum_{0 \le z < 2^{m-2}} \Pr_{\widetilde{\mathcal{HG}}(m)}(z) \cdot \left(1 - 2 \cdot \Phi_{\widetilde{\mathcal{HG}}(n)}(2^{n-m} \cdot z)\right)^{|\mathcal{K}|-1}.$$

Proof. Under our idealized model, the functions $(\varepsilon_j \circ \varphi^* + \varphi_i)_{\varphi \in \{\varphi_k : k \in \mathcal{K}\}}$ are mutually independent, hence their bias, i.e., the variables $(Y_k)_{k \in \mathcal{K}}$, are also mutually independent. Moreover, the $(Y_{k^\times})_{k^\times \in \mathcal{K} \setminus \{k^*\}}$ are identically distributed. We can then write:

$$\begin{aligned}
\Pr\left(|Y_{k^*}| > \max_{k^\times} |Y_{k^\times}|\right) &= \sum_{y=-2^n}^{2^n} \Pr\left(Y_{k^*} = y\right) \cdot \Pr\left(|y| > \max_{k^\times} |Y_{k^\times}|\right) \\
&= \sum_{y=-2^n}^{2^n} \Pr\left(Y_{k^*} = y\right) \cdot \prod_{k^\times} \Pr\left(|y| > |Y_{k^\times}|\right) \\
&= \sum_{y=-2^n}^{2^n} \Pr\left(Y_{k^*} = y\right) \cdot \Pr\left(|y| > |Y_{k^\times}|\right)^{|\mathcal{K}|-1}.
\end{aligned}$$

We further have that the distributions of Y_{k^*} and $Y_{k^\times}$ (for every $k^\times$) are both symmetric centered in 0, that is $\Pr(Y_k = y) = \Pr(Y_k = -y)$ for every k and y, which gives

$$\begin{aligned}
\Pr\left(|Y_{k^*}| > \max_{k^\times} |Y_{k^\times}|\right) &= 2 \sum_{-2^n \le y < 0} \Pr\left(Y_{k^*} = y\right) \cdot \Pr\left(|y| > |Y_{k^\times}|\right)^{|\mathcal{K}|-1} \\
&= 2 \sum_{-2^n \le y < 0} \Pr\left(Y_{k^*} = y\right) \cdot \left(1 - 2 \cdot \Phi_{Y_{k^\times}}(y)\right)^{|\mathcal{K}|-1}.
\end{aligned}$$

Using a change of variable $y = 2^{n-m+2}z - 2^n$, we have $\Pr\left(Y_{k^*} = y\right) = \Pr_{\widetilde{\mathcal{HG}}(m)}(z)$ and $\Phi_{Y_{k^\times}}(y) = \Phi_{\widetilde{\mathcal{HG}}(n)}(2^{n-m} \cdot z)$, which finally yields:

$$\Pr\left(|Y_{k^*}| > \max_{k^\times} |Y_{k^\times}|\right) = 2 \sum_{0 \le z < 2^{m-2}} \Pr_{\widetilde{\mathcal{HG}}(m)}(z) \cdot \left(1 - 2 \cdot \Phi_{\widetilde{\mathcal{HG}}(n)}(2^{n-m} \cdot z)\right)^{|\mathcal{K}|-1}.$$

□

We observe that the probability $\Pr\left(|Y_{k^*}| > \max_{k^\times} |Y_{k^\times}|\right)$ only depends on n, m and $|\mathcal{K}|$ in our idealized model. To illustrate Proposition 4.1, we plot in Figure 4.2 this probability for several values of n and m, taking $|\mathcal{K}| = 2^n$ (which would basically occur for a target function of the form $\varphi_k(x) = \varphi'(x \oplus k)$). For instance, for $n = 8, m = 4$, we have more than $\frac{1}{2}$ probability to get $|Y_{k^*}|$ greater than $|Y_{k^\times}|$ for the 255 wrong key guesses $k^\times$. This illustrates why DCA works on nibble encoding of the AES s-box. We also see that for $m = 8$, the probability $\Pr\left(|Y_{k^*}| > \max_{k^\times} |Y_{k^\times}|\right)$ increases with n and also exceeds $\frac{1}{2}$ for $n \ge 13$. This suggests that DCA can also work on byte encodings by targeting an intermediate variable depending on e.g., 16 plaintext bits.

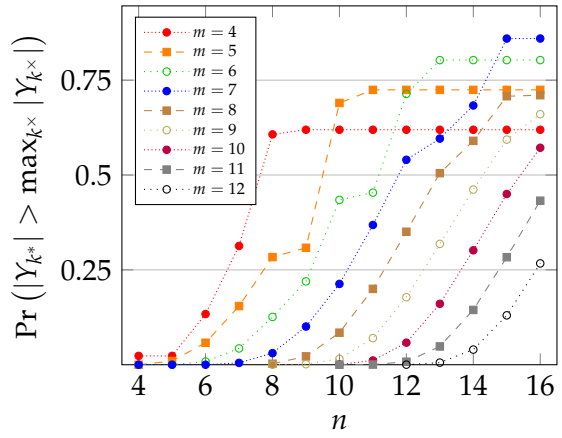

Figure 4.2: $\Pr\left(|Y_{k^*}| > \max_{k^\times} |Y_{k^\times}|\right)$ for $n \in \{4, \cdots, 16\}$ and $m \in \{4, \cdots, \max(n, 12)\}$ and $|\mathcal{K}| = 2^n$.

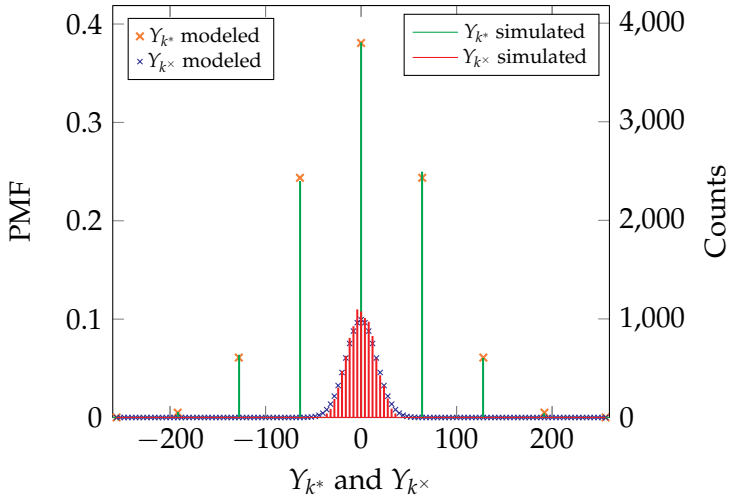

Figure 4.3: The histogram for the simulation (using 10 thousand trials) matches the theoretical analysis when $(n, m) = (8, 4)$.

Figure 4.3 further plots the distributions of Y_{k^*} and $Y_{k^\times}$ for $n = 8, m = 4$ (as well as some simulations commented below). We observe that the difference in variances makes the two distributions easily distinguishable. We further observe that whenever $Y_{k^*} \neq 0$, we have a very high probability that $|Y_{k^*}| > |Y_{k^\times}|$. This is because the values taken by Y_{k^*} are multiples of 2^{n-m+2} (which equals 64 for $n = 8, m = 4$), therefore $Y_{k^*} \neq 0$ implies $|Y_{k^*}| \geq 2^{n-m+2}$. On the other hand, the standard deviation of $Y_{k^\times}$, which is close to $2^{n/2}$ according to Equation 4.2 (i.e., around 16 for $n = 8$), might be significantly smaller than 2^{n-m+2}. More generally, if for a small constant q_α, n is chosen such that

$$q_\alpha \cdot \sigma(Y_{k^\times}) \approx q_\alpha \cdot 2^{\frac{n}{2}} \leq 2^{n-m+2} \quad \Leftrightarrow \quad n \geq 2m + 2(\log_2 q_\alpha - 2)$$

we have an overwhelming probability α that $|Y_{k^*}| > |Y_{k^\times}|$.[2] For instance, taking

[2]Here q_α is the quantile of α meaning that we have probability α that $Y_{k^\times}$ is smaller (in absolute value) than q_α times its standard deviation. In particular, α quickly converges towards 1 as q_α grows.

$n \geq 2m+2$ gives $q_\alpha \geq 8$ which (under a Gaussian approximation of $Y_{k^\times}$) implies $\alpha \geq 1 - 10^{-14}$. Consequently, choosing n slightly greater than $2m$ we get

$$\Pr\left(|Y_{k^*}| > \max_{k^\times} |Y_{k^\times}|\right) \approx 1 - \Pr(Y_{k^*} = 0) = 1 - \Pr_{\widetilde{\mathcal{HG}}(m)}(2^{m-2}) \tag{4.3}$$

where $\Pr_{\widetilde{\mathcal{HG}}(m)}(2^{m-2}) = \binom{2^{m-1}}{2^{m-2}}^2 / \binom{2^m}{2^{m-1}}$.

It can be checked from Figure 4.2 that $\Pr\left(|Y_{k^*}| > \max_{k^\times} |Y_{k^\times}|\right)$ indeed converges towards the above approximation as n grows and that the convergence is indeed achieved for $n \geq 2m+2$.

Full Success Probability. Let us now extend Proposition 4.1 to the general case of $\mathsf{Succ}_\varepsilon$ where the max is taken over all the coordinates $j \in [m]$. We shall extend our idealized model by assuming that the coordinate functions φ_j are mutually independent random functions of $\mathcal{B}(n)$. We provide a comparison of $\Pr\left(\mathsf{Succ}_\varepsilon\right)$ in this ideal setting and in a real setting in Section 4.3.2.

Proposition 4.2. *Under our idealized model, we have*

$$\Pr\left(\mathsf{Succ}_\varepsilon\right) = \sum_{0 \leq z < 2^{m-2}} \mu(z) \cdot \left(1 - 2 \cdot \Phi_{\widetilde{\mathcal{HG}}(n)}\left(2^{n-m} \cdot z\right)\right)^{m \cdot (|\mathcal{K}|-1)}$$

where

$$\mu(z) = \sum_{\ell=1}^{m} \binom{m}{\ell} \cdot \left(2 \Pr_{\widetilde{\mathcal{HG}}(m)}(z)\right)^{\ell} \cdot \left(1 - 2\Phi_{\widetilde{\mathcal{HG}}(m)}(z)\right)^{m-\ell}.$$

Proof. For any $i, j \in [m]$, and $k \in \mathcal{K}$, let $Y_k^j = \mathrm{B}(\varepsilon_j \circ \varphi^* + \varphi_i)$, then

$$\begin{aligned}
\Pr\left(\mathsf{Succ}_\varepsilon\right) &= \Pr\left(\max_{j \in [m]} |Y_{k^*}^j| > \max_{j \in [m], k^\times \in \mathcal{K} \setminus \{k^\times\}} |Y_{k^\times}^j|\right) \\
&= \sum_{y=1}^{2^n} \Pr\left(\max_{j \in [m]} |Y_{k^*}^j| = y\right) \cdot \Pr\left(y > \max_{j,k^\times} |Y_{k^*}^j|\right)
\end{aligned}$$

where

$$\begin{aligned}
\Pr\left(\max_{j \in [m]} |Y_{k^*}^j| = y\right) &= \Pr\left(\left\{j : |Y_{k^*}^j| = y\right\} \bigcap \left\{j : |Y_{k^*}^j| \leq y\right\} \neq \varnothing\right) \\
&= \sum_{\ell=1}^{m} \Pr\left(\left|\left\{j : |Y_{k^*}^j| = y\right\}\right| = \ell \bigwedge \left|\left\{j : |Y_{k^*}^j| < y\right\}\right| = m - \ell\right) \\
&= \sum_{\ell=1}^{m} \binom{m}{\ell} \cdot \Pr\left(|Y_{k^*}^j| = y\right)^{\ell} \cdot \Pr\left(|Y_{k^*}^j| < y\right)^{m-\ell} \\
&= \sum_{\ell=1}^{m} \binom{m}{\ell} \cdot \Pr\left(Y_{k^*}^j = \pm y\right)^{\ell} \cdot \Pr\left(-y < Y_{k^*}^j < y\right)^{m-\ell}
\end{aligned}$$

and

$$\begin{aligned}\Pr\left(y > \max_{j,k^\times} |Y^j_{k^\times}|\right) &= \prod_{j,k^\times} \Pr\left(y > |Y^j_{k^\times}|\right) = \Pr\left(y > |Y^j_{k^\times}|\right)^{m\cdot(|\mathcal{K}|-1)} \\ &= \left(1 - 2\cdot\Phi_{Y_{k^\times}}(-y)\right)^{m\cdot(|\mathcal{K}|-1)}\end{aligned}$$

according to the mutual independence of the $(\varphi_k)_{k\in\mathcal{K}}$ and the mutual independence of their coordinate functions in our idealized model.

Using a change of variable $y = 2^n - 2^{n-m+2}z$, we then have

$$\Pr\left(Y^j_{k^*} = \pm y\right) = 2 \Pr_{\widetilde{\mathcal{HG}}(m)}(z)$$

and

$$\Pr\left(-y < Y^j_{k^*} < y\right) = 1 - 2\Phi_{\widetilde{\mathcal{HG}}(m)}(z)$$

and

$$\Phi_{Y_{k^\times}}(-y) = \Phi_{\widetilde{\mathcal{HG}}(n)}(2^{n-m}\cdot z)\,,$$

which give

$$\Pr\left(\max_{j\in[m]} |Y^j_{k^*}| = y\right) = \sum_{\ell=1}^{m} \binom{m}{\ell} \cdot \left(2 \Pr_{\widetilde{\mathcal{HG}}(m)}(z)\right)^{\ell} \cdot \left(1 - 2\Phi_{\widetilde{\mathcal{HG}}(m)}(z)\right)^{m-\ell}$$

(denoted by $\mu(z)$ hereafter) and

$$\Pr\left(y > \max_{j,k^\times} |Y^j_{k^\times}|\right) = \left(1 - 2\cdot\Phi_{\widetilde{\mathcal{HG}}(n)}(2^{n-m}\cdot z)\right)^{m\cdot(|\mathcal{K}|-1)}\,.$$

In summary,

$$\Pr\left(\mathsf{Succ}_\varepsilon\right) = \sum_{z=1}^{2^{m-2}} \mu(z)\cdot\left(1 - 2\cdot\Phi_{\widetilde{\mathcal{HG}}(n)}(2^{n-m}\cdot z)\right)^{m\cdot(|\mathcal{K}|-1)}\,.$$

□

As above, taking $n \geq 2m+2$, we get $|Y_{k^\times}| < 2^{n-m+2}$ with overwhelming probability (for all the wrong key guesses $k^\times$ and coordinates j), and hence $\mathsf{Succ}_\varepsilon$ occurs whenever $Y_{k^*} = \mathrm{B}(\varepsilon_j \circ \varphi^* + \varphi^*_i)$ is non-zero for a single $j \in [m]$. That is

$$\Pr\left(\mathsf{Succ}_\varepsilon\right) \approx 1 - \Pr(Y_{k^*} = 0)^m = 1 - \Pr_{\widetilde{\mathcal{HG}}(m)}\left(2^{m-2}\right)^m\,. \tag{4.4}$$

This approximation is also empirically validated in Section 4.3.2.

We have analyzed the probability that an internal encoding ε makes it possible for a DCA to succeed. Let us now extend the analysis by considering the full computation trace. Under our idealized model, the latter is composed of the m coordinates

of $\varepsilon \circ \varphi^*(X)$ and of $t - m$ samples generated from fresh random functions of $\mathcal{B}(n)$. We have the following corollary of Proposition 4.2.

Corollary 4.1. *Let* Full-Succ$_\varepsilon$ *denote the event*

$$\text{Full-Succ}_\varepsilon \ : \ \max_{j \in [t]} \left| \text{Cor}(\varphi_i^*, V_j) \right| > \max_{j \in [t],\, k^\times} \left| \text{Cor}(\varphi_i^\times, V_j) \right| .$$

Under our idealized model, we have

$$\Pr\left(\text{Full-Succ}_\varepsilon\right) \geq \sum_{0 \leq z < 2^{m-2}} \mu(z) \cdot \left(1 - 2 \cdot \Phi_{\widetilde{\mathcal{HG}}(n)}\left(2^{n-m} \cdot z\right)\right)^{t \cdot (|\mathcal{K}|-1)} , \tag{4.5}$$

where $\mu(z)$ is defined as in Proposition 4.2.

The inequality in Equation 4.5 directly results from

$$\max_{j \in [t]} \left| \text{Cor}(\varphi_i^*, V_j) \right| \geq \max_{j \in [m]} \left| \text{Cor}(\varphi_i^*, \varepsilon_j \circ \varphi^*) \right| .$$

The rest of the proof is similar to the proof of Proposition 4.2.

In the above propositions (and corollary), we have exhibited the probability that the exact correlation score is greater for the right key guess k^* than for any wrong key guess $k^\times$. Although this is a necessary condition for DCA to succeed, one further needs to get some estimations of the correlation scores which are accurate enough to ensure the superiority of the right correlation peak. We analyze hereafter the number of traces necessary to meet such a practical condition.

Trace Complexity

Let us recall that $\text{Cor}(\varphi_i^*, \varepsilon_j \circ \varphi^*) = 2^{-n} \cdot Y_{k^*}$. We have seen that, with high probability (see Equation 4.4), we have $Y_{k^*} \neq 0$, and hence $|Y_{k^*}| \geq 2^{n-m+2}$, for at least one coordinate j. We consider hereafter that this event indeed occurs from which we get

$$\gamma_{k^*}^{\text{dca}} = \max_{j \in [m]} \left| \text{Cor}(\varphi_i^*, \varepsilon_j \circ \varphi^*) \right| \geq 2^{-m+2} .$$

Moreover, taking $n \geq 2m + 2$, we have seen that the variables $Y_{k^\times}$ are an order of magnitude lower than the 2^{n-m+2}, hence the correlation scores $\gamma_{k^\times}^{\text{dca}}$ are an order of magnitude lower than $\gamma_{k^*}^{\text{dca}}$. According to Equation 3.1, we get a trace complexity of $N = \mathcal{O}\left(2^{2m}\right)$ in which ρ takes $\gamma_{k^*}^{\text{dca}}$ in our context.

4.3.2 Simulations

To verify that our ideal analysis soundly captures the behavior of an actual DCA, we perform several attack simulations taking an AES s-box output as target variable, with $n = 8$, and for different encoding size $m = \{4, 5, 6, 7, 8\}$. Specifically, we look at

the distributions of $Y_{k^*} = \mathrm{B}(\varepsilon_j \circ \varphi + \varphi_i^*)$ and $Y_{k^\times} = \mathrm{B}(\varepsilon_j \circ \varphi + \varphi_i^\times)$, and the probabilities of events: $|Y_{k^*}| > \max_{k^\times} |Y_{k^\times}|$ and $\mathsf{Succ}_\varepsilon$, with $\varphi_k(x) = S^{(m)}(x \oplus k)$ where $S^{(m)}$ is the AES s-box shrunk to its m least significant bits (and hence φ_k is an $(8, m)$-VBF). For all settings, our simulation results are averaged over 10,000 trials and ε is a fresh random m-bit bijection in each trial. The full simulations are done according to the procedures depicted in Algorithm 4.1 on the next page.

As an illustration, we plot the histogram for $(n, m) = (8, 4)$ in Figure 4.3, which demonstrates that our theoretical analysis on distributions of Y_{k^*} and $Y_{k^\times}$ matches the real distributions obtained in a DCA experiment. We further compare in Table 4.1 the ideal and simulation settings for the probabilities $\Pr\left(|Y_{k^*}| > \max_{k^\times} |Y_{k^\times}|\right)$ and $\Pr\left(\mathsf{Succ}_\varepsilon\right)$. We can observe that the figures obtained through our ideal analysis match pretty well the simulation results. For instance, when $(n, m) = (8, 4)$, the probabilities of $|Y_{k^*}| > \max_{k^\times} |Y_{k^\times}|$ are about 0.6071 (ideal setting) and 0.6194 (simulation). Note that the difference is of the same order of magnitude (i.e., 10^{-2}) as the precision of the simulation results (based on $10,000$ trials).

Table 4.1: The simulation and theoretical (ideal) analysis results for $n = 8, m = \{4, 5, 6, 7, 8\}$ by using AES-128 first round s-box as the selection function.

(n, m)	$\Pr(\lvert Y_{k^*}\rvert > \max_{k^\times} \lvert Y_{k^\times}\rvert)$		$\Pr\left(\mathsf{Succ}_\varepsilon\right)$	
	ideal	**simulation**	**ideal**	**simulation**
(8,4)	0.6071	0.6194	0.9264	0.9722
(8,5)	0.2837	0.2859	0.7598	0.8032
(8,6)	0.1259	0.1281	0.3556	0.3749
(8,7)	0.0305	0.0299	0.0716	0.0723
(8,8)	0.0027	0.0021	0.0025	0.0020

We also verify the soundness of the approximation in Equation 4.3 and Equation 4.4 when one takes $n = 2m + 2$. For this purpose, we compare in Table 4.2 the probabilities of the events $|Y_{k^*}| > \max_{k^\times} |Y_{k^\times}|$ and $\mathsf{Succ}_\varepsilon$ obtained from our approximations, from our propositions in the ideal model, and from simulations. The simulations are based on 10,000 attack trials, where φ_k is defined as $\varphi_k(x) = \varphi(x \oplus k)$ for some (n, m)-VBF φ randomly picked in each trial.

Table 4.2: The simulation and theoretical (ideal) analysis results for $n = 2m + 2$ where $m = \{3, 4, 5, 6, 7\}$ by targeting at an n-bit random bijection.

(n, m)	$\Pr(\lvert Y_{k^*}\rvert > \max_{k^\times} \lvert Y_{k^\times}\rvert)$			$\Pr\left(\mathsf{Succ}_\varepsilon\right)$		
	Equation 4.3	**ideal**	**simulation**	Equation 4.4	**ideal**	**simulation**
(8,3)	0.4857	0.4857	0.4853	0.8640	0.8640	0.8828
(10,4)	0.6193	0.6193	0.6123	0.9790	0.9790	0.9736
(12,5)	0.7244	0.7244	0.7141	0.9984	0.9984	0.9960
(14,6)	0.8029	0.8029	0.8027	0.999941	0.999941	1.0000
(16,7)	0.8598	0.8598	0.8615	0.999998934	0.999998934	1.0000

Algorithm 4.1 DCASIMULATION(φ, n, m, N)

Input: A (n, m)-VBF φ and the number of trails N

Output: Biases $\left(B_{k^*,l,j}, B_{k^\times,l,j}\right)_{1\le l\le N}$, and success rates p_1 and p_2

procedure DCASIMULATION(φ, n, m, N)
  $k^*, k^\times \leftarrow_\$ \mathcal{K}$
  $i, j \leftarrow_\$ \{1, 2, \cdots, m\}$
  for $k \in \mathcal{K}$ **do**
    for $x \in \mathbb{F}_2^n$ **do**
      $T_k(x) \leftarrow \left(\varphi_k(x) \gg (i-1)\right)$ & 1
    end for
  end for
  $c_1, c_2 \leftarrow 0$
  for $l \leftarrow 1$ **to** N **do**
    $\varepsilon \leftarrow_\$$ a m-bit permutation
    for $j' \in [m], x \in \mathbb{F}_2^n$ **do**
      $E_{j'}(x) \leftarrow \left(\varepsilon \circ \varphi_{k^*}(x) \gg (j'-1)\right)$ & 1
    end for
    for $j' \in [m], k \in \mathcal{K}$ **do**
      $B_{k,l,j'} \leftarrow$ BIAS$(E_{j'}, T_k, n)$
    end for
    if $B_{k^*,l,j} > \max_{k\in\mathcal{K}\setminus\{k^*\}} B_{k,l,j}$ **then**
      $c_1 \leftarrow c_1 + 1$
    end if
    if $\max_{j'} B_{k^*,l,j'} > \max_{k^\times,j'} B_{k^\times,l,j'}$ **then**
      $c_2 \leftarrow c_2 + 1$
    end if
  end for
  $p_1 \leftarrow \frac{c_1}{N}$, $p_2 \leftarrow \frac{c_2}{N}$
  return $\left(B_{k^*,l,j}, B_{k^\times,l,j}\right)_{1\le l\le N}$, p_1, p_2
end procedure

procedure BIAS(T_1, T_2, n)
  $w \leftarrow 0$
  for $i \leftarrow 1$ **to** 2^n **do**
    $w \leftarrow w + T_1(i) \oplus T_2(i)$
  end for
  return $2^n - 2 \cdot w$
end procedure

4.3.3 Discussion

The Case of Linear Encodings

We address hereafter the case of *linear encodings*, which are encodings ε that can be expressed as

$$\varepsilon(s) = \boldsymbol{A} \cdot (s_1, \cdots, s_m)^{\mathsf{T}} \oplus \boldsymbol{b}^{\mathsf{T}} .$$

where $(s_1, \cdots, s_m)$ is the binary representation of s, $\boldsymbol{A} = \left(a_{j,\ell}\right)_{j,\ell \in [m]} \in \mathbb{F}_2^{m \times m}$ is an invertible binary matrix and $\boldsymbol{b} = \left(b_j\right)_{j \in [m]} \in \mathbb{F}_2^m$ is a binary vector. For such a linear encoding, we get $\varepsilon_j \circ \varphi^*(x) = b_j + \sum_{\ell=1}^m a_{j,\ell}\, \varphi_\ell^*(x)$ for every $j \in [m]$ and hence

$$\mathrm{Cor}(\varphi_i, \varepsilon_j \circ \varphi^*) = \frac{1}{2^n} \cdot B\Big(\varphi_i + b_j + \sum_{\ell=1}^m a_{j,\ell}\, \varphi_\ell^*\Big) .$$

Then we differentiate three cases:

- if $\varphi = \varphi^*$ (i.e., the key guess is correct) and if $a_{j,i} = 1$ and $a_{j,\ell} = 0$ for every $\ell \in [m] \setminus \{j\}$, then we have $|\mathrm{Cor}(\varphi_i, \varepsilon_j \circ \varphi^*)| = \frac{1}{2^n} \cdot |B(b_j)| = 1$;
- if $\varphi = \varphi^*$ (i.e., the key guess is correct) and if $a_{j,\ell} = 1$ for some $\ell \in [m] \setminus \{j\}$, then $\varphi_i^* + b_j + \sum_{\ell=1}^m a_{j,\ell}\, \varphi_\ell^*$ is balanced implying $|\mathrm{Cor}(\varphi_i, \varepsilon_j \circ \varphi^*)| = 0$;
- if $\varphi = \varphi^\times$ (i.e., the key guess is incorrect), under our idealized model, we have $|\mathrm{Cor}(\varphi_i, \varepsilon_j \circ \varphi^*)| = \frac{1}{2^n} |B(\varphi_i + f)|$ where $f = b_j + \sum_{\ell=1}^m a_{j,\ell}\, \varphi_\ell^*$ is a random function of $\mathcal{B}(n)$ (since f is a linear combination of the coordinates of a random balanced VBF φ^*) and independent of φ_i, namely $|\mathrm{Cor}(\varphi_i, \varepsilon_j \circ \varphi^*)|$ is distributed as the variable $\frac{1}{2^n} \cdot Y_{k^\times}$ (see Equation 4.1).

We can deduce that if the matrix $\boldsymbol{A}$ has at least one row –say the j^{th} row– of Hamming weight 1, i.e., with a single coefficient to $a_{j,i} = 1$, then for the corresponding $i \in [m]$, we have $|\mathrm{Cor}(\varphi_i, \varepsilon_j \circ \varphi^*)| = 1$ which implies that DCA (targeting the i^{th} bit of φ) will succeed with overwhelming probability. If no such row of Hamming weight 1 occurs, then the right guess correlation is indistinguishable from the wrong guess correlations and DCA fails with high probability.

There is a certain probability that a random encoding happens to be a linear encoding. This is especially likely when $m = 2, 3$. In particular, when $m = 2$, there are only 6 possible $\varepsilon_j(s_1, s_2)$ with ANF

$$s_1 + b\ ,\ s_2 + b\ ,\ \text{and}\ s_1 + s_2 + b\ ,\ \text{where}\ b \in \mathbb{F}_2 .$$

Hence, given i and j, all the possible encodings are linear, and only $s_i + b$ satisfies the condition $a_{j,i} = 1$ and $a_{j,\ell} = 0$ for every $\ell \in [m] \setminus \{j\}$. This high probability of getting a linear encoding implies that the success of DCA against encodings of size $m = 2, 3$ is less likely than for greater values of m as indicated by Proposition 4.2.

Comparison to Previous Analysis

Bock et al. (Bock, Brzuska, et al., 2018; Bock, Bos, et al., 2019) conduct an analysis to explain the ineffectiveness of linear and/or nibble encodings against DCA. In comparison, our analysis covers random (non-linear) encodings of any size m. Regarding the cases of linear encodings and (non-linear) nibble encodings, our analysis is consistent with the results of Bock et al. while providing more formal statements (under some ideal assumption) and close formulas for the success rate of DCA with respect to the attack parameters $(n, m, |\mathcal{K}|)$.

More precisely, for the case of linear encodings, our analysis of Section 4.3.3 is similar to Theorem 1 in (Bock, Brzuska, et al., 2018; Bock, Bos, et al., 2019), but the latter does not deal with the correlation scores of wrong key guesses, whereas our analysis characterizes these scores under an ideal assumption. For the case of nibble encodings, our analysis exhibits the distribution of the right guess correlation score as $\mathrm{Cor}(\varepsilon_j \circ \varphi^*, \varphi_i^*) = \frac{1}{4} \cdot N_{00}^{\varepsilon_j \circ \varphi^*, \varphi_i^*} - 1$, with $N_{00}^{\varepsilon_j \circ \varphi^*, \varphi_i^*} \sim \widetilde{HG}(4)$. This result implies in particular that the possible correlation scores for the right guess are multiples of $\frac{1}{4}$, which is the purpose of Theorem 2 in (Bock, Brzuska, et al., 2018). Besides the correlation scores for the good key guess, our analysis further characterizes the distribution for wrong guess correlation scores, whereas this distribution is considered "close to 0" in (Bock, Brzuska, et al., 2018).

Bock et al. also look at the empirical distribution of the correlation scores for the correct key guess with 10,000 attack simulations on the AES s-box protected by nibble encodings. In the considered scenario, the max correlation score is taken over the 8 predicted bits and the 4 output bits of the encoding. Tweaking Proposition 4.2 to this case, the probability to have a correlation peak of $\frac{1}{4}(4 - z)$ is given by

$$\mu(z) = \sum_{\ell=1}^{32} \binom{32}{\ell} \cdot \left(2 \Pr_{\widetilde{\mathcal{HG}}(4)}(z)\right)^{\ell} \cdot \left(1 - 2\Phi_{\widetilde{\mathcal{HG}}(4)}(z)\right)^{32-\ell} .$$

We compares the above formula to the figures given in (Bock, Brzuska, et al., 2018) in Table 4.3 which shows a good match between the two.

Table 4.3: Simulation results in (Bock, Brzuska, et al., 2018) *vs.* our formula for nibble encodings.

Score	Count (Bock, Brzuska, et al., 2018)	Probability
1	55	$\mu(0) = 0.0050$
0.75	2804	$\mu(1) = 0.2724$
0.50	7107	$\mu(2) = 0.7118$
0.25	34	$\mu(3) = 0.0108$
0	0	$\mu(4) = 0.0000$

4.4 Collision Attack

Generating and analyzing collisions in computation is a common attack technique in the side-channel context (Schramm et al., 2004; Moradi et al., 2010). In this section, we propose a new class of gray-box DCA-like collision attacks to break white-box implementations protected by internal encodings. We first give in Section 4.4.1 a formal description of our collision distinguisher within the previously introduced passive attack model. We then give a theoretical analysis of the success probability and the trace complexity of our collision attack in Section 4.4.2.

4.4.1 Collision Attack Distinguisher

Following the passive attack model introduced in Section 3.2, the adversary first collects N computation traces $\left(\boldsymbol{v}^{(i)}\right)_i$ corresponding to N inputs $\left(x^{(i)}\right)_i$ for the target function φ. Then for each pair of inputs $\left(x^{(i_1)}, x^{(i_2)}\right)$ where $i_1, i_2 \in [N], i_1 \neq i_2$, and their corresponding computation traces $\left(\boldsymbol{v}^{(i_1)}, \boldsymbol{v}^{(i_2)}\right)$, the adversary computes a *collision computation trace* (CCT):

$$\boldsymbol{w}^{(i_1,i_2)} = \left(w_1^{(i_1,i_2)}, w_2^{(i_1,i_2)}, \cdots, w_T^{(i_1,i_2)}\right),$$

with $w_j^{(i_1,i_2)} = v_j^{(i_1)} \odot v_j^{(i_2)}$ for every $j \in [t]$ where the operator $\odot$ is defined as

$$a \odot b := \begin{cases} 1 & \textbf{if } a = b, \\ 0 & \textbf{otherwise}. \end{cases}$$

Namely, the CCT for indexes (i_1, i_2) has a 1 at the j^{th} sample position iff a collision occurs between the j^{th} samples of the computation traces $\boldsymbol{v}^{(i_1)}$ and $\boldsymbol{v}^{(i_2)}$. Similarly, the *collision prediction* for a key guess $k \in \mathcal{K}$ and input values $\left(x^{(i_1)}, x^{(i_2)}\right)$ is defined as

$$\psi_k\left(x^{(i_1)}, x^{(i_2)}\right) := \varphi_k\left(x^{(i_1)}\right) \odot \varphi_k\left(x^{(i_2)}\right).$$

The collision distinguisher for a key guess k is then defined as the maximal correlation between the CCT and the corresponding collision prediction for k, i.e.,

$$\gamma_k^{\text{ca}} = \max_{j \in [t]} \text{Cor}\left(\psi_k\left(X^{(i_1)}, X^{(i_2)}\right), W_j^{(i_1,i_2)}\right).$$

As for DCA, the above correlation coefficient is estimated based on the collected samples $x^{(i)}$ and $\boldsymbol{w}^{(i,j)}$, for $i, j \in [N]$.

The soundness of our collision attack against internal encodings can be summarized with the following observation: if some sensitive variable collides for a pair of inputs, so does the corresponding encoded variable in the computation trace. Conversely, if some sensitive variable does not collide for a pair of inputs, neither does the corresponding encoded variable in the computation trace. As a consequence, there is a perfect match between the collision prediction and the target sample in the

CCT for the correct key guess (implying a correlation score to 1) whereas this should not hold for an incorrect key guess.

For a typical selection function φ, the instances $(\varphi_k)_k$ corresponding to the different key guesses behave like independent random functions. Hence, the success probability of our collision attack quickly grows with the number of collision pairs, as we analyze in more detail in Section 4.4.2.

4.4.2 Theoretical Analysis

We analyze hereafter the success rate and the trace complexity of our collision attack in the idealized model. For this purpose, we first introduce a random experiment that we shall call the *balls-and-containers game*.

The Balls-and-Containers Game

In an (α, β, γ)-balls-and-containers game experiment, a player randomly places α different balls in γ different containers of β slots each, such that, at each step, the random placement of a ball is done uniformly among the remaining free slots.[3] As an illustration, the outcome of 4 independent experiments of the $(5, 3, 6)$-balls-and-containers game is represented in Figure 4.4.

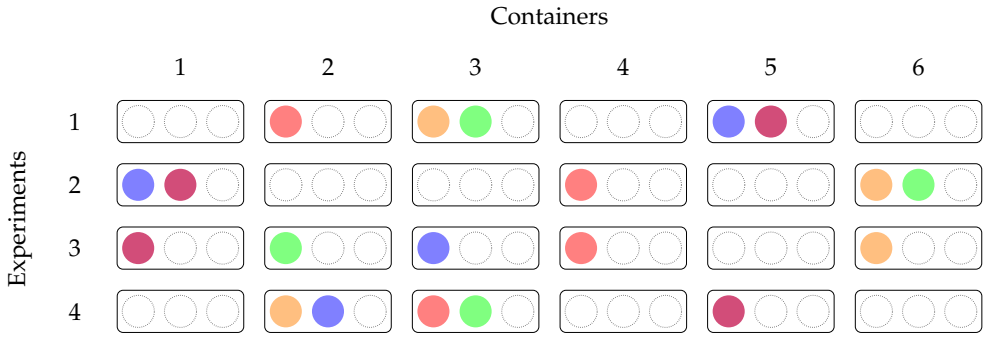

Figure 4.4: The outcome of 4 independent experiments of the $(5, 3, 6)$-balls-and-containers game, in which different balls are in different colors.

We say that a container *collides* when it contains more than one ball at the end of an experiment. For instance, the 3rd and 5th containers collide in the first experiment in Figure 4.4 whereas the other containers do not collide. We further say that the outcomes of two experiments are *isomorphic* whenever a reordering of the containers in one outcome yields a distribution of the balls among the containers which is the same as for the other outcome. For instance, the outcomes of the two first experiments in Figure 4.4 are isomorphic.

[3] In particular, a container with one or several ball(s) has a lower probability to receive a new ball than a container with only free slots.

Collision Probability. Consider an (α, β, γ)-balls-and-containers game for which $\gamma > \alpha$ (i.e., there are more containers than balls). Let Col be the event that at least one container collides. We have

$$\Pr(\neg\,\mathsf{Col}) = 1 - \Pr(\mathsf{Col}) = g(\alpha, \beta, \gamma)\,,$$

where

$$g(\alpha, \beta, \gamma) := \prod_{i=1}^{\alpha-1} \frac{\beta(\gamma - i)}{\gamma\beta - i}\,. \tag{4.6}$$

Lemma 4.2. *If $\alpha < \beta$ and $\alpha < \gamma$, we have*

$$g(\alpha, \beta, \gamma) < \exp\left(-\frac{(\alpha-2)(\alpha-1)}{2\gamma}\right)\,.$$

Proof. We have $\frac{\gamma\cdot\beta - i\cdot\beta}{\gamma\cdot\beta - i} = 1 - \frac{i - \frac{i}{\beta}}{\gamma - \frac{i}{\beta}}$ and $\frac{i-1}{\gamma-1} < \frac{i-\frac{i}{\beta}}{\gamma-\frac{i}{\beta}}$ (since $i \leq \beta$ and $i < \gamma$). We deduce

$$g(\alpha, \beta, \gamma) < \prod_{i=1}^{\alpha-1}\left(1 - \frac{i-1}{\gamma-1}\right)\,.$$

According to the mean inequality, we get

$$\begin{aligned}\sqrt[\alpha-1]{\prod_{i=1}^{\alpha-1}\left(1 - \frac{i-1}{\gamma-1}\right)} &< \frac{1}{\alpha-1}\sum_{i=1}^{\alpha-1}\left(1 - \frac{i-1}{\gamma-1}\right)\\ &= 1 - \frac{\alpha-2}{2\gamma-2} < 1 - \frac{\alpha-2}{2\gamma} < \exp\left(-\frac{\alpha-2}{2\gamma}\right)\,.\end{aligned}$$

Combining the two above formulas concludes the proof. □

Isomorphism Probability. Let us denote Iso the event that two independent experiments of the (α, β, γ)-balls-and-containers game are isomorphic. Given γ' the number of containers with at least one ball in the first experiment, we have

$$\Pr(\mathsf{Iso}) \leq \prod_{i=1}^{\gamma'-1} \frac{\beta(\gamma - i)}{\gamma\beta - i} \cdot \prod_{i=\gamma'}^{\alpha-1} \frac{\beta - 1}{\gamma\beta - i} \leq g(\alpha, \beta, \gamma)\,.$$

The above probability can be interpreted as follows. In order to have the second experiment isomorphic to the first one, the two followings shall occur:

(1) taking one ball from each γ' non-empty container in the first experiment, one must get that these γ' balls are placed in different containers in the second experiment;

(2) each of the remaining balls must end in a specific container (with at most $\beta - 1$ free slots) to satisfy the isomorphic property.

The first inequality comes from the fact that there might be less than $\beta - 1$ remaining free slots in a container for the placing of the remaining balls. The second inequality holds by definition of g (see Equation 4.6) and from $\beta(\gamma - i) > \beta - 1$ (since $\gamma > \alpha$).

Collision Attack Success Probability

We analyze our collision attack under an idealized model as the one considered for our analysis of DCA (see Section 4.3.1). In particular, the functions $(\varphi_k)_{k\in\mathcal{K}}$ are assumed to be mutually independent random balanced (n,m)-VBFs. Unlike DCA, the collision attack does not split the samples in the computation trace into bits. We hence consider that the definition space of the V_j's matches the encoding definition space, i.e., $\mathcal{V} = \mathbb{F}_{2^m}$. For some $j^* \in [T]$, we have $V_{j^*} = \varepsilon \circ \varphi^*(X)$, for the other $j \in [T] \setminus \{j^*\}$, we ideally assume that the samples can be expressed as $V_j = f_j(X)$ where the f_j's are uniform random balanced (n,m)-VBFs.

We first consider the success event

$$\mathsf{Succ} \ : \ \mathrm{Cor}\left(\psi_{k^*}\left(X^{(i_1)}, X^{(i_2)}\right), W_{j^*}^{(i_1,i_2)}\right) > \max_{k^\times} \mathrm{Cor}\left(\psi_{k^\times}\left(X^{(i_1)}, X^{(i_2)}\right), W_{j^*}^{(i_1,i_2)}\right),$$

i.e., the correlation is maximal for the correct key guess at the right sample index j^*. Note that for j^*, we have

$$W_{j^*}^{(i_1,i_2)} = \varepsilon \circ \varphi^*(X^{(i_1)}) \odot \varepsilon \circ \varphi^*(X^{(i_2)}) = \varphi^*(X^{(i_1)}) \odot \varphi^*(X^{(i_2)}) = \psi_{k^*}\left(X^{(i_1)}, X^{(i_2)}\right).$$

For some given set of inputs $(x^{(i)})_{i\in N}$, the above success event relies on two events E_1 and E_2, with $\mathsf{Succ} = E_1 \cap E_2$, which are defined as

$$E_1 \ : \ \exists\,(i,j),\ 1 \le i < j \le N,\ \text{s.t. } \varphi^*(x^{(i)}) = \varphi^*(x^{(j)}),$$

and

$$E_2 \ : \ \forall\, k^\times \in \mathcal{K} \setminus \{k^*\},\ \exists\,(i,j),\ 1 \le i < j \le N,\ \text{s.t. } \psi_{k^*}(x^{(i)}, x^{(j)}) \neq \psi_{k^\times}(x^{(i)}, x^{(j)}).$$

The event E_1 ensures that the collision predictions $\psi_{k^*}\left(x^{(i)}, x^{(j)}\right)$ are not all equal to zero for the right key guess, which must hold so that the correlation score for k^* is well defined.[4] The event E_2 ensures that for all the wrong key guesses $k^\times \neq k^*$, $(\psi_{k^\times}(x^{(i)}, x^{(j)}))_{i,j}$ does not perfectly match $(\psi_{k^*}(x^{(i)}, x^{(j)}))_{i,j}$, which implies a correlation score strictly lower than 1.

Assuming that we have $N < 2^m$, we can express the collision attack success in terms of balls-and-containers game experiments, by considering

- the inputs $(x^{(i)})_{i\in N}$ as N different balls,

[4] In principle, we should also ensure that the collision predictions for the right key guess are not all equal to one, i.e., the inputs $(x^{(i)})_{i\in N}$ do not all map to the same output through φ_{k^*}, but this shall occur with overwhelming probability so we neglect this requirement.

- the output values of φ as 2^m different containers,
- the number of preimages of a given output through φ as the 2^{n-m} slots in each container.

Then each key guess k gives rise to an (α, β, γ)-balls-and-containers game experiment with $\alpha = N$, $\beta = 2^{n-m}$, $\gamma = 2^m$ where the randomness of φ_k acts as a random placement of the inputs $(x_i)_{i \in N}$ in the 2^m output values with a maximum of 2^{n-m} slots per output value (which results from the balanceness of φ_k). The mutual independence of the $(\varphi_k)_{k \in \mathcal{K}}$ implies the mutual independence of the balls-and-containers game experiments.

The event E_1 then holds if at least one container collides in the experiment corresponding to k^*, i.e.,

$$\Pr(E_1) = \Pr(\mathsf{Col}) = 1 - g(N, 2^{n-m}, 2^m) \,. \tag{4.7}$$

On the other hand, the event E_2 holds if none of the experiments for $k^\times \in \mathcal{K} \setminus \{k^*\}$ is isomorphic to the experiment for k^*. The mutual independence of these experiments implies

$$\Pr(E_2) = \big(1 - \Pr(\mathsf{Iso})\big)^{|\mathcal{K}|-1} \geq \big(1 - g(N, 2^{n-m}, 2^m)\big)^{|\mathcal{K}|-1} \,. \tag{4.8}$$

Proposition 4.3. *Under our idealized model, we have*

$$\Pr(\mathsf{Succ}) \geq \big(1 - g(N, 2^{n-m}, 2^m)\big)^{|\mathcal{K}|} \geq 1 - |\mathcal{K}| \cdot \exp\left(-\frac{(N-2)(N-1)}{2^{m+1}}\right) \,.$$

The proposition is a direct consequence of Equation 4.7 and Equation 4.8 (first inequality), and Lemma 4.2 (second inequality).

Let us now extend the analysis by considering the full computation trace. Under our idealized model, the latter is composed of $\varepsilon \circ \varphi^*(X)$ and of $t-1$ samples generated from fresh random balanced (n, m)-VBFs. We have the following corollary of Proposition 4.3.

Corollary 4.2. *Let us denote* Full-Succ *the success event*

$$\mathsf{Full\text{-}Succ} \;:\; \max_{j \in [t]} \mathrm{Cor}\left(\psi_{k^*}\big(X^{(i_1)}, X^{(i_2)}\big), W_j^{(i_1,i_2)}\right) > \max_{j \in [t], k^\times} \mathrm{Cor}\left(\psi_{k^\times}\big(X^{(i_1)}, X^{(i_2)}\big), W_j^{(i_1,i_2)}\right) .$$

Under our idealized model, we have

$$\begin{aligned} \Pr(\mathsf{Full\text{-}Succ}) &\geq \big(1 - g(N, 2^{n-m}, 2^m)\big)^{t \cdot |\mathcal{K}|} \\ &\geq 1 - t \cdot |\mathcal{K}| \cdot \exp\left(-\frac{(N-2)(N-1)}{2^{m+1}}\right) \,. \end{aligned} \tag{4.9}$$

Corollary 4.2 is a straightforward extension of Proposition 4.3 where $(|\mathcal{K}|-1)+1$ in the exponent is replaced by $t\cdot(|\mathcal{K}|-1)+1$ which directly implies the above inequality.

Trace Complexity

From the above analysis, we can easily deduce the trace complexity of our collision attack. Let c be some parameter such that one wants to achieve a success probability $1-10^{-c}$. By Corollary 4.2, taking

$$N=\sqrt{2^{m+1}\left(c\cdot\ln 10+\ln t+\ln|\mathcal{K}|\right)}+1\,, \tag{4.10}$$

implies $\Pr(\text{Full-Succ})\geq 1-10^{-c}$. Given a (high) success probability, a trace size and a key space, the number of required computation traces is hence $N=\Theta\left(2^{\frac{m}{2}}\right)$, which is a significant improvement over DCA for which we have $N=\mathcal{O}\left(2^{2m}\right)$.

In order to illustrate our analysis, Figure 4.5 plots the lower bound on the success probability (Equation 4.9) for $n=16$, $m=8$, $|\mathcal{K}|=2^n$, and $t\in\{1,10^3,10^6\}$. We see that multiplying the size of the computation trace by a factor of 1000 only implies a small gap (less than 20) in the number of required traces. In order to illustrate the tightness of the bounds, we further plot the lower bound in the middle in Equation 4.9, i.e., $(1-g(N,2^{n-m},2^m))^{t\cdot|\mathcal{K}|}$, as well as the lower bound obtained by a straight application of Lemma 4.2. We observe that our explicit lower bound only implies a gap of 5 in the number of required computation traces, which is fairly tight.

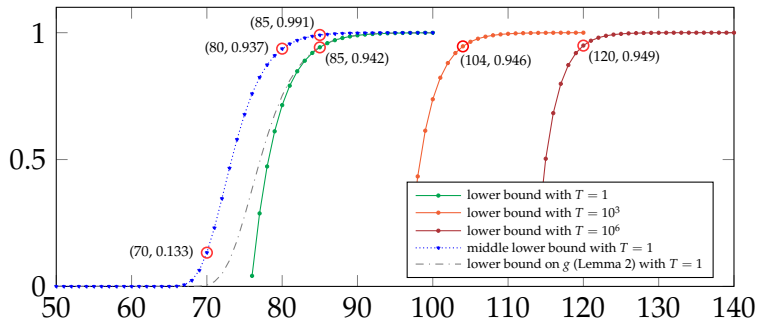

Figure 4.5: Success probability lower bound (Equation 4.9) over an increasing N for $n=16$, $m=8$, and $|\mathcal{K}|=2^n$.

4.5 Mutual Information Analysis

Mutual Information Analysis (MIA) was introduced in the side-channel context for an adversary that has very limited knowledge about the leakage distribution and how it relates to computed data (Gierlichs et al., 2008; Batina et al., 2011). In particular, MIA can deal with any kind of –possibly uncommon, odd, or complex– leakage

function. It therefore naturally applies in the white-box context to attack implementations protected with internal encodings since the latter can be thought of as particular cases of –especially complex– leakage functions. We first recall the MIA distinguisher in Section 4.5.1, then we give a brief analysis of its behavior in the considered white-box setting in Section 4.5.2.

4.5.1 MIA Distinguisher

The MIA distinguisher for a key guess k is calculated as the maximal mutual information between the prediction $\varphi_k(X)$ and each trace sample V_j, that is

$$\gamma_k^{\text{mia}} = \max_{j \in [t]} \mathrm{I}\left(\varphi_k(X); V_j\right) \ .$$

The basic notions of information theory are recalled in Section 3.1.3. Note that unlike the side-channel context in which evaluating the mutual information usually involves complex PDF estimation methods, we are only dealing with discrete variables here which makes the practical evaluation simpler.

4.5.2 Analysis and Improvement

In practice, the adversary computes the MIA distinguisher based on sample values. In the following, we shall use denote $\hat{\mathrm{I}}$ and $\hat{\mathrm{H}}$ the sample versions of the mutual information and the entropy which are computed based on a uniform random selection of the inputs $(x^{(i)})_{i \in [N]}$.

Let j^* be the sample index such that $V_{j^*} = \varepsilon \circ \varphi_{k^*}(X)$. We have

$$\hat{\mathrm{I}}(\varphi(X)\ ;\ V_{j^*}) = \hat{\mathrm{I}}(\varphi\ ;\ \varepsilon \circ \varphi^*) = \hat{\mathrm{I}}(\varphi\ ;\ \varphi^*)\ ,$$

where we drop the argument X in φ and φ^* for the sake of clarity, and where the last equality holds by the bijectivity of ε. Let us look at the success event that, for the right sample index j^*, the mutual information score is the greatest for the correct key guess, that is

$$\mathsf{Succ} \ : \ \hat{\mathrm{I}}(\varphi^*\ ;\ \varphi^*) \geq \max_{k^\times} \hat{\mathrm{I}}(\varphi^\times\ ;\ \varphi^*)\ .$$

For the correct key guess, we have

$$\hat{\mathrm{I}}(\varphi^*\ ;\ \varphi^*) = \hat{\mathrm{H}}(\varphi^*) \xrightarrow{N \to \infty} \mathrm{H}(\varphi^*) = m\ .$$

On the other hand, for an incorrect key guess, we have

$$\hat{\mathrm{I}}(\varphi^\times\ ;\ \varphi^*) = \hat{\mathrm{H}}(\varphi^*) - \hat{\mathrm{H}}(\varphi^\times | \varphi^*)\ .$$

We hence deduce that Succ occurs if and only if $\hat{\mathrm{H}}(\varphi^\times|\varphi^*) \neq 0$ for every $k^\times \in \mathcal{K} \setminus \{k^*\}$, which is equivalent to the following event

$$E \ : \ \forall\, k^\times \in \mathcal{K} \setminus \{k^*\},\ \exists\,(i,j),\ 1 \leq i < j \leq N, \\ \text{s.t. } \varphi^*(x^{(i)}) = \varphi^*(x^{(j)}) \text{ and } \varphi^\times(x^{(i)}) \neq \varphi^\times(x^{(j)})\,.$$

Note that this event is close but different from the intersection $E_1 \cap E_2$ analyzed for the collision attack. In particular, we have $E \Rightarrow E_1 \cap E_2$ but $E_1 \cap E_2 \not\Rightarrow E$. In other words, our collision attack succeeds whenever MIA succeeds but the converse is not true.

Nevertheless, the event E has still a high probability to occur when the parameters are chosen as in our collision attack. For instance, let $n = 2m$ and $N = \Theta\left(2^{\frac{m}{2}}\right)$. According to the birthday paradox, we have a high probability to get a small number q of collisions $\varphi^*(x^{(i)}) = \varphi^*(x^{(j)})$. The event E does not occur, if for some $k^\times$ we also have $\varphi^\times(x^{(i)}) = \varphi^\times(x^{(j)})$ for all these q collisions, which happens with probability lower than $\left(\frac{2^m-1}{2^n-N}\right)^q \leq \frac{1}{2^{qm}}$. We thus obtain a high probability $\Pr(E) \geq 1 - \frac{|\mathcal{K}|}{2^{qm}}$ and hence we also have $N = \mathcal{O}\left(2^{\frac{m}{2}}\right)$ for MIA.

Improved MIA

We show hereafter that a simple improvement of MIA can make it as successful as our collision attack. We know that for the right key guess we have $\hat{\mathrm{I}}(\varphi^*;\varphi^*) = \hat{\mathrm{H}}(\varphi^*)$. So for a guess k, we know that if $\hat{\mathrm{I}}(\varphi;\varphi^*) \neq \hat{\mathrm{H}}(\varphi)$ then $k \neq k^*$. Our improvement simply consists in setting the score associated to k to 0 whenever such an inequality occurs, that is

$$\gamma_k^{\mathrm{mia}} = \begin{cases} \mathrm{H}(\varphi_k(X)) & \text{if } \max_{j\in[t]} \mathrm{I}(\varphi_k(X);V_j) = \mathrm{H}(\varphi_k(X)), \\ 0 & \text{otherwise.} \end{cases}$$

For this new distinguisher, we still have $\hat{\mathrm{H}}(\varphi^*)$ as a score for the right key guess. But for a wrong key guess, we get a zero score whenever

$$\hat{\mathrm{I}}(\varphi^\times;\varphi^*) = \hat{\mathrm{H}}(\varphi^\times) - \hat{\mathrm{H}}(\varphi^*|\varphi^\times) \neq \hat{\mathrm{H}}(\varphi^\times) \ \Leftrightarrow \ \hat{\mathrm{H}}(\varphi^*|\varphi^\times) \neq 0\,.$$

Therefore, for this improved distinguisher, the success occurs if and only if for every $k^\times \in \mathcal{K} \setminus \{k^*\}$ we have either $\hat{\mathrm{H}}(\varphi^\times|\varphi^*) \neq 0$ (as for standard MIA) or $\hat{\mathrm{H}}(\varphi^*|\varphi^\times) \neq 0$. Equivalently, the failure occurs if for one $k^\times \in \mathcal{K} \setminus \{k^*\}$ we have $\hat{\mathrm{H}}(\varphi^\times|\varphi^*) = 0$ and $\hat{\mathrm{H}}(\varphi^*|\varphi^\times) = 0$. This failure event holds if the distribution of the inputs $(x^{(i)})_{i\in[N]}$ among the different output values of φ are *isomorphic* for $k^\times$ and k^* in the sense of the balls-and-containers game introduced in Section 4.4.2. We deduce that the success of the improved MIA is equivalent to the event E_2 considered in the analysis of the collision attack. In other words, the improved MIA succeeds if and only if our collision attack succeeds, except that the improved MIA does not need the event

E_1 to occur. We then have similar success probabilities than in Proposition 4.3 where $|\mathcal{K}|$ can be replaced by $|\mathcal{K}| - 1$. However, this difference has a negligible impact on the number of traces (which clearly appears while looking at Equation 4.10) and one can consider that the two attacks have similar trace complexities in our idealized model.

4.6 Comparison

In this section, we compare the three attack techniques analyzed in this chapter. First of all, they all belong to the same family of *computation analysis* attacks that record computed values during the execution of white-box implementation and then apply side-channel attack techniques. In particular, these attacks are *gray-box* in the sense that they can work with only limited knowledge of the implementation and of the computation traces (e.g., one does not need to know the implementation details or the location of the target variables in the traces).

Another apparent similarity is that these three techniques require a *non-injective* property of the target variable. However, we stress that this requirement is intrinsic to collision and mutual information attacks, as already noted in (Prouff and Rivain, 2009; Batina et al., 2011) for the latter, whereas it is not for DCA. Indeed the necessity of targeting non-injection is implied by the context, namely the presence of random encodings to protect the implementation, and DCA could work without such a requirement in other contexts.

Table 4.4: Trace complexity of DCA in Section 4.3, collision attack (CA) in Section 4.4 and MIA in Section 4.5 against internal encodings in our theoretical analysis and in breaking NSC variant, where m is the encoding bit-size ($m = 8$ in the NSC variant).

	DCA	**CA**	**MIA**
Theoretical	$\mathcal{O}(2^{2m})$	$\mathcal{O}\left(2^{\frac{m}{2}}\right)$	$\mathcal{O}\left(2^{\frac{m}{2}}\right)$
NSC variant	1800	60	70

The trace complexities of the three approaches are summarized in Table 4.4. Notably, compared to DCA, our collision attack and MIA have low trace complexities. Namely, they only require about $2^{\frac{m}{2}}$ traces and are thus very effective at defeating internal encodings. For instance, while using 16-bit encodings would imply a huge code size,[5] our collision attack or improved MIA would still break the implementation with a few hundred traces.

In terms of time complexity, DCA and MIA take $\Theta(t \cdot |\mathcal{K}| \cdot N)$ operations while the collision attack takes $\Theta\left(t \cdot |\mathcal{K}| \cdot N^2\right)$ operations. Assuming that we have $|\mathcal{K}| = 2^n$ and that we take $n = 2m$ as suggested in our analyses these time complexities

[5] With 16-bit encodings, a single encoded table taking two arguments (e.g., an encoded XOR) requires $2^{2\times16} \times 2$ bytes which is more than 8 GB.

become $\mathcal{O}(2^{4m} \cdot t)$, $\mathcal{O}(2^{3m} \cdot t)$, and $\mathcal{O}(2^{2.5m} \cdot t)$ for DCA, CA and MIA respectively. Despite its slightly better asymptotic time complexity, MIA was slower than our collision attack in our practical attack experiments. This would probably not be the case in a setting where more computation traces are required (typically with a higher m).

Finally, we note that our collision attack and MIA are especially suited to attack internal encodings since both attacks rely on the collision behavior of some target variables which is not affected by the application of a random bijection. This explains their superiority over DCA in this context. However, in the presence of different countermeasures (especially affecting the collision behavior) these attacks could fail where a DCA could still succeed.

Chapter 5

Linear Decoding Analysis and Higher-Degree Extension

Part of the results presented in this chapter have been published in (Goubin, Paillier, et al., 2020).

5.1 Introduction

Linear encodings a.k.a. linear masking, are widely used in the protection against side-channel attacks (Chari, Jutla, et al., 1999; Goubin and Patarin, 1999; Rivain and Prouff, 2010). To linearly mask a secret variable, one splits it into several linear shares that are then processed independently. As a consequence, to conquer a cryptographic implementation protected by linear masking, a side-channel adversary has to exploit joint leakage of all the independently manipulated shares, which may imply an unaffordable computation overhead in practice whenever the number of shares is large enough. Since the DCA attack relies on the same ideas as DPA, the prospect of applying masking to secure a software implementation against DCA is promising. As such, linear encoding is one of the many considered countermeasures in industrial white-box solutions, as well as in open white-box challenges, e.g., the winning challenge WhibOx 2017 context – `Adoring Poitras`.

In the chapter, we

- **Formalize *Linear Decoding Analysis* (LDA).** In Section 5.2, we first describe a passive gray-box attack – *linear decoding analysis* (LDA), which is capable of extracting the key information from a set of encoding intermediate variables, as long as the underlying plain value can be recovered through a linear decoding of those encoding variables. We show in Section 5.3 that the number of required computation traces to perform a successful LDA attack is only slightly larger than the size t of trace window under attack, assuming that all the target shares are in this window. The obtained computation complexity of LDA is $\mathcal{O}\left(|\mathcal{K}| \cdot t^{2.8}\right)$.

- **Extend LDA to Higher-Degree**. We then explain in Section 5.4 how LDA can be extended to break implementations protected with higher-degree encodings. The generalized *higher-degree decoding analysis* (HDDA) is achieved by firstly converting the computation traces into higher-degree ones, then launching the LDA attack on the higher-degree traces. Therefore, the computation complexity of HDDA is $\mathcal{O}\left(|\mathcal{K}| \cdot t^{2.8d}\right)$ for a decoding function of degree d.

5.2 Linear Decoding Analysis

A *linear decoding analysis* (LDA) attacker against a white-box implementation can extract the key information contained in a set of encoded intermediate variables, provided that the underlying plain variable can be recovered through a linear decoding. In this section, we formalize the LDA attack.

Without loss of generality, we assume that the white-box implementation processes intermediate variables (that can be represented) in some finite field. Typically the finite field is $\mathbb{F}_2$ for a Boolean circuit, but it could be $\mathbb{F}_{2^{32}}$ for a 32-bit architecture program, or more generally it is $\mathbb{F}_q$ for any prime (power) q. For clarity, we abuse

notation by denoting the finite field by $\mathbb{F}$. Let us denote $s = \varphi_k(x) \in \mathbb{F}$ the target sensitive variable where φ is a deterministic function, $k \in \mathcal{K}$ is a subkey for some subkey space $\mathcal{K}$, and x is a part of the input plaintext (or output ciphertext).

As a passive adversary (see Section 3.2), an LDA adversary controls a white-box implementation and she can execute it for several plaintexts and dynamically record the corresponding *computation traces*. These traces consist of ordered t-tuples

$$\boldsymbol{v} = (v_1,\ v_2,\ \cdots,\ v_t)$$

of the values taken by the intermediate variables (e.g., values read/stored in memory, results of CPU instructions, etc.), where $v_i \in \mathbb{F}$ for every i. These computation traces might be related to a small part of the full execution, e.g., when targeting a specific operation either localized by data dependency analysis or guessed using an automated search. The adversary collects N such computation traces $\boldsymbol{v}^{(i)} = (v_1^{(i)},\ v_2^{(i)},\ \cdots,\ v_t^{(i)})$ that correspond to N (chosen) plaintexts $x^{(i)}$ for $1 \leq i \leq N$. Then, for every key guess $k \in \mathcal{K}$, she constructs the following system of linear equations:

$$\begin{bmatrix} 1 & v_1^{(1)} & v_2^{(1)} & \cdots & v_t^{(1)} \\ 1 & v_1^{(2)} & v_2^{(2)} & \cdots & v_t^{(2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & v_1^{(N)} & v_2^{(N)} & \cdots & v_t^{(N)} \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_t \end{bmatrix} = \begin{bmatrix} \varphi_k(x^{(1)}) \\ \varphi_k(x^{(2)}) \\ \vdots \\ \varphi_k(x^{(N)}) \end{bmatrix}, \tag{5.1}$$

where $(a_0, a_1, a_2, \cdots, a_t)$ are the unknown coefficients in $\mathbb{F}$. If the system is unsolvable for every key guess k, then the attack fails. If the system is solvable for a single key guess k, there is a strong presumption that it is the right key guess i.e., $k = k^*$, the adversary then returns k as the (candidate) correct key.

For N sufficiently greater than t, if the above system is solvable, it means that the target intermediate variables satisfy

$$a_0 + \sum_{i=1}^{t} a_i \cdot v_i = \varphi_k(x) = s\ . \tag{5.2}$$

Namely, the white-box implementation encodes the sensitive variable s in the v_i's through the above (decoding) relation. In particular the variables $\{v_i\ ;\ a_i \neq 0\}$ form a linear sharing of s. We stress that such encoding encompasses any kind of Boolean masking or linear secret sharing of *any order*, see for instance (Ishai et al., 2003; Rivain and Prouff, 2010; Beimel, 2011). Moreover, the encoding function is not necessarily linear: one would generate the masks (or the shares) pseudorandomly from the full input plaintext p, implying that the encoding function

$$\mathsf{enc} : (p, k) \ \mapsto\ (v_1, v_2, \cdots, v_t)$$

could be of high degree in p, whereas the decoding function

$$\mathsf{dec} : (v_1, v_2, \cdots, v_t) \mapsto s = \varphi_k(x)$$

is linear.

Complexity. LDA has complexity $\mathcal{O}\big(|\mathcal{K}| \cdot t^{2.8}\big)$. For each key guess $k \in \mathcal{K}$, the attack can be split into two phases: first solve a linear system of $t+1$ equations in $t+1$ variables (we assume that the corresponding square matrix is full rank without loss of generality), and then check whether the $N-(t+1)$ equations match the recovered solution. The complexity of the first phase is $\mathcal{O}\big(t^{2.8}\big)$ by using the Strassen algorithm (Strassen, 1969).[1] The second phase is then of complexity $\mathcal{O}(t \cdot (N-t))$ which is negligible compared to the first phase since, as shown in Section 5.3, a high success probability can be obtained by taking a (small) constant number of additional traces $N-t$. We thus obtain a total complexity of $\mathcal{O}\big(|\mathcal{K}| \cdot t^{2.8}\big)$ for the recovery of one subkey $k^* \in \mathcal{K}$.

Window Search. When the adversary is not able to accurately localize the target encoding among the intermediate variables then she might apply LDA to the full computation trace (i.e., the computation trace of the full execution). If we denote by τ the size of this full trace, then the obtained complexity is of $\mathcal{O}\big(|\mathcal{K}| \cdot \tau^{2.8}\big)$, which might be too large. For instance, this would have made about 2^{59} operations for a trace of size $\tau \approx 280,000$ as obtained for the `Adoring Poitras` minimized circuit before data dependency analysis (see Section 8.4.3).

In practice, one can significantly improve this complexity by searching the potential encoding variables in a relatively small window of the computation trace. In a practical white-box implementation, the computation for some specific (encoded) intermediate result, has some *locality* property which implies that the related intermediate variables are located in a t-size subtrace of the full τ-size computation trace. Formally, in a full computation trace $(v_1, v_2, \cdots, v_\tau)$, t consecutive points

$$(v_{i+1}, v_{i+2}, \cdots, v_{i+t}),$$

for some index i, contain all variables to decode the target sensitive variable s. Without knowing the locality parameter t and the right position i in the full trace, the adversary can try LDA for several t and i. Specifically, we suggest applying LDA on the subtrace obtained for every $i \in \{1, 2, \ldots, \tau - t\}$ for an increasing $t = 2^1, 2^2, 2^3, \ldots$ The total complexity is then of $\mathcal{O}\big(|\mathcal{K}| \cdot \tau \cdot t^{2.8}\big)$, where t is the right locality parameter, which is better than the full-trace attack complexity whenever $t < \tau^{0.64}$.

[1] This could theoretically be reduced to $\mathcal{O}\big(t^{2.376}\big)$ using the Coppersmith–Winograd algorithm for very large t, see for instance (Golub and Van Loan, 1996), but in practice, one shall prefer the Strassen algorithm.

5.3 Analysis of LDA

The soundness of LDA results from the fact that if a decoding relation such as Equation 5.2 does exist for the target intermediate variable s, and if the shares are well selected in the computation trace $v = (v_1, v_2, \cdots, v_t)$, then LDA will solve the system for the right key guess k^*. For a wrong key guess, on the other hand, no solution should be found unless (1) φ is a linear function w.r.t. the field $\mathbb{F}$, or (2) an encoding $\varphi_k(x)$ is computed by the implementation for a wrong key guess $k^\times \neq k^*$ (to fool the attacker). These two limitations can simply be mitigated: (1) can be avoided by targeting an appropriate intermediate result (such as an s-box output), and it is unlikely that (2) occurs for all the possible subkeys $k \in \mathcal{K}$ which would arguably represent a huge computational overhead for the implementation (and would become intractable as we go deeper in the computation).

We analyze hereafter the success probability of LDA under the following assumptions:

- a linear decoding relation [such as Equation 5.2] does exist between v and s,
- the plaintext (part) x is uniformly distributed,
- v is uniformly distributed among the t-tuples satisfying the decoding relation $a_0 + \sum_i a_i \cdot v_i = \varphi_{k^*}(x)$,

The two first assumptions are necessary conditions of the LDA attack context which are arguably satisfied in some real white-box design/attack use cases (as typically considered in this chapter). The last assumption is *ideal* and is not necessary for LDA to work but only for the purpose of our formal analysis. It could somehow be relaxed by considering potential statistical dependencies between the variables which would complicate the analysis without strongly impacting the result.

Proposition 5.1. *Under the above assumptions, the probability that the LDA linear system Equation 5.1 is solvable for an incorrect key guess $k^\times \neq k^*$ is lower than $|q|^{N-t-1}$, where*

$$q \stackrel{\text{def}}{=} \max \left\{ \Pr\left(\varphi_{k^*}(X) = \alpha \cdot \varphi_{k^\times}(X)\right) \; ; \; \alpha \in \mathbb{F}^*, \; (k^*, k^\times) \in \mathcal{K}^2 \right\} \tag{5.3}$$

for a uniform distribution of X.

Proof. Without loss of generality, we assume that there exists a subsystem $\mathcal{S}$ containing $t+1$ equations from Equation 5.1 such that the corresponding matrix is full-rank (implying that $\mathcal{S}$ has one and only one solution whatever the target vector).[2] The solution of $\mathcal{S}$ is denoted $\boldsymbol{a}^* = (a_0^*, a_1^*, \cdots, a_t^*)$ for the correct key guess k^* and $\boldsymbol{a}^\times = (a_0^\times, a_1^\times, \cdots, a_t^\times)$ for the wrong key guess $k^\times$. In the following, we will consider that the $t+1$ equations in $\mathcal{S}$ are the $t+1$ first equations of the system. Then, two possible cases occur:

[2] According to our three assumptions, the probability that there does not exist any full rank subsystem containing $t+1$ equations is negligible.

1. There exists a constant $\alpha \in \mathbb{F}$ such that $\boldsymbol{a}^{\times} = \alpha \cdot \boldsymbol{a}^{*}$. This implies that

$$\varphi_{k^{\times}}(x^{(i)}) = \alpha \cdot \varphi_{k^{*}}(x^{(i)}) \tag{5.4}$$

for every $1 \leq i \leq t+1$. Moreover, the full system has a solution for the guess $k^{\times}$ if and only if Equation 5.4 is further satisfied for every $i \in \{t+2, \ldots, N\}$. Since the $x^{(i)}$ are uniformly distributed, this happens with probability at most $q^{N-(t+1)}$.

2. There does not exist a constant $\alpha \in \mathbb{F}$ such that $\boldsymbol{a}^{\times} = \alpha \cdot \boldsymbol{a}^{*}$. In that case, from our ideal assumption, we have

$$a_0^{\times} + \sum_{j=1}^{N} a_j^{\times} \cdot v_j^{(i)} \sim \mathcal{U}(\mathbb{F}) \,,$$

(where $\mathcal{U}(\mathbb{F})$ denotes the uniform distribution over $\mathbb{F}$) for every $i \in \{t+2, \ldots, N\}$. Then the full system has a solution for the guess $k^{\times}$ if and only if

$$a_0^{\times} + \sum_{j=1}^{N} a_j^{\times} \cdot v_j^{(i)} = \varphi_{k^{\times}}(x^{(i)})$$

is satisfied for every $i \in \{t+2, \ldots, N\}$, which occurs with probability

$$\left(\frac{1}{|\mathbb{F}|}\right)^{N-(t+1)} < q^{N-(t+1)} \,.$$

□

By Proposition 5.1, the probability that the system Equation 5.1 is solvable for the incorrect key guess $k^{\times}$ is exponentially small in N. In practice, an appropriately chosen φ makes q close to $\frac{1}{|\mathbb{F}|}$ and the probability quickly becomes negligible as N grows over $t+1$. Moreover, the number of extra traces required to get a given (negligible) probability of false positive depends on the target function φ, but is constant with respect to t.

As an illustration, if the φ_k is the output a first round s-box of AES, then

- for the Boolean case ($\mathbb{F} = \mathbb{F}_2$), i.e., we target φ_j for some $1 \leq j \leq 8$: according to Equation 5.3, we obtain $q = \frac{9}{16}$ and taking, e.g., 40 extra equations makes the false-positive probability lower than 2^{-32};
- for the full field case ($\mathbb{F} = \mathbb{F}_{2^8}$), i.e., we target φ: according to Equation 5.3, we obtain $q = \frac{7}{256}$ and taking, e.g., 7 extra equations makes the false-positive probability lower than 2^{-32}.

5.4 Extension to Higher Degrees

The linear decoding assumption necessary to LDA might not be satisfied in practice for some white-box implementations. Depending on the algebraic structure of the encoding scheme used to protect intermediate variables, the decoding function might have an algebraic degree greater than 1. We explain in this section how LDA can be generalized to break implementations with higher degree decoding functions. This generalization shall be called *higher-degree decoding analysis* (HDDA) in the following.

For each collected computation trace $\boldsymbol{v}$, the HDDA adversary computes a corresponding *higher-degree trace* defined as:

$$\boldsymbol{w} = (1) \parallel \boldsymbol{v} \parallel \boldsymbol{v}^2 \parallel \cdots \parallel \boldsymbol{v}^d \,,$$

where $\parallel$ is the concatenation operator and where $\boldsymbol{v}^j$ is the vector of degree-j monomials:

$$\boldsymbol{v}^j = (v_{i_1} \cdot v_{i_2} \cdot \ldots \cdot v_{i_j})_{1 \le i_1 \le i_2 \le \cdots \le i_j \le t} \,.$$

The size of the vector $\boldsymbol{v}^j$ is the number of degree-j monomials in t variables, which equals $\binom{j+t-1}{j}$. The size of the higher-degree trace is the number of monomials of degree lower than or equal to d, which is

$$t' = \sum_{j=0}^{d} \binom{j+t-1}{j} = \binom{t+d}{d} \le \frac{(t+d)^d}{d!} \ll t^d \,.$$

From the computation traces obtained for N executions (with random input plaintext), the adversary computes N such higher-degree traces

$$\boldsymbol{w}^{(i)} = (w_1^{(i)},\, w_2^{(i)},\, \cdots,\, w_{t'}^{(i)}) \,.$$

Then, for every key guess $k \in \mathcal{K}$, she constructs the linear system:

$$\begin{bmatrix} 1 & w_1^{(1)} & w_2^{(1)} & \cdots & w_{t'}^{(1)} \\ 1 & w_1^{(2)} & w_2^{(2)} & \cdots & w_{t'}^{(2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & w_1^{(N)} & w_2^{(N)} & \cdots & w_{t'}^{(N)} \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{t'} \end{bmatrix} = \begin{bmatrix} \varphi_k(x^{(1)}) \\ \varphi_k(x^{(2)}) \\ \vdots \\ \varphi_k(x^{(N)}) \end{bmatrix},$$

where $(a_0,\, a_1,\, a_2,\, \cdots,\, a_{t'})$ are the unknown coefficients in $\mathbb{F}$.

If the above system is solvable for N sufficiently greater than t' then (with overwhelming probability) there exists a degree-d decoding function dec (with the a_i's as coefficients) such that

$$\mathsf{dec}(v_1,\, v_2,\, \ldots, v_t) = \varphi_k(x) \,.$$

In particular, if the white-box encoding of the sensitive variable $s = \varphi_{k^*}(x)$ can

be decoded with a degree-d function and if the shares of the encoding are well included in the computation trace, then the above system will be solvable for k^* and the solution will give the right decoding function.

On the other hand, and as for the LDA case (i.e., the case $d = 1$) analyzed above, the probability that the system is solvable for a wrong key guess $k^\times \neq k^*$ quickly becomes negligible as N increases (over t'), provided that there exists no degree-d relation between $\varphi_{k^\times}$ and φ_{k^*} (in particular φ is of degree greater than d).

Complexity

Following the complexity analysis of LDA in Section 5.2, HDDA has complexity $\mathcal{O}(|\mathcal{K}| \cdot {t'}^{2.8})$. For a small constant d, this makes a complexity of $\mathcal{O}(|\mathcal{K}| \cdot t^{2.8d})$. The complexity of HDDA with window search in a computation trace of size τ with an (unknown) locality parameter of t is then of $\mathcal{O}(|\mathcal{K}| \cdot \tau \cdot t^{2.8d})$.

Chapter 6

Higher-Order DCA against Masking and Shuffling

The results presented in this chapter have been published in (Bogdanov, Rivain, et al., 2019).

6.1 Introduction

The *differential computation analysis* (DCA) adversary is highly reminiscent of the standard side-channel adversary. A natural approach when attempting to mitigate the threat of DCA attacks is to apply known countermeasures, i.e., *linear masking* and *operation shuffling* from the side-channel literature. However, it is not clear how well these countermeasures carry over to the white-box context and what level of security can be achieved by such countermeasures against a DCA adversary. To address these issues, we achieve the following in this chapter:

1. **Higher-order DCA:** We develop *higher-order DCA* in Section 6.2 to analyze the security of the proposed protection. This attack combines several coordinates of a computation trace into a higher-order trace to defeat the masking countermeasure. We show that higher-order DCA is able to break a masked implementation of any order using a few traces. However, by introducing noise in the form of shuffling, the security of the implementation can be dramatically increased. As a demonstration, a typical AES implementation with 2^{nd} order masking (and shuffling degree of 16) requires 2^{21} traces to break with 3^{rd} order DCA.

2. **Multivariate higher-order DCA:** We extend the above attack by introducing a multivariate version in Section 6.3, which reduces the computational complexity by decreasing the number of required traces for a successful attack. Using this multivariate variant, the number of traces required to successfully attack the AES implementation mentioned above can be reduced to 2^{10}.

3. **Formal analysis and experimental verification:** We derive analytic expressions for the success probability and attack complexities of both the higher-order DCA and its multivariate variant. Using these expressions, we are able to give estimates for the security level of a masked and shuffled implementation in the DCA setting. As an example, an AES implementation with 7^{th}-order masking would have a security level of about 85 bits in this setting. The accuracy of our expressions for the success probability of the multivariate higher-order DCA is verified in Equation 6.3.2 through extensive experiments for a wide range of implementation and attack parameters. 2 000 attacks of up to order 4 were simulated, using as many as 30 000 traces per attack.

6.2 Higher-Order DCA

6.2.1 Higher-Order DCA Distinguisher

While masking has been proven to be an effective defense against standard DPA, and we have argued for its effectiveness against standard first-order DCA, there are ways to attack such masked implementations. For hardware implementations, it is well

known that an $(n-1)^{\text{th}}$-order masked implementation can be defeated by n^{th}-order DPA, if no other protection is employed. We will therefore develop a higher-order version of DCA.

An n^{th}-order DCA consists of a preprocessing step followed by a first-order DCA. The adversary first preprocesses each computation trace $\boldsymbol{v}$ to obtain an n^{th}-order computation trace $\boldsymbol{w}$ by applying a so-called (n^{th}-order) *combination function* ψ. Specifically, the n^{th}-order computation trace $\boldsymbol{w}$ consists of $q = \binom{t}{n}$ points $(w_1, \ldots, w_q)$ given by

$$w_j = \psi(v_{j_1}, v_{j_2}, \ldots, v_{j_n}), \quad \{j_1, \ldots, j_n\} = \phi(j),$$

where $\phi(j)$ is the j^{th} subset of $\{1, \ldots, t\}$ of size n (for some ordering). After computing the set of n^{th}-order traces $\boldsymbol{w}^{(1)}, \ldots, \boldsymbol{w}^{(N)}$, the adversary proceeds as for first-order DCA, using the $\boldsymbol{w}_i$'s as input to the distinguisher D. Specifically, the adversary computes the score vector

$$(\gamma_k)_{k \in \mathcal{K}} = \mathsf{D}\big((\boldsymbol{w}^{(i)})_i \,;\, (x^{(i)})_i\big)$$

in order to determine a candidate for k^*.

For side-channel analysis of hardware implementations, it has been shown that a good combination function for higher-order DPA is the centered product

$$\psi : (v_1, \ldots, v_n) \;\mapsto\; \prod_j (v_j - \mu_j)\,,$$

where μ_j is the average of the leakage point v_j over several encryptions. Nevertheless, since the measurements in this setting are inherently noisy, a larger masking degree will require a larger number of traces to obtain a good success probability. Note that this is not the case in the DCA context, if no noise is introduced in the implementation, e.g., by using shuffling as described in Section 3.3.3. In this case, the exact value of each variable that appears in an execution of the implementation appears at the same position of every computation trace $\boldsymbol{v}^{(i)}$. Then there exists a fixed j^*, such that for $\phi(j^*) = (j_1^*, \ldots, j_n^*)$, the elements $v_{j_1^*}, \ldots, v_{j_n^*}$ of the trace are the shares of the target secret variable s. In that case, an optimal choice for the combination function is the XOR sum of the trace values, that is

$$\psi(v_{j_1}, v_{j_2}, \ldots, v_{j_n}) = v_{j_1} \oplus v_{j_2} \oplus \cdots \oplus v_{j_n}\,.$$

For this combination function, we have that $\psi(v_{j_1^*}, v_{j_2^*}, \ldots, v_{j_n^*}) = s$ for all the n^{th}-order traces. By counting the number of times this equality holds, we can easily determine the correct key. That is, we set

$$\gamma_k = \max_j \big(C_k(\boldsymbol{v}_{\phi(j)}, (x_i)_i)\big), \text{ with } C_k(\boldsymbol{v}_{\phi(j)}, (x^{(i)})_i) = \left|\left\{ i \,;\, \bigoplus_{l \in \phi(j)} v_l^{(i)} = \varphi_k(x_i) \right\}\right|.$$

For the correct key k^*, we deterministically have that $\gamma_{k^*} = N$. Thus, if no noise is

present, the higher-order DCA is successful when $\gamma_{k^\times} < N$ for all $k^\times \neq k^*$. The probability of this happening is quite close to 1, even for small N. Thus, the introduction of some noise in the traces is required to secure a masked white-box implementation against DCA.

Note that a *linear decoding analysis* (LDA) attack, described in Chapter 5 can break a noise-free masked implementation with complexity approximately cubic in the size of the computation trace. LDA also demonstrates that a linear masking is a weak countermeasure. However, LDA would completely fail in presence of a bit of noise (e.g. introduced by shuffling), which is not the case of the higher-order DCA described above as it will be analysed hereafter.

6.2.2 Higher-Order DCA against Masking and Shuffling

We now consider how well the masked and shuffled implementation resists the higher-order DCA attack described above. Due to the shuffling, the adversary is no longer guaranteed that her prediction for the correct key guess will correspond to a single time point for all traces. Thus, she must compensate by increasing the number of traces. The higher the degree of shuffling, the more traces need to be collected.

Attack Analysis

In the following, we assume that the adversary knows exactly where in the computation trace to attack. That is, for a masking order $n-1$ and a shuffling degree λ, she knows the range of the $t = \lambda \cdot n$ time points that contain the shares of the target secret variable. In other words, the length of each computation trace $\boldsymbol{v}$ is t. This, intuitively, represents the optimal situation for the adversary.[1] We seek an expression for the success probability of the attack, i.e., the probability that the correct key has a higher score than all other key candidates.

The adversary proceeds as above and computes the n^{th}-order computation trace. However, there will no longer be a single value j such that $C_{k^*}(\boldsymbol{v}_{\phi(j)}, (x_i)_i) = N$ deterministically for the correct key k^*. Thus, we need to know the distribution of γ_k, both for a wrong and a right guess of the key.

Theorem 6.1. *Consider a masked white-box implementation of order $n-1$ with shuffling degree λ. Let $p = \binom{t}{n}^{-1}$ where $t = \lambda \cdot n$, and let $F(x; m, q)$ be the CDF (cumulative distribution function) of the binomial distribution with parameters m and q. Let $|\mathcal{K}|$ be the number of possible key values and define*

$$F^{\times}_{\max}(x) = F\Big(x; N, (1-p)\tfrac{1}{|\mathcal{K}|}\Big)^{\binom{t}{n}},$$
$$F^{*}_{\max}(x) = F\Big(x; N, p + (1-p)\tfrac{1}{|\mathcal{K}|}\Big)^{\binom{t}{n}}.$$

[1] In practice, the adversary could exhaustively search the correct location of the $(\lambda \cdot n)$-length subtrace in the full computation trace of length t_{full}, which increases the complexity at most t_{full} times.

Then the probability of recovering a key using n^{th}-order DCA with N traces is

$$p_{succ} = \left(\sum_{i=0}^{N} \left(F^*_{\max}(i) - F^*_{\max}(i-1) \right) \cdot F^{\times}_{\max}(i-1) \right)^{|\mathcal{K}|-1}.$$

Proof. Consider a specific value w_j of the higher-order trace $\boldsymbol{w}$. Denote by E the event that w_j corresponds to the combination of the correct shares. The probability of E occurring, i.e., of choosing the correct n shares out of the t elements of the original computation trace $\boldsymbol{v}$, is $p = \binom{t}{n}^{-1}$.

Fix some plaintext and the corresponding trace. By the law of total probability, the probability that a value w_j of the n^{th}-order trace is equal to a prediction $s = \varphi_k(x)$ for some key guess k is

$$\Pr(w_j = s) = \Pr(w_j = s|E) \cdot \Pr(E) + \Pr(w_j = s|\neg E) \cdot \Pr(\neg E).$$

For a wrong key guess $k^{\times} \neq k^*$, $\Pr(w_j = s^{\times}|E) = 0$ where $s^{\times} = \varphi_{k^{\times}}(x)$, while for a right key guess $\Pr(w_j = s^*|E) = 1$ where $s^* = \varphi_{k^*}(x)$. In both cases, we have $\Pr(w_j = s|\neg E) = 1/|\mathcal{K}|$. In total,

$$p^{\times} = \Pr(w_j = s^{\times}) = \frac{1-p}{|\mathcal{K}|},$$
$$p^{*} = \Pr(w_j = s^{*}) = p + \frac{1-p}{|\mathcal{K}|}.$$

Thus, for N traces,

$$C_{k^{\times}}\left(\boldsymbol{v}_{\phi(j)}, (x^{(i)})_i \right) \sim \text{Bin}(N, p^{\times})$$

for a wrong key guess, and

$$C_{k^{*}}\left(\boldsymbol{v}_{\phi(j)}, (x^{(i)})_i \right) \sim \text{Bin}(N, p^{*})$$

for the right key guess. Note that $|\boldsymbol{w}| = \binom{t}{n}$. Let $X_1, \ldots, X_{|\boldsymbol{w}|}$ be distributed as $\text{Bin}(N, p^{\times})$. Then $\gamma_{k^{\times}} \sim \max X_i$, and we denote the CDF by $F^{\times}_{\max}(x)$. If the X_i were independent, we would have

$$F^{\times}_{\max}(x) = F(x; N, p^{\times})^{\binom{t}{n}}.$$

While the $(X_i)_i$ are pairwise independent, they are not mutually independent. However, we find that in practice, the dependence is so weak that $\gamma_{k^{\times}}$ approximately has CDF $F^{\times}_{\max}$, even for small values of $|\boldsymbol{w}|$ and N. We define $F^{*}_{\max}(x)$ similarly.

The attack is successful if $\gamma_{k^*} > \gamma_{k^{\times}}$ for all $k^{\times}$. As there are $|\mathcal{K}| - 1$ wrong keys, and all $\gamma_{k^{\times}}$ are independent and identically distributed, we have

$$p_{\text{succ}} = \Pr(\gamma_{k^*} > \gamma_{k^{\times}})^{|\mathcal{K}|-1},$$

where

$$\Pr(\gamma_{k^*} > \gamma_{k^\times}) = \sum_{i=0}^{N} \left(F^*_{\max}(i) - F^*_{\max}(i-1) \right) \cdot F^\times_{\max}(i-1) \,,$$

which concludes the proof. □

Attack Complexity

We can use Theorem 6.1 to calculate the required N to obtain a desired probability of success. The number of traces required to obtain 90% success probability for a range of parameters is shown in Table 6.1. Here, $|\mathcal{K}| = 256$, and the parameters would be typical choices for e.g., a protected AES implementation.

Table 6.1: The number of traces N and the time needed to successfully attack an implementation with $(n-1)$-order masking and shuffling of degree λ with n^{th}-order DCA. Here, $|\mathcal{K}| = 256$, and we fix the success probability at 90%. The parameters chosen would be typical for a protected AES implementation.

n	λ	$\log_2 N$	$\log_2$ time
2	8	8.6	23.5
2	16	11.0	28.0
3	8	15.7	34.7
3	16	21.6	43.7
4	8	23.6	46.7
4	16	31.7	59.0

We consider the time complexity of recovering the secret key k^* using the higher-order DCA attack. For a fixed probability of success p_{succ}, let N_n be the number of computation traces required to obtain this probability for an n^{th}-order implementation. We again assume that $t = \lambda \cdot n$. The cost of computing the higher-order trace is $N_n \cdot \binom{t}{n}$. Then, for each key guess k and each time point in the higher-order trace, the adversary computes C_k. The complexity of this is $|\mathcal{K}| \cdot N_n \cdot \binom{t}{n}$. Thus, the time complexity is $\mathcal{O}\left(|\mathcal{K}| \cdot N_n \cdot \binom{t}{n}\right)$. Table 6.1 shows the time complexity of the attack for a range of parameters.

6.3 Multivariate Higher-Order DCA

In the higher-order DCA, presented in Section 6.2, the adversary tries to correlate each sample of the higher-order trace with the predicted variable independently, finally taking the maximum over the obtained correlation scores. Such an approach is not optimal, as successive samples may carry joint information on the secret. As in the side-channel context, one can take advantage of this joint information by performing a *multivariate attack*, namely an attack in which the distinguisher exploits the multivariate distribution of different samples in the higher-order trace. Emblematic multivariate attacks in the classical side-channel context are the so-called *template*

attacks (Chari, Rao, et al., 2003). In the following section, we describe a similar attack in the setting of the DCA adversary.

6.3.1 Multivariate Higher-Order DCA against Masking and Shuffling

Our proposed multivariate higher-order DCA attack is based on the principle of maximum likelihood. Similar techniques have been adopted in side-channel template attacks. Let K, $(X^{(i)})_i$, and $(\boldsymbol{V}^{(i)})_i$ be random variables representing the subkey k, the public inputs $(x^{(i)})_i$, and the computation traces $(\boldsymbol{v}^{(i)})_i$. The likelihood distinguisher is then defined as

$$\begin{aligned} &\mathsf{L} : \left((\boldsymbol{v}^{(i)})_i, (x^{(i)})_i\right) \mapsto (\ell_k)_{k\in\mathcal{K}}, \\ &\ell_k \propto \Pr\left(K = k \mid (\boldsymbol{V}^{(i)})_i = (\boldsymbol{v}^{(i)})_i \wedge (X^{(i)})_i = (x^{(i)})_i\right), \end{aligned} \tag{6.1}$$

where $\propto$ means equal up to some factor constant w.r.t. k. To evaluate this likelihood function, we need a model for the distribution of the traces (also called a *template* in the side-channel context). It is well known that if Equation 6.1 is evaluated from the true distributions of $(X^{(i)})_i$ and $(\boldsymbol{V}^{(i)})_i$, then the above distinguisher is optimal. This is sound, as in this case, the score is the exact probability that the target subkey equals a key guess k, for all $k \in \mathcal{K}$.

In the following, we will assume that $\boldsymbol{V}^{(i)}$ is composed of t uniformly distributed random variables $V_1^{(i)}, V_2^{(i)}, \ldots, V_t^{(i)}$, with the constraint that for a uniformly chosen j, we have $\bigoplus_{l\in\phi(j)} V_l^{(i)} = \varphi_K(X^{(i)})$. This assumption matches the setting of a masked and shuffled implementation. The public inputs $X^{(i)}$ and the subkey K are also assumed to be uniformly distributed and mutually independent. Under this model, we have the following result:

Proposition 6.1. *The likelihood distinguisher, Equation 6.1, satisfies:*

$$\ell_k \propto \prod_{i=1}^{N} C_k(\boldsymbol{v}^{(i)}, x^{(i)}),$$

where $C_k(\boldsymbol{v}, x)$ is the number of n-tuples in a trace $\boldsymbol{v}$ with bitwise sum equals to $\varphi_k(x)$, that is

$$C_k(\boldsymbol{v}, x) = \left|\left\{(v_{j_1}, \ldots, v_{j_d})\ ;\ v_{j_1} \oplus \cdots \oplus v_{j_n} = \varphi_k(x)\right\}\right|.$$

Proof. By applying the Bayes' rule, one gets (we skip random variables for the sake of clarity):

$$\Pr\left(k \mid (\boldsymbol{v}^{(i)})_i \wedge (x^{(i)})_i\right) = \frac{\Pr\left((\boldsymbol{v}^{(i)})_i \mid k \wedge (x^{(i)})_i\right) \cdot \Pr\left(k \wedge (x^{(i)})_i\right)}{\Pr\left((\boldsymbol{v}^{(i)})_i \wedge (x^{(i)})_i\right)}$$

By mutual independence of the $X^{(i)}$s and K, we have

$$\Pr\left(k \wedge (x^{(i)})_i\right) = \frac{1}{|\mathcal{K}|}\left(\frac{1}{|\mathcal{X}|}\right)^N$$

for every $k \in \mathcal{K}$. Moreover, $\Pr\left((\boldsymbol{v}^{(i)})_i \wedge (x^{(i)})_i\right)$ is constant with respect to k. We hence get

$$\Pr\left(k \mid (\boldsymbol{v}^{(i)})_i \wedge (x^{(i)})_i\right) \propto \Pr\left((\boldsymbol{v}^{(i)})_i \mid k \wedge (x^{(i)})_i\right) .$$

By mutual independence of the $\boldsymbol{V}^{(i)}$s and the $X^{(i)}$s we further deduce

$$\Pr\left((\boldsymbol{v}^{(i)})_i \mid k \wedge (x^{(i)})_i\right) = \prod_{i=1}^{N} \Pr(\boldsymbol{v}^{(i)} \mid k \wedge x^{(i)}) .$$

For the sake of simplicity, we skip the index i in the following. By the law of total probability, we have

$$\Pr(\boldsymbol{v} \mid k \wedge x) = \sum_{\phi(j)} \Pr(\mathcal{S}_{\phi(j)}) \cdot \Pr(\boldsymbol{v} \mid k \wedge x \wedge \mathcal{S}_{\phi(j)}) ,$$

where $\mathcal{S}_{\phi(j)}$ denotes the event that the set $\phi(j)$ is selected for the sharing of $\varphi_K(X)$. By definition, we have

$$\Pr(\mathcal{S}_{\phi(j)}) = \frac{1}{\binom{t}{n}}$$

and

$$\Pr(\boldsymbol{v} \mid k \wedge x \wedge \mathcal{S}_{\phi(j)}) = \begin{cases} \left(\frac{1}{|\mathcal{V}|}\right)^{t-1} & \textbf{if } \bigoplus_{l \in \phi(j)} v_l = \varphi_k(x), \\ 0 & \textbf{otherwise}. \end{cases} \tag{6.2}$$

which finally gives

$$\Pr(\boldsymbol{v} \mid k \wedge x) \propto C_k(\boldsymbol{v}, x) . \tag{6.3}$$

□

For practical reasons, it is more convenient to evaluate the log-likelihood, which is $\log \ell_k = \sum_{i=1}^{N} \log C_k(\boldsymbol{v}^{(i)}, x^{(i)})$. Note that this does not affect the ranking of the key guesses (as the logarithm is a monotonically increasing function) and therefore has no impact on the success probability of the attack.

6.3.2 Analysis of the Likelihood Distinguisher

In this section, we analyze the success probability of the likelihood distinguisher. For the sake of simplicity, we only consider two key guesses, namely the right key guess k^* and a wrong key guess $k^\times$. We then consider their likelihood scores ℓ_{k^*} and $\ell_{k^\times}$ random variables, since

$$\ell_k = \prod_{i=1}^{N} C_k(\boldsymbol{V}^{(i)}, X^{(i)}) ,$$

for $k \in \{k^*, k^\times\}$, where $(\boldsymbol{V}^{(i)})_i$ and $(X^{(i)})_i$ are the random variables defined above for the computation traces and the corresponding public inputs. We then consider the probability $p_{\text{succ}} = \Pr\left(\ell_{k^*} > \ell_{k^\times}\right)$ in Theorem 6.2.

Theorem 6.2. *For a multivariate n^{th}-order DCA attack using the likelihood distinguisher on N traces of length t, the probability that a correct key guess is ranked higher than an incorrect key guess is approximately given by*

$$p_{succ} \approx p_{\mathcal{U}} + (1 - p_{\mathcal{U}}) \left(\frac{1}{2} + \frac{1}{2} \mathrm{erf} \left(\frac{\sqrt{N|\mathcal{V}|}}{2\sqrt{q}} \right) \right)$$

where $q = \binom{t}{n}$ and $p_{\mathcal{U}} = 1 - \left(1 - \left(1 - |\mathcal{V}|^{-1}\right)^q\right)^N$.

The total success probability of the attack $p_{\text{full-succ}}$, i.e., the probability that the correct key guess has the largest likelihood, is then heuristically $p_{\text{full-succ}} \approx p_{\text{succ}}^{|\mathcal{K}|-1}$. Moreover, it can be checked that $p_{\mathcal{U}} \approx N \cdot \left(1 - |\mathcal{V}|^{-1}\right)^q$ becomes negligible as q grows. Theorem 6.2 then implies

$$p_{\text{succ}} = \Theta \left(\mathrm{erf} \left(\frac{\sqrt{N|\mathcal{V}|}}{2\sqrt{q}} \right) \right) ,$$

from which we deduce that the data complexity of the attack is $N = \Theta(q)$. Namely, the number of required traces N to achieve certain p_{succ} is linear in the number of combinations $q = \binom{t}{n}$. We also have

$$N = \Theta \left(\frac{\binom{t}{n}}{|\mathcal{V}|} \right)$$

to show the impact of the definition set $\mathcal{V}$.

In order to prove Theorem 6.2, we introduce the concept of the *zero-counter event*. Denoted by $\mathcal{U}_k$, this is the event that $C_k(v^{(i)}, x^{(i)}) = 0$ for at least one $i \in [1, N]$ for a key guess k. Note that this event can never happen for $k = k^*$, since for all i, there exists a j such that $\bigoplus_{l \in \phi(j)} v_l^{(i)} = \varphi_{k^*}(x^{(i)})$. Thus, $\Pr(\ell_{k^*} > \ell_{k^\times} \mid \mathcal{U}_{k^\times}) = 1$, since in this case the likelihood $\ell_{k^\times}$ equals zero (or equivalently, the log-likelihood equals $-\infty$). This is intuitively sound, as the right key guess could not give rise to a zero counter for any of the N computation traces. Then, by the law of total probability, we can write

$$p_{\text{succ}} = \Pr(\mathcal{U}_{k^\times}) + \Pr(\neg \mathcal{U}_{k^\times}) \cdot \Pr(\ell_{k^*} > \ell_{k^\times} \mid \neg \mathcal{U}_{k^\times}). \tag{6.4}$$

We are therefore interested in the probabilities $\Pr(\mathcal{U}_{k^\times})$ and $\Pr(\ell_{k^*} > \ell_{k^\times} \mid \neg \mathcal{U}_{k^\times})$, which are given in Lemma 6.1 and Lemma 6.2 respectively. Theorem 6.2 then follows directly from Equation 6.4 and these two results.

Probability of the Zero-Counter Event

Lemma 6.1. *Given N traces of length t, the probability of the zero-counter event for a wrong key guess $k^\times$ in an n^{th}-order attack is approximately given by*

$$\Pr(\mathcal{U}_{k^\times}) \approx 1 - \left(1 - \left(1 - |\mathcal{V}|^{-1}\right)^q\right)^N,$$

where $q = \binom{t}{n}$.

Proof. We first define $\mathcal{Z}_k$ as the zero-counter event for key k for a *single* computation trace $\boldsymbol{V}$. Formally,

$$\mathcal{Z}_k \;=\; \text{“ } \forall j \subseteq \left\{1, \ldots, \binom{t}{n}\right\} : \bigoplus_{i \in \phi(j)} V_i \neq \varphi_k(X) \text{ ”}.$$

The zero-counter event $\mathcal{Z}_k$ occurs if and only if none of the $q = \binom{t}{n}$ combinations $\bigoplus_{i\in\phi(j)} V_i$ match the predicted value $\varphi_k(X)$. As discussed, $\mathcal{Z}_{k^*}$ never occurs for the correct key guess k^*. For the incorrect key guess $k^\times$, intuitively, the zero-counter probability $\Pr(\mathcal{Z}_{k^\times})$ should quickly become negligible as the number of combinations q grows. While all q combinations are not strictly independent, we can approximate the probability of $\mathcal{Z}_{k^\times}$ by

$$\Pr(\mathcal{Z}_{k^\times}) \approx \left(1 - \frac{1}{|\mathcal{V}|}\right)^q. \tag{6.5}$$

We verified this approximation by estimating the zero-counter probability over some sampled computation traces. As illustrated in Table 6.2, the obtained estimations match the approximation pretty well.

Table 6.2: Approximation and estimation of the zero-counter probability.

(t, n)	(16,2)	(16,3)	(16,4)	(24,2)	(24,3)	(32,2)	(32,3)
Approximation Equation 6.5	0.625	0.112	$8{\cdot}10^{-4}$	0.340	$4{\cdot}10^{-4}$	0.144	$4{\cdot}10^{-9}$
Estimation (prec. $\sim 10^{-3}$)	0.628	0.135	$< 10^{-3}$	0.342	$< 10^{-3}$	0.145	$< 10^{-3}$

Then, by definition, the zero-counter event for N traces is the union

$$\mathcal{U}_k = \mathcal{Z}_k^{(1)} \vee \mathcal{Z}_k^{(2)} \vee \cdots \vee \mathcal{Z}_k^{(N)},$$

where $\mathcal{Z}_k^{(i)}$ denotes the zero-counter event for k on trace $\boldsymbol{V}^{(i)}$. Taking the negation we obtain

$$\neg\,\mathcal{U}_{k^\times} = (\neg\mathcal{Z}_k^{(1)}) \wedge (\neg\mathcal{Z}_k^{(2)}) \wedge \cdots \wedge (\neg\mathcal{Z}_k^{(N)}),$$

and since the zero events $\mathcal{Z}_{k^\times}^{(i)}$ are mutually independent, we get

$$\Pr(\mathcal{U}_{k^\times}) = 1 - \prod_{i=1}^{N} \Pr(\neg\mathcal{Z}_{k^\times}^{(i)}) = 1 - \left(1 - \Pr(\mathcal{Z}_{k^\times})\right)^N.$$

This finishes the proof. □

Success Probability with No Zero Counters

Lemma 6.2. *Given N traces of length t, let $q = \binom{t}{n}$, and assume that the zero-counter event does not occur. The probability that a correct key guess has a higher likelihood score than a wrong key guess in an n^{th}-order attack is approximately*

$$\Pr(\ell_{k^*} > \ell_{k^\times} \mid \neg \mathcal{U}_{k^\times}) \approx \frac{1}{2} + \frac{1}{2}\operatorname{erf}\left(\frac{\sqrt{N|\mathcal{V}|}}{2\sqrt{q}}\right).$$

If the zero counter event does not occur, we can think of each trace $\boldsymbol{V}^{(i)}$ as a random variable uniformly distributed over $\mathcal{V}^t$. Since the public input $X^{(i)}$ is also random, the counters $C_k(\boldsymbol{V}, X)$ follow some probability distribution. In order to prove Lemma 6.2, we first prove the following result regarding these distributions.

Lemma 6.3. *Let k^* and $k^\times$ be a right and wrong key guess. Let $q = \binom{t}{n}$ and $\kappa = (q-1)\frac{1}{|\mathcal{V}|}$. Then for a trace of length t and an n^{th}-order attack,*

$$C_{k^*}(\boldsymbol{V}, X) \sim \mathcal{N}(\kappa + 1, \kappa)$$

and

$$C_{k^\times}(\boldsymbol{V}, X) \sim \mathcal{N}(\kappa, \kappa)\,,$$

where $\mathcal{N}(\mu, \sigma^2)$ denotes the normal distribution with mean μ and variance σ^2.

Proof. Let $\delta : \mathcal{V}^2 \to \{0, 1\}$ be the function defined as

$$\delta(v_1, v_2) = \begin{cases} 1 & \textbf{if } v_1 = v_2, \\ 0 & \textbf{otherwise}. \end{cases}$$

The counter $C_k(\boldsymbol{V}, X)$ can be rewritten as a sum

$$C_k(\boldsymbol{V}, X) = \sum_{j=1}^{q} \delta(W_j, \varphi_k(X))\,,$$

where the variables $(W_j)_j$ are defined as the $q = \binom{t}{n}$ combinations $\bigoplus_{i \in \phi(j)} V_i$. We recall that for one index j we have $W_j = \varphi_{k^*}(X)$, whereas for the other indices the W_j are randomly distributed independently of X. The counter expectation then satisfies

$$\mathrm{E}\big(C_k(\boldsymbol{V}, X)\big) = \sum_{j=1}^{q} \mathrm{E}\big(\delta(W_j, \varphi_k(X))\big) = \begin{cases} (q-1)\frac{1}{|\mathcal{V}|} & \textbf{if } k \neq k^*, \\ (q-1)\frac{1}{|\mathcal{V}|} + 1 & \textbf{if } k = k^*. \end{cases}$$

On the other hand, the counter variance can be expressed as:

$$\begin{aligned}\operatorname{Var}\big(C_k(\boldsymbol{V}, X)\big) = &\sum_{j=1}^{q} \operatorname{Var}\big(\delta(W_j, \varphi_k(X))\big) \\ &+ 2 \sum_{1 \le j_1 < j_2 \le q} \operatorname{Cov}\big(\delta(W_{j_1}, \varphi_k(X)), \delta(W_{j_2}, \varphi_k(X))\big).\end{aligned}$$

It can be checked that the covariances will be equal to 0 most of the time. Indeed, the covariances are non-zero only when $W_{j_1} \oplus W_{j_2} = \varphi_{k^*}(X)$, which never happens when n is odd and which happens for a few pairs (j_1, j_2) when n is even. Therefore these covariance terms will only have a small impact on the overall variance. Moreover, it can be checked that this impact is negative, i.e., it reduces the variance.[2] Therefore we will ignore the sum of covariances, which yields a correct result when n is odd and a slight overestimation when n is even. We then have

$$\operatorname{Var}\big(\delta(W_j, \varphi_k(X))\big) = \begin{cases} \frac{1}{|\mathcal{V}|}\big(1 - \frac{1}{|\mathcal{V}|}\big) & \textbf{if } j \neq j^*, \\ 0 & \textbf{if } j = j^*, \end{cases}$$

where j^* denotes the index of the right combination matching $\varphi_{k^*}(X)$. Combining the two above equations gives:

$$\operatorname{Var}\big(C_k(\boldsymbol{V}, X)\big) = (q-1)\frac{1}{|\mathcal{V}|}\Big(1 - \frac{1}{|\mathcal{V}|}\Big) \approx (q-1)\frac{1}{|\mathcal{V}|}.$$

Since the counter is defined as a sum of *somewhat independent* random variables, we can soundly approximate its distribution by a Gaussian, and setting

$$\kappa = (q-1)\frac{1}{|\mathcal{V}|}$$

concludes the proof. □

In the above proof, we use that the $\delta(W_j, \varphi_k(X))$ are somewhat independent. By *somewhat independent* we mean that these variables are pairwise independent (for most or all of them, as discussed). Note that variants of the central limit theorem exist that take some form of dependence between the summed variables into account. We have experimentally verified that the Gaussian approximation is sound for various parameters (t, n).

[2] Most of the time we have $\varphi_{k^*}(X) \neq 0$ so that the pairs (j_1, j_2) with $W_{j_1} \oplus W_{j_2} = \varphi_{k^*}(X)$ are such that $W_{j_1} \neq W_{j_2}$ with high probability. In that case $\delta(W_{j_1}, \varphi_k(X)) = 1$ implies $\delta(W_{j_2}, \varphi_k(X)) = 0$ and conversely which yields a negative covariance.

Proof (of Lemma 6.2). Using Lemma 6.3, we can now prove Lemma 6.2. As mentioned, it is more convenient to focus on the log-likelihood in practice, i.e., we consider

$$\Pr(\ell_{k^*} > \ell_{k^\times} \mid \neg\mathcal{U}_{k^\times}) = \Pr\left(\log \ell_{k^*} - \log \ell_{k^\times} > 0 \mid \neg\mathcal{U}_{k^\times}\right),$$
$$\log \ell_{k^*} - \log \ell_{k^\times} = \sum_{i=1}^{N} \underbrace{\log C_{k^*}(\boldsymbol{V}^{(i)}, X^{(i)}) - \log C_{k^\times}(\boldsymbol{V}^{(i)}, X^{(i)})}_{Y^{(i)}} .$$

As introduced above, we denote by $Y^{(i)}$ the difference between the log-counters for the trace $\boldsymbol{V}^{(i)}$. Since the $Y^{(i)}$ are mutually independent and identically distributed, the central limit theorem implies that, for N sufficiently large,

$$\frac{1}{N}(\log \ell_{k^*} - \log \ell_{k^\times}) \sim \mathcal{N}\left(\mu_Y, \sigma_Y^2 N^{-1}\right) \quad \text{with} \quad \begin{cases} \mu_Y = \mathrm{E}(Y), \\ \sigma_Y^2 = \mathrm{Var}(Y), \end{cases}$$

for $Y = \log C_{k^*}(\boldsymbol{V}, X) - \log C_{k^\times}(\boldsymbol{V}, X)$. Thus

$$\Pr(\ell_{k^*} > \ell_{k^\times} \mid \neg\mathcal{U}_{k^\times}) = 1 - \Phi_{\mu_Y, \sigma_Y^2/N}(0) = \frac{1}{2} + \frac{1}{2}\mathrm{erf}\Big(\frac{\sqrt{N}\,\mu_Y}{\sqrt{2}\,\sigma_Y}\Big), \tag{6.6}$$

where $\Phi_{\mu,\sigma}$ is the CDF of $\mathcal{N}(\mu, \sigma^2)$. By the heuristic assumption that $C_{k^*}(\boldsymbol{V}, X)$ and $C_{k^\times}(\boldsymbol{V}, X)$ are mutually independent, and using the Taylor expansion of the logarithm at $\mathrm{E}(C)$, as well as Lemma 6.3, we have

$$\mu_Y \approx \log(\kappa+1) - \frac{\kappa}{2(\kappa+1)^2} - \log\kappa + \frac{\kappa}{2\kappa^2} \approx \frac{1}{\kappa}, \quad \text{and} \quad \sigma_Y^2 \approx 2\frac{\kappa}{\kappa^2} = \frac{2}{\kappa},$$

where the approximation of the mean is sound if κ is large enough (e.g., $\kappa > 10$). Inserting these approximations into Equation 6.6, remembering that

$$\kappa = (q-1)\frac{1}{|\mathcal{V}|} \approx \frac{q}{|\mathcal{V}|},$$

finishes the proof. □

6.4 Experimental Verification and Security Evaluation

The proof of Theorem 6.2 relies on a number of approximations. We therefore verified the accuracy of the estimate by simulating the multivariate higher-order DCA attack for various choices of the masking order n and the size of attacking window t. We chose to simulate traces of a masked and shuffled AES implementation, that is, the target secret variable was taken to be $\varphi_k(x) = S(x \oplus k)$ where S denotes the AES s-box. The computation traces were generated according to the model described at the beginning of Section 6.3.1, namely by sampling random values v_j over $\mathcal{V} = \mathbb{F}_{2^8}$

with the constraint that one randomly chosen n-tuple of each trace has XOR-sum $\varphi_{k^*}(x)$.

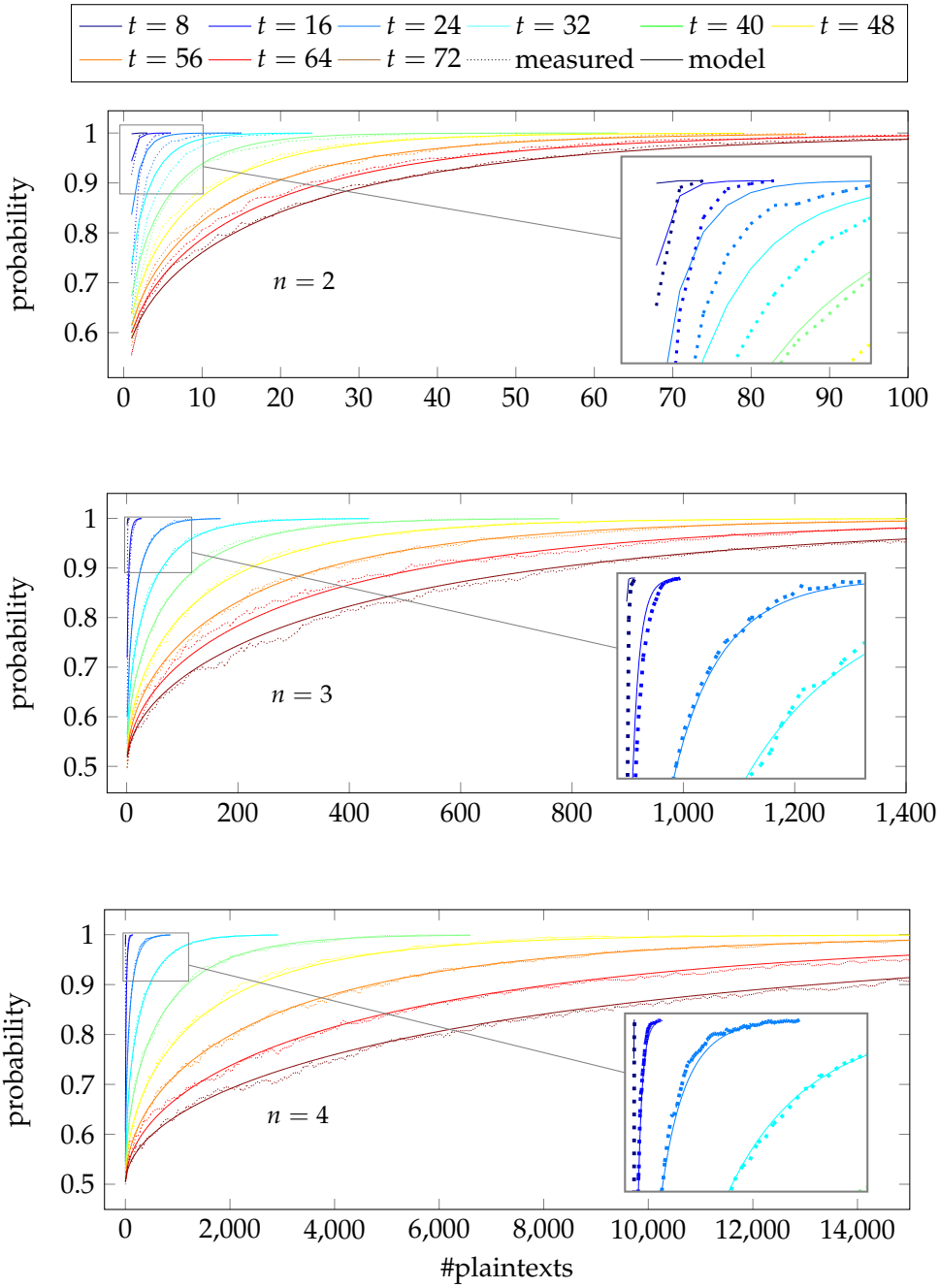

Figure 6.1: The measured probability of ranking a correct key higher than an incorrect key in the multivariate higher-order DCA attack, compared to Theorem 6.2. The measurement is based on 2 000 simulations of the attack. Here, n is the attack order and t is the length of the obtained traces.

We generated traces for $n \in \{2, 3, 4\}$ and $t \in \{8, 16, 24, 32, 40, 48, 56, 64, 72\}$, and calculated the log-likelihood scores for the correct key and a randomly chosen wrong key. This was repeated 2 000 times, and the probability $\Pr(\ell_{k^*} > \ell_{k^\times})$ was calculated for varying values of N. The results are shown in Figure 6.1. The figure shows

that the estimate of Theorem 6.2 is quite accurate in most cases, only deviating from the experimental measurements for very small values of $q = \binom{t}{n}$ ($q < 300$). Note that in practice, this would rarely be a problem. For example, if all shares of the full AES state were shuffled in a first order masked implementation, as described in Section 3.3.3, the smallest trace that would always contain the correct shares would have $q = \binom{2 \cdot 16}{2} = 496$.

The attack complexity of the multivariate higher-order DCA is the same as that of the higher-order DCA, namely $\mathcal{O}\left(|\mathcal{K}| \cdot N_n \cdot \binom{t}{n}\right)$. Using this and Theorem 6.2 we can provide an estimate of the security level obtained by a masked and shuffled implementation against the DCA adversary. Table 6.3 shows the complexities of attacking e.g., a protected AES implementation where the operations are shuffled among all 16 state bytes (the shuffling degree is $\lambda = 16$ implying $t = 16 \cdot n$). When fixing $p_{\text{full-succ}}$ at 90%, we see that an implementation which uses 7^{th} order masking will obtain an estimated security level of 85 bits.

Table 6.3: The number of traces N and the time needed to successfully attack an implementation with $(n-1)^{\text{th}}$-order masking and shuffling of degree $\lambda = 16$ ($t = 16 \cdot n$) using multivariate n^{th}-order DCA. Here, $|\mathcal{K}| = 256$, and we fix the success probability at 90%.

n	$\log_2 N$	$\log_2$ time
3	10.6	32.7
4	15.8	43.1
5	21.0	53.5
6	26.3	64.1
7	31.6	74.6
8	36.9	85.3

Chapter 7

Data Dependency Gray-Box Attacks

Part of the results presented in this chapter have been published in (Goubin, Rivain, et al., 2020).

7.1 Introduction

To prevent DCA-like passive gray-box attacks, it is natural to consider classic side-channel countermeasures, i.e., *linear masking* and *shuffling*. Roughly speaking, linear masking (a.k.a. Boolean masking) splits any sensitive intermediate variable into multiple linear shares and processes them in a way which ensures the correctness of the computation while preventing sensitive information leakage to some extent. The principle of shuffling is to randomly permute the order of several independent operations (possibly including dummy operations) in order to increase the noise in the instantaneous leakage on a sensitive variable. We have shown in Chapter 5 an implementation solely protected with linear masking is vulnerable to *linear decoding analysis* (LDA) which is able to recover the locations of shares by solving a linear system. In Chapter 6, we analyze the combination of linear masking and shuffling and show that it can achieve some level of resistance against advanced gray-box attacks.

At Asiacrypt 2018, (Biryukov and Udovenko, 2018) introduced the notion of *algebraically-secure* non-linear masking to protect white-box implementations against LDA, formalized as *algebraic attacks* in (Biryukov and Udovenko, 2018). Non-linear masking ensures that applying any linear function to the intermediate variables of the protected implementation should not compute a *predictable* variable with probability (close to) 1. However, the non-linear encoding is vulnerable to the DCA attack because the encoded sensitive variable is still linearly correlated with some shares in the non-linear encodings. It was then suggested by the authors to combine linear and non-linear masking, which was conjectured to be able to counter DCA-like and LDA-like attacks at the same time since the adversary is neither able to build predictable variable by a low-degree function over the computation traces, nor able to locate a low number of shares (i.e., lower than the linear masking order) from which she can extract sensitive information.

In this chapter, our contribution is threefold:

1. **Proposal of combination of linear masking and non-linear masking.** We introduce possible ways to combine state-of-the-art countermeasures, namely linear masking and non-linear masking for white-box cryptography (such as the winning implementations of WhibOx 2019) in Section 7.2.

2. **Comprehensive study of advanced gray-box attacks.** In Section 7.3, we briefly recall the advanced gray-box attacks which can be used to break white-box implementations in this context, including higher-degree decoding analysis, (integrated) higher-order DCA. We then analyze their (in)effectiveness against state-of-the-art countermeasures and exhibit their trace and time complexities.

3. **New data-dependency attack.** In Section 7.4, we propose a data-dependency gray-box attack which achieves significant complexity improvements in different attack scenarios by precisely locating the target shares within a computation trace and avoiding the standard combinatorial explosion. We show

that our approach can efficiently break several combinations of linear and non-linear masking in the presence of shuffling and obfuscation.

7.2 Countermeasure Combinations

In this section, we first state the necessity to combine different countermeasures in the white-box context to resist against gray-box attacks. We then propose three different natural ways to combine linear masking and non-linear masking countermeasures.

7.2.1 On the Necessity to Combine Countermeasures

A single defensive line is never enough in front of a white-box adversary. In the following, we exhibit the vulnerability of linear masking or non-linear masking when used alone in the white-box context, which stresses the necessity to combine different countermeasures in order to achieve some level of practical security.

The soundness of the linear masking countermeasure in the noisy-leakage model comes from the fact that the computation complexity exponentially grows with the masking order (Prouff and Rivain, 2013). However, in the white-box context, the adversary can record the values of arbitrary intermediate variables without any noise. Although the exact location of the shares might not be obvious for the adversary because of some obfuscation or obscurity in the implementation structure, linear masking can be completely smashed using a simple gray-box attack. The so-called *linear decoding analysis* (LDA) introduced in Chapter 5 –and also independently discussed in (Biryukov and Udovenko, 2018)– as an effective way to break linear masking (or any other linear encoding scheme) in the white-box model. The complexity of LDA is $\mathcal{O}\left(|\mathcal{K}| \cdot t^{2.8}\right)$ where $\mathcal{K}$ is the key space and t is the window size. Notably, this complexity is independent of the masking order n, as long as all the shares of the target variable appear in the computation trace. This attack was applied to break `Adoring Poitras`, the winning implementation of the WhibOx 2017 competition as presented in Section 8.4.

A DCA adversary can easily break an implementation protected by non-linear masking only. For instance, if the minimalist quadratic masking scheme (Biryukov and Udovenko, 2018) is applied without additional linear masking, a simple first-order DCA is sufficient to break the scheme. Indeed, by definition, the shares a and b are uniformly picked at random which implies that the share $c = x \oplus ab$ is correlated to x. Precisely, we have $\mathrm{Cor}(ab \oplus c, c) = \frac{1}{2}$. A more detailed analysis of (higher-order) DCA against non-linear masking (possibly combined with linear masking) is given in Section 7.3.

7.2.2 Combination of Linear and Non-Linear Masking

As suggested in (Biryukov and Udovenko, 2018), a combination of linear masking (see Section 3.3.1) and non-linear masking (see Section 3.3.2) is empirically secure against both DCA and LDA attacks of order/degree lower than the respective linear masking order and non-linear masking degree. The intuition behind is two-fold: on the one hand, an algebraically-secure countermeasure mixed with linear masking should not decrease the algebraic degree to construct a predictable value; on the other hand, the biased non-linear shares are further linearly masked and only DCA of order n can be used to break a linear masking of order $n-1$.

From (Biryukov and Udovenko, 2018) it is not clear how to combine linear and non-linear masking. In order to discuss the possible attack path, we suggest hereafter three natural ways to combine them. Analyzing the security properties of the resulting combined masking is an interesting follow-up research direction of the present thesis.

In the first two ways, we simply apply one masking scheme on top of the other. Taking the quadratic encoding example in Equation 3.6, the first way is to apply linear masking on top of non-linear masking

$$x = \left(a_1 \oplus a_2 \oplus \cdots \oplus a_n\right)\left(b_1 \oplus b_2 \oplus \cdots \oplus b_n\right) \oplus \left(c_1 \oplus c_2 \oplus \cdots \oplus c_n\right) , \tag{7.1}$$

and the second way is to apply non-linear masking on top of linear masking

$$x = \left(a_1 b_1 \oplus c_1\right) \oplus \left(a_2 b_2 \oplus c_2\right) \oplus \cdots \oplus \left(a_n b_n \oplus c_n\right) . \tag{7.2}$$

The combined masking gadget can be simply derived from the original gadgets of both schemes. For the 1$^{\text{st}}$ combination, one starts from the linear masking gadgets then non-linearly shares each variable and replaces each gate by the corresponding non-linear masking gadget. For the 2$^{\text{nd}}$ combination, one starts from the non-linear masking gadgets then linearly shares each variable and replaces each gate by the corresponding linear masking gadget.

The third way is to merge the two maskings into a new encoding achieving the two features (high order security and prediction security). Also taking as an example the quadratic encoding from (Biryukov and Udovenko, 2018), the new encoding would be

$$x = ab \oplus c_1 \oplus c_2 \oplus \cdots \oplus c_n . \tag{7.3}$$

There are two interpretations of this new encoding. On one hand, the linear part of the non-linear masking in Equation 3.6 is linearly encoded by Equation 3.5

$$x = ab \oplus \left(c_1 \oplus c_2 \oplus \cdots \oplus c_n\right) ,$$

on the other hand, the first share x_1 from linear encoding in Equation 3.5 is non-linearly masked by Equation 3.6

$$x = \big(ab \oplus c_1\big) \oplus c_2 \oplus \cdots \oplus c_n \ .$$

It is not clear how to derive secure gadgets for such encoding. One could probably mix linear masking and non-linear masking gadgets. For instance, one could use the non-linear masking gadgets and replace the appearance of c in by n linear shares for which one would involve the corresponding linear masking gadgets. The exact description and security analysis of these mixed gadgets are beyond the scope of the present thesis.

7.2.3 Bitslicing

In this chapter, we analyze different advanced gray-box attacks against the combinations of countermeasures described in the previous section in the exemplary setting of bitsliced implementations. Bitslicing is a common technique to derive efficient software implementation of a cipher from its Boolean circuit representation (Biham, 1997; Rebeiro et al., 2006). The main idea is to manipulate several data slots in parallel by making the most of bitwise and/or SIMD instructions on modern CPU. Bitslicing has been in particular applied as a strategy to design efficient implementations in the presence of linear masking (Goudarzi and Rivain, 2017; Journault and F.-X. Standaert, 2017; Goudarzi, Jean, et al., 2019; Bellizia et al., 2019). In the context of white-box cryptography, this approach has also been empowered (with additional layers of obfuscation and virtualization) to design implementations with a good level of resistance in practice. In particular, the winning implementations of the two editions of the WhibOx competition, due to Biryukov and Udovenko, were based on this principle (WhibOx, 2017; WhibOx, 2019). In this chapter, we consider a white-box implementation in the paradigm of a randomized Boolean circuit with a hard-coded key represented in software as a bitsliced program.

7.3 Advanced Gray-Box Attacks

We assume that the target implementation is protected by an n^{th}-order linear masking and a d^{th}-degree non-linear masking in one of the three possible composition ways discussed in Section 7.2.2. We will optionally consider the application of shuffling on top of this combination of masking and denote λ the shuffling degree (meaning that the target shares are each shuffled among λ possible locations). Finally, we shall assume that the attacker is able to locate a t-large window in the computation trace for which she knows that it contains the shares of the target encoding. The complexity of the attack discussed in this section shall hence be expressed with respect to the parameters n, d, λ, t, as well as the size of the key space $|\mathcal{K}|$ of the target variable.

7.3.1 Higher-Degree Decoding Analysis

In this section, we first consider HDDA (see Section 5.4) attack against a target implementation with combined masking only and then extend HDDA to deal with shuffling.

HDDA against Combined Masking

Let $(v_1, v_2, \cdots, v_t)$ denote the computation trace corresponding to the t-large target window. By definition of the linear and non-linear masking, we know that there exists a d^{th} degree decoding function f such that $x = f(v_1, v_2, \cdots, v_t)$, where x denotes the target sensitive variable. This function f can be recovered by an HDDA as follows. The principle is first to extend probed traces into higher-degree traces up to degree d by multiplying all the d-tuples, then apply the LDA decoding analysis on the higher-degree traces. Precisely, for each computation trace $(v_1, v_2, \cdots, v_w)$, the adversary compute a higher-degrees trace compromised by all at most d-degree multiplicative combinations $v_{i_1} \cdot v_{i_2} \cdot \ldots \cdot v_{i_j}$ where $1 \leq i_1 \leq i_2 \leq \cdots \leq i_j \leq t$ and $1 \leq j \leq d$. Since a d^{th} degree decoding function f can be decomposed to a linear combination of several (at most d-degree) monomials, an LDA attack on the higher-order traces should recover all monomials of f. The higher-degree traces contain $\mathcal{O}(t^d)$ samples, then the computation complexity of HDDA is $\mathcal{O}(|\mathcal{K}| \cdot t^{2.8\,d})$ and it requires $\mathcal{O}(t^d)$ computation traces, according to Section 5.4.

HDDA in the Presence of Shuffling

If the shuffling countermeasure is applied together with the combination of two masking countermeasures, HDDA would not work in general because there would not exist a decoding function that could recover the predictable values for all different inputs. However, we remark that the HDDA is able to bypass certain shuffling methods. For instance, if there exists λ different sensitive variables

$$\begin{aligned} s_1 &= g(v_{i_1,1}, \cdots, v_{i_1,t}) \\ s_2 &= g(v_{i_2,1}, \cdots, v_{i_2,t}) \\ &\vdots \\ s_\lambda &= g(v_{i_\lambda,1}, \cdots, v_{i_\lambda,t}) \end{aligned}$$

for some decoding function g and $(i_1, i_2, \cdots, i_\lambda)$ is a permutation of $(1, 2, \cdots, \lambda)$ depending on the randomness (i.e., the computation order of $s_1, s_2, \cdots, s_\lambda$ is shuffled correspondingly), there exists a decoding function

$$\begin{aligned} & f(v_{1,1}, \cdots, v_{1,t},\ v_{2,1}, \cdots, v_{2,t},\ \cdots,\ v_{\lambda,1}, \cdots, v_{\lambda,t}) \\ =\ & g(v_{1,1}, \cdots, v_{1,t}) \oplus g(v_{2,1}, \cdots, v_{2,t}) \oplus \cdots \oplus g(v_{\lambda,1}, \cdots, v_{\lambda,t}) \\ =\ & s_1 \oplus \cdots \oplus s_\lambda \end{aligned}$$

over a enlarged attacking trace window

$$(v_{1,1}, \cdots, v_{1,t},\ v_{2,1}, \cdots, v_{2,t},\ \cdots,\ v_{\lambda,1}, \cdots, v_{\lambda,t})\,.$$

Assuming one $s_j, 1 \leq j \leq \lambda$ contains the real execution and the others are simply identical computation except with shuffled constant plaintext, we have $s_1 \oplus \cdots \oplus s_\lambda = s_j + cst$ where cst denotes some unknown constants depending upon the constant plaintext used to compute $(s_{j'})_{j' \neq j}$. In this case, we can still recover function f and the sub-function g. Additionally, shuffling can also be defeated if we are able to enforce some s_j to be constant. For instance, if the targeted $(s_j)_j$ are the first round s-box output, we can make them constant by fixing some plaintext bytes.

7.3.2 Higher-Order DCA

The principle of a higher-order DCA (HO-DCA) is to exploit joint leakage of several independent variables. It consists of a preprocessing step similar to HDDA followed by a standard DCA. Given a computation trace $(v_1, v_2, \cdots, v_t)$, the preprocessing step in an n^{th}-order DCA outputs an n^{th}-order computation trace consisting in $q = \binom{t}{n}$ items formed in $v_{i_1} \oplus v_{i_2} \cdot \ldots \oplus v_{i_n}$ where $1 \leq i_1 \leq i_2 \leq \cdots \leq i_n \leq t$. Then the adversary predicts some sensitive variables based on some key guess and computes the correlations between the predicted values and higher-order trace samples. If there exists a distinguishable significant peak for a key guess from the other key guesses in the correlation traces, this key is very likely to be the good key candidate. Section 6.2 provides a comprehensive formalization and analysis of HO-DCA.

HO-DCA Correlation Scores

Against Combined Masking. We analyze hereafter the expected correlation scores for an HO-DCA against the considered combination of linear and non-linear masking. Our analysis is based on the following simple lemma.

Lemma 7.1. *Let X, A_1, B_1, …, A_n, B_n be mutually independent uniform random variables over $\{0,1\}$. We have*

$$\mathrm{Cor}\left(X, X \oplus \bigoplus_{i=1}^{n} A_i \cdot B_i\right) = \frac{1}{2^n}\,.$$

Proof. Since X and $X \oplus \bigoplus_{i=1}^{n} A_i \cdot B_i$ are both balanced Boolean variables, we have

$$\mathrm{Cor}\left(X, X \oplus \bigoplus_{i=1}^{n} A_i \cdot B_i\right) = \mathrm{Cor}\left(0, \bigoplus_{i=1}^{n} A_i \cdot B_i\right) = 1 - \frac{1}{2^{2n-1}}\left|\bigoplus_{i=1}^{n} A_i \cdot B_i\right|\,.$$

by combining Equation 3.3 and Equation 3.2.

It is obvious that the weight of a 2-ary function $A_1 \cdot B_1$ is $|A_1 \cdot B_1| = 2^{1-1}(2^1 - 1) = 1$. Now we assume the weight of a $(2n)$-ary function $\bigoplus_{i=1}^{n} A_i \cdot B_i$ is

$$\left|\bigoplus_{i=1}^{n} A_i \cdot B_i\right| = 2^{n-1}(2^n - 1)\,,$$

then by induction, the weight of a $(2n+2)$-ary function $\bigoplus_{i=1}^{n+1} A_i \cdot B_i$ is

$$\left|\bigoplus_{i=1}^{n+1} A_i \cdot B_i\right| = \left(2^{n-1}(2^n-1)\right) * 3 + \left(2^{2n} - 2^{n-1}(2^n-1)\right) * 1$$
$$= 2^{(n+1)-1}(2^{n+1}-1) .$$

All in all,

$$\mathrm{Cor}\left(X, X \oplus \bigoplus_{i=1}^{n} A_i \cdot B_i\right) = 1 - \frac{1}{2^{2n-1}} \cdot 2^{n-1}(2^n-1) = \frac{1}{2^n} .$$

□

Based on this lemma, we can derive the correlation scores for the different types of combinations. In each case, an HO-DCA targeting the n shares $c_1, \ldots, c_n$ is possible.

- For Combination 1 (linear masking on top of non-linear masking), we have

  $$c_1 \oplus \cdots \oplus c_n = x \oplus a \cdot b$$

  with $a = a_1 \oplus \cdots \oplus a_n$ and $b = b_1 \oplus \cdots \oplus b_n$ which from Lemma 7.1 implies

  $$\mathrm{Cor}(x,\ c_1 \oplus \cdots \oplus c_n) = \frac{1}{2} .$$

- For Combination 2 (non-linear masking on top of linear masking), we have

  $$c_1 \oplus \cdots \oplus c_n = x \oplus a_1 \cdot b_1 \oplus \cdots \oplus a_n \cdot b_n$$

  which from Lemma 7.1 implies

  $$\mathrm{Cor}(x,\ c_1 \oplus \cdots \oplus c_n) = \frac{1}{2^n} .$$

- For Combination 3 (merged linear and non-linear masking), we have

  $$c_1 \oplus \cdots \oplus c_n = x \oplus a \cdot b$$

  which from Lemma 7.1 implies

  $$\mathrm{Cor}(x,\ c_1 \oplus \cdots \oplus c_n) = \frac{1}{2} .$$

We observe that the second combination (non-linear on top of linear) provides stronger resistance against HO-DCA since the correlation score is exponentially low with respect to the linear masking order. For the two other options, we always obtain a $\frac{1}{2}$ correlation.

In the Presence of Shuffling. Now suppose shuffling is applied on top of the combination of masking. The impact on the correlation score can be simply analyzed thanks to the following lemma.

Lemma 7.2. *Let $(X_i)_{i\in[\lambda]}$ be λ mutually independent and identically distributed random variables. Let $j \in \{1,\ldots,\lambda\}$. Let Y be a random variable such that $\mathrm{Cor}(Y, X_j) = \rho$ and Y is mutually independent of $(X_i)_{i\in[\lambda]\setminus\{j\}}$. Let X^* be the random variable defined by picking i^* uniformly at random over $[\lambda]$ and setting $X^* = X_{i^*}$. We have*

$$\mathrm{Cor}(Y, X^*) = \frac{1}{\lambda} \cdot \mathrm{Cor}(Y, X_j) = \frac{\rho}{\lambda} \,. \tag{7.4}$$

Proof. By definition, we have $\sigma^2(X^*) = \sigma^2(X_j)$ and

$$\mathrm{Cov}(Y, X^*) = \left(1 - \frac{1}{\lambda}\right) \cdot \mathrm{Cov}(Y, X_{i^*\neq j}) + \frac{1}{\lambda}\,\mathrm{Cov}(Y, X_j) = \frac{1}{\lambda}\,\mathrm{Cov}(Y, X_j) \,,$$

which directly yield Equation 7.4. □

According to Lemma 7.2, a shuffling of degree λ implies a reduction of the correlation score by a factor λ. In case of a combination of horizontal shuffling (of degree λ_h) and vertical shuffling (of degree λ_v), the overall shuffling degree is the product of the degrees i.e., $\lambda = \lambda_h \cdot \lambda_v$.

Trace and Time Complexities

According to Equation 3.1, the number of traces necessary for a successful HO-DCA in presence of combined masking and shuffling is

$$N_1 = N_3 = 4\,\eta\,\lambda^2$$

for Combinations 1 and 3, and

$$N_2 = \eta\,4^n\,\lambda^2$$

for Combination 2. Therefore, the time complexity of the HO-DCA attack is

$$\mathcal{O}(N\,t^n\,|\mathcal{K}|) = \begin{cases} \mathcal{O}\big(t^n\,\lambda^2\,|\mathcal{K}|\big) & \text{for Combinations 1 and 3,} \\ \mathcal{O}\big((4t)^n\,\lambda^2\,|\mathcal{K}|\big) & \text{for Combinations 2.} \end{cases}$$

7.3.3 Integrated Higher-Order DCA

Suppose the attacker is able to locate the shuffling and in consequence splits the computation trace into λ subtraces (each of size t) such that the target encoding appears in one of these subtraces (of a random index). She can then apply a so-called integrated attack (Clavier et al., 2000; Rivain, Prouff, and Doget, 2009). The principle is to compute the correlation between the prediction and the sum (over the integers) of the combined samples for the λ subtraces.

The penalty implied by the shuffling after integration is reduced to the square root of its degree λ as formally stated in the following lemma.

Lemma 7.3. *Let $(X_i)_{i\in[\lambda]}$ be λ mutually independent and identically distributed random variables. Let Y be a random variable such that $\mathrm{Cor}(Y, X_j) = \rho$ for some $j \in \{1, \ldots, \lambda\}$ and Y is mutually independent of $(X_i)_{i\in[\lambda]\setminus\{j\}}$. We have*

$$\mathrm{Cor}\left(Y\,,\,\textstyle\sum_i X_i\right) = \frac{1}{\sqrt{\lambda}}\,\mathrm{Cor}(Y, X_j) = \frac{\rho}{\sqrt{\lambda}}\,.$$

Proof. On one hand, we have $\mathrm{Cov}(Y, \sum_i X_i) = \mathrm{Cov}(Y, X_j)$, and on the other hand, we have $\sigma^2(\sum_i X_i) = \lambda\sigma^2(X_j)$. □

If such an integration HO-DCA can be applied, then the number of traces scales down to $N_1 = N_2 = 4\,c\,\lambda$ and $N_2 = c\,4^n\,\lambda$ and the complexities to $\mathcal{O}(t^n\,\lambda\,|\mathcal{K}|)$ for Combinations 1 and 3, and $\mathcal{O}((4t)^n\,\lambda\,|\mathcal{K}|)$ for Combinations 2.

Partial Integration Attack. In the bitslicing circuit-based model considered here, the horizontal shuffling can be easily defeated by applying an integration attack over the different bitslice slots while the vertical shuffling might be harder to remove. In such a case, the shuffling factor in the number of traces becomes $\lambda_h \cdot \lambda_v^2$. And the attack complexity is finally given by

$$\mathcal{O}(N\,t^n\,|\mathcal{K}|) = \begin{cases} \mathcal{O}\big(t^n\,\lambda_h\,\lambda_v^2\,|\mathcal{K}|\big) & \text{for Combinations 1 and 3,} \\ \mathcal{O}\big((4t)^n\,\lambda_h\,\lambda_v^2\,|\mathcal{K}|\big) & \text{for Combinations 2.} \end{cases}$$

7.4 Data-Dependency HO-DCA

In the previous sections, we have analyzed state-of-the-art gray-box attacks against a combination of linear and non-linear masking, possibly strengthened with some shuffling. We have seen that HDDA has complexity at least $\mathcal{O}\big(|\mathcal{K}|\,t^{2.8\,d}\big)$, that is at least $\mathcal{O}\big(|\mathcal{K}|\,t^{5.6}\big)$ with a simple quadratic masking and assuming that shuffling can be defeated (which might not be trivial). On the other hand, HO-DCA has complexity at least $\mathcal{O}\big(|\mathcal{K}|\,t^n\,\lambda^2\big)$, which can be improved to $\mathcal{O}\big(|\mathcal{K}|\,t^n\,\lambda_h\lambda_v^2\big)$ by a partial integration attack in the bitslicing paradigm, where $\lambda = \lambda_h \cdot \lambda_v$ (horizontal shuffling and vertical shuffling). For a typical white-box implementation including some level of obfuscation, the window size t that needs to be considered in practice might range from 10^4 to 10^6, which makes the above attacks very heavy in computation. This motivated us to investigate new attack techniques to overcome this barrier.

In this section, we develop what we shall call *data-dependency* HO-DCA. By exploiting the data dependency graph of the computation, this attack can bypass the exponential factor $\mathcal{O}(t^n)$ brought by the linear masking. The basic principle of our technique relies on the fact that for the considered combination of masking, all the linear shares $c_1, \ldots, c_n$ of some sensitive variable can be recovered by looking at all

the multipliers of a particular intermediate variable, i.e., all the co-operands to this variable through an AND instruction.

7.4.1 Data-Dependency Traces

In our exposition, we consider the target implementation as a (possibly bitsliced) Boolean circuit $\mathcal{C}$. A circuit is a directed acyclic graph (DAG) in which the vertices are *gates* and the edges between two nodes are *wires*. A gate might either be an operation gate (fan-in 2) which outputs the XOR or AND of the input wires, or a constant gate (fan-in 0) which outputs a constant value (0 or 1). The output of a gate might be the input to several gates which means that each gate has arbitrary fan-out. The output of each gate g can be associated with a variable v_g which is a deterministic (Boolean) function of the input plaintext.

We denote the co-operands of a gate g for operation $\circ$ by $\Theta_\circ(g)$, which is composed of the set of gates g' for which $(v_g, v_{g'})$ enters a subsequent $\circ$ gate, where $\circ \in \{\oplus, \otimes\}$. For instance, the set of co-operands for a gate g for an AND operation is denoted by $\Theta_\otimes(g)$. Deriving the set of co-operands for all the gates in a circuit $\mathcal{C}$ can be done in a single pass on the circuit as described in Algorithm 7.1.

Algorithm 7.1 DETECTCOOPERANDS($\mathcal{C}, \circ$)

Input: A Boolean circuit $\mathcal{C}$ and an operator $\circ \in \{\oplus, \otimes\}$

Output: A associative array M mapping from a gate in $\mathcal{C}$ to a set of gates in $\mathcal{C}$

```
1:  M ← empty associative array
2:  for g ∈ GATES(C) do
3:      if g is an ∘ gate then
4:          g1, g2 ← the two incoming gates of g
5:          if M does not have key g1 then
6:              M[g1] ← ∅
7:          end if
8:          if M does not have key g2 then
9:              M[g2] ← ∅
10:         end if
11:         M[g1] ← M[g1] ∪ {g2}
12:         M[g2] ← M[g2] ∪ {g1}
13:     end if
14: end for
```

In this algorithm, we first declare an empty associative array M, which maps a gate in the input circuit $\mathcal{C}$ to a set of gates in $\mathcal{C}$. $M[g]$ hence denotes a lookup operation in M, resulting in the set of gates associated to g. To construct M, we visit all the gates of the circuit, and for each gate with incoming gates g_1 and g_2, we add g_1 to $M[g_2]$

(the set of co-operands of g_2) and we add g_2 to $M[g_1]$ (the set of co-operands of g_1). At the end of the algorithm, for every gate g, $M[g]$ contains the co-operands of g, i.e., $\Theta_\circ(g)$.

As mentioned above, we intuit that the combination of masking can be defeated by targeting the shares contained in a set of multipliers of some intermediate variables. We therefore produce a new computation trace composed of the bitwise sum of each set, as depicted in Algorithm 7.2. For clarity, we abuse the notation by denoting $|M|$ the number of mappings in an associative array M and by denoting v_g the sample output by gate g in a computation trace T. If the intuition is correct, the linear masking should be removed in the new computation trace, leaving non-linear masking and shuffling as the only remaining protections.

Algorithm 7.2 TRACEPROCESSING$(T, \mathcal{C})$

Input: A Boolean circuit $\mathcal{C}$ and a computation trace $T = (v_1, \ldots, v_t)$
Output: A new computation trace $T' = (v'_1, \ldots, v'_{|M|})$

1: $M \leftarrow$ DETECTCOOPERAND$(\mathcal{C}, \otimes)$
2: **for each** g exists in M **do**
3: $\quad \mathcal{G} \leftarrow M[g]$
4: $\quad v'_g \leftarrow \bigoplus_{g' \in \mathcal{G}} v_{g'}$
5: **end for**

Algorithm 7.3 TRACEPROCESSINGSUBSET$(T, \mathcal{C}, n)$

Input: A Boolean circuit $\mathcal{C}$, a computation trace $T = (v_1, \ldots, v_t)$, and an integer n
Output: A new computation trace T'

1: $T' \leftarrow \varnothing$
2: $M \leftarrow$ DETECTCOOPERAND$(\mathcal{C}, \otimes)$
3: **for each** g exists in M **do**
4: $\quad \mathcal{G} \leftarrow M[g]$
5: $\quad T' \leftarrow T' \cup \left\{ \bigoplus_{g \in \mathcal{G}'} v_g : \mathcal{G}' \subset \mathcal{G}, |\mathcal{G}'| \leq n \right\}$
6: **end for**

"Polluted" Multipliers. A trivial method to counter the previous attack is to pollute the set of multipliers $M[g]$ for the sensitive gates g by adding "random" `AND` gates on the premise that the functionality and underlying security assumptions are not affected. In this case, if the sum of random wires is biased, we can still exploit the vulnerability as analyzed in the coming discussion. Alternatively, we could consider summing up all the subsets of cardinality n since there always exists a subset

of $M[g]$ with n elements, where n is the linear masking order and the subset contains all the linear shares of a sensitive variable. This variant computing the sum of a subset of multipliers is presented in Algorithm 7.3, where the linear masking order is expected to be at most n.

7.4.2 Application to Combined Masking

Effectiveness against Combinations 1 & 3. We demonstrate hereafter that our attack can break two out of three combinations of linear masking and non-linear (quadratic) masking. In a linear masking scheme, `AND` gates only appear in secure `AND` gadgets, and the set of multipliers of each linear share is the set of all the shares of the co-operand. In quadratic minimalist masking (Biryukov and Udovenko, 2018), `AND` gates appear in all gadgets. We enumerate in Table 7.1 the set of multipliers of each variable appearing in the `AND` gadget. For each set of operands, we further give the non-zero correlation between the sum over the set of multipliers or any subset and one of the sensitive operand (either $x = ab \oplus c$ or $y = de \oplus f$).

Table 7.1: Multipliers for each variable in `AND` gadget of (Biryukov and Udovenko, 2018).

variable	multipliers	correlation (full set)	correlation (subset)
a	$\{e, m_a\}$	$\mathrm{Cor}(y, e \oplus m_a) = \frac{1}{4}$	$\mathrm{Cor}(y, m_a) = \frac{1}{4}$
b	$\{d, f, r_f\}$	–	$\mathrm{Cor}(y, f) = \mathrm{Cor}(y, d \oplus f) = \frac{1}{2}$
c	$\{e, f\}$	$\mathrm{Cor}(y, e \oplus f) = \frac{1}{2}$	$\mathrm{Cor}(y, f) = \frac{1}{2}$
d	$\{b, m_d\}$	$\mathrm{Cor}(x, b \oplus m_d) = \frac{1}{4}$	$\mathrm{Cor}(x, m_d) = \frac{1}{4}$
e	$\{a, c, r_c\}$	–	$\mathrm{Cor}(x, c) = \mathrm{Cor}(x, a \oplus c) = \frac{1}{2}$
f	$\{b, c\}$	$\mathrm{Cor}(x, b \oplus c) = \frac{1}{2}$	$\mathrm{Cor}(x, c) = \frac{1}{2}$
m_a	$\{a\}$	–	–
r_c	$\{e, r_f\}$	–	–
m_d	$\{d\}$	–	–
r_f	$\{b, r_c\}$	–	–

For Combination 1 (linear masking on top of non-linear masking), the correlations exhibited Table 7.1 hold for any arbitrarily high linear-masking order n. Let us for instance consider the case of the variable c which is multiplied with both e and f. In the presence of linear masking, we get a sharing $(c_1, \ldots, c_n)$ of c which is multiplied by with a sharing $(e_1, \ldots, e_n)$ of e and a sharing $(f_1, \ldots, f_n)$ of f. By definition of the linear-masking `AND` gadget, this means that each share c_j shall be multiplied with all the shares $(e_i)_i$ and all the shares $(f_i)_i$. For every $j \in [n]$, the sum of the multipliers of c_j hence equals $\bigoplus_i e_i \oplus \bigoplus_i f_i = e \oplus f$ which is correlated to y. In the

same way, all the correlations reported in Table 7.1 are persistent to the application of linear masking.

Note that applying refresh gadgets at the linear masking level would not fix this kind of flaw. Indeed assume that the sharing $(c_1, \ldots, c_n)$ would be refreshed between the multiplication with $(e_1, \ldots, e_n)$ and that with $(f_1, \ldots, f_n)$. Then every share of the refresh sharing would be multiplied with all the shares of $(f_1, \ldots, f_n)$ giving rise to f as a sum of multipliers (which is correlated to y).

For Combination 3 (merged linear and non-linear masking), the analysis above for Combination 1 applies similarly. Let us once again consider the variable c is linearly shared. Then a sharing $(c_1, \ldots, c_n)$ of c is multiplied by variable e and a sharing $(f_1, \ldots, f_n)$ of f. By definition of the linear-masking `AND` gadget, this means that each share c_j shall be multiplied with e and all the shares $(f_i)_i$. For every $j \in [n]$, the sum of the multipliers of c_j hence equals $e \oplus \bigoplus_i f_i = e \oplus f$ which is correlated to y. In the same way, all the correlations reported in Table 7.1 are persistent to the application of linear masking. Here as well, the application of refresh gadgets at the linear masking level would not fix this kind of flaw.

Ineffectiveness against Combination 2. We demonstrate that the previous attack is ineffective if non-linear masking is applied on top of linear masking. Two sensitive intermediate variables x and y are encoded as

$$x = (a_1 b_1 \oplus c_1) \oplus (a_2 b_2 \oplus c_2) \oplus \cdots \oplus (a_n b_n \oplus c_n) \tag{7.5}$$

and

$$y = (d_1 e_1 \oplus f_1) \oplus (d_2 e_2 \oplus f_2) \oplus \cdots \oplus (d_n e_n \oplus f_n)$$

respectively in Combination 2. Since in an `AND` gadget for linear masking, each linear share of one variable is required to multiply with all linear shares of another variable, the secure multiplication gadget for xy must consist of applying `AND` gadget for non-linear masking between non-linear encodings (a_i, b_i, c_i) and (d_j, e_j, f_j) for all $1 \leq i, j \leq n$. However, as shown in Algorithm 3.3, the first step in an `AND` gadget for non-linear masking is to refresh the input non-linear encodings. As a consequence, there exist n different refreshed c_i and f_i, denoted by (abuse of the superscript notation) $\left(c_i^{(j)}\right)_{1 \leq j \leq n}$ and $\left(f_i^{(j)}\right)_{1 \leq j \leq n}$ for all $1 \leq i \leq n$. Obviously, $c_i^{(j)} \cdot f_j^{(i)}$ is calculated for all $1 \leq i, j \leq n$ in the overall secure gadget. Without loss of generality, $c_i^{(j)}$'s multipliers are $e_j^{(i)}$ and $f_j^{(i)}$, and $\mathrm{Cor}\left(y, e_j^{(i)} \oplus f_j^{(i)}\right) = 0$. In Section 7.4.3, we generalize the attack to an advanced one and exhibit its ability to defeat Combination 2.

7.4.3 Generalized Data-Dependency HO-DCA

The outcomes of a gate g for an operation $\circ$ is denoted by $\Psi_\circ(g)$, which is the set of gate computing $v_g \circ v_{g'}$ for another gate g'. Deriving the set of outcomes for all

the gates in a circuit can also be done in a single pass on the circuit as described in Algorithm 7.4.

Algorithm 7.4 DETECTOUTCOME$(\mathcal{C}, \circ)$

Input: A Boolean circuit $\mathcal{C}$ and an operator $\circ \in \{\oplus, \otimes\}$

Output: A associative array N mapping from a gate in $\mathcal{C}$ to a set of gates in $\mathcal{C}$

1: $N \leftarrow$ empty associative array
2: **for** $g \in$ GATES$(\mathcal{C})$ **do**
3: **if** g is an $\circ$ gate **then**
4: $g_1, g_2 \leftarrow$ the two incoming gates of g
5: **if** N does not have key g_1 **then**
6: $N[g_1] \leftarrow \varnothing$
7: **end if**
8: **if** N does not have key g_2 **then**
9: $N[g_2] \leftarrow \varnothing$
10: **end if**
11: $N[g_1] \leftarrow N[g_1] \cup \{g\}$
12: $N[g_2] \leftarrow N[g_2] \cup \{g\}$
13: **end if**
14: **end for**

Algorithm 7.5 GENERALIZEDTRACEPROCESSING$(T, \mathcal{C}, \circ, \star, n)$

Input: A Boolean circuit $\mathcal{C}$, a computation trace $T = (v_1, \ldots, v_t)$, and two operators $\circ, \star \in \{\oplus, \otimes\}$, and an integer n

Output: A new computation trace T'

1: $N_\circ \leftarrow$ DETECTOUTCOME$(\mathcal{C}, \circ)$
2: $M_\star \leftarrow$ DETECTCOOPERANDS$(\mathcal{C}, \star)$
3: $T' \leftarrow \varnothing$
4: **for each** g exists in $N_\circ$ **do**
5: $\mathcal{G} \leftarrow N_\circ[g]$
6: $\mathcal{G}' \leftarrow \bigcup_{g \in \mathcal{G}} M_\star[g]$
7: $T' \leftarrow T' \cup \left\{ \bigoplus_{g \in \mathcal{G}''} v_g : \mathcal{G}'' \subset \mathcal{G}', |\mathcal{G}''| \leq n \right\}$
8: **end for**

We generalize the trace preprocessing step in Algorithm 7.5, in which we first derive the outcomes $\Psi_\circ(g)$ of each gate g for operation $\circ$ and the "secondary" co-operands for operands $\star$ (which is the union of the $\Theta_\star(g')$ for all $g' \in \Theta_\circ(g)$). The adversary has the flexibility to choose $\circ$, $\star$, and the expected linear masking order

n. Finally, she applies the HO-DCA on the preprocessed traces according to data-dependency.

Effectiveness against Combination 2. We show that if choosing $(\circ, \star) = (\oplus, \otimes)$ in the attack described above, the Combination 2 is vulnerable. Here, we consider again a share c_i in Equation 7.5, then c_i have n refreshed version $(c_i^{(j)})_{1 \leq j \leq n}$ and $\{c_i^{(j)} : 1 \leq j \leq n\} \subset \Psi_{\oplus}(c_i)$ for $1 \leq i \leq n$. As already addressed in last section $\Theta_{\otimes}(c_i^{(j)}) = \{e_j^{(i)}, f_j^{(i)}\}$, the union $\bigcup_{i=1}^{n} \Theta_{\otimes}(c_i^{(j)})$ contains all $\{e_1^{(i)}, f_1^{(i)}, e_2^{(i)}, f_2^{(i)}, \cdots, e_n^{(i)}, f_n^{(i)}\}$ and

$$\mathrm{Cor}\left(y, e_1^{(i)} \oplus f_1^{(i)} \oplus e_2^{(i)} \oplus f_2^{(i)} \oplus \cdots \oplus e_n^{(i)} \oplus f_n^{(i)}\right) = \frac{1}{2^n}$$

by Lemma 7.3, implying that the HO-DCA bias is still exploitable.

Further Improvements. The principle of our data-dependency attack can be used in many additional ways. Essentially, our technique enables to derive a cluster of intermediate variables related to each gate in a circuit. By targeting close neighbors of a gate from a gadget processing the encoding of some target intermediate variable, we might succeed in getting all the shares in a small cluster. By doing this, we essentially prevent the exponential explosion of the window size w which could be leveraged in any kind of gray-box attack (such as LDA or HDDA for instance). In fact, the explosion still exists in some attacks but over localized small windows of traces instead of the full trace. There are many possible ways to extend the attack above by playing with Algorithm 7.1, Algorithm 7.4 and Algorithm 7.5. In the trace preprocessing algorithm, we can for instance iteratively apply the data-dependency algorithm a few times, such that the new trace encompasses a relevant cluster of intermediate variables.

7.5 Comparison between Different Advanced Attacks

In this section, we compare the trace and computation complexities between different advanced gray-box attacks discussed in this chapter. The target implementation is protected by an n^{th}-order linear masking and the first-degree secure non-linear masking from (Biryukov and Udovenko, 2018) in one of the three possible composition ways discussed in Section 7.2.2. We optionally consider the application of shuffling of degree λ on top of this combination of masking. We also assume that the attacker is able to locate a t-large window in the computation trace which contains all shares of the target encoding. The complexity of the attacks is expressed with respect to the parameters the linear masking order n, the shuffling degree λ, the size of trace window t and the size of the key space $|\mathcal{K}|$, as shown in Table 7.2.

Table 7.2: Complexity comparison between different advanced attacks. where three masking combinations from Section 7.2.2 are considered. Recall that η is the constant factor in Equation 3.1.

	trace complexity		**computation complexity**	
	masking composition		masking composition	
	1^{st}, 3^{rd}	2^{nd}	1^{st}, 3^{rd}	2^{nd}
	without shuffling			
HDDA	$t^d + \mathcal{O}(1)$		$\mathcal{O}(\lvert\mathcal{K}\rvert \cdot t^{2.8d})$	
HODCA	$4\,\eta$	$4^n\,\eta$	$\mathcal{O}(\lvert\mathcal{K}\rvert \cdot t^n)$	$\mathcal{O}(\lvert\mathcal{K}\rvert \cdot (4t)^n)$
DD-DCA	$4\,\eta$	$4^n\,\eta$	$\mathcal{O}(\lvert\mathcal{K}\rvert \cdot t)$	$\mathcal{O}(\lvert\mathcal{K}\rvert \cdot 4^n \cdot t)$
	with shuffling of degree λ			
HO-DCA	$4\,\eta\,\lambda^2$	$4^n\,\eta\,\lambda^2$	$\mathcal{O}(\lvert\mathcal{K}\rvert \cdot t^n \cdot \lambda^2)$	$\mathcal{O}(\lvert\mathcal{K}\rvert \cdot (4t)^n \cdot \lambda^2)$
Intg. HO-DCA	$4\,\eta\,\lambda$	$4^n\,\eta\,\lambda$	$\mathcal{O}(\lvert\mathcal{K}\rvert \cdot t^n \cdot \lambda)$	$\mathcal{O}(\lvert\mathcal{K}\rvert \cdot (4t)^n \cdot \lambda)$
DD-DCA	$4\,\eta\,\lambda^2$	$4^n\,\eta\,\lambda^2$	$\mathcal{O}(\lvert\mathcal{K}\rvert \cdot t \cdot \lambda^2)$	$\mathcal{O}(\lvert\mathcal{K}\rvert \cdot 4^n \cdot t \cdot \lambda^2)$
Intg. DD-DCA	$4\,\eta\,\lambda$	$4^n\,\eta\,\lambda$	$\mathcal{O}(\lvert\mathcal{K}\rvert \cdot t \cdot \lambda)$	$\mathcal{O}(\lvert\mathcal{K}\rvert \cdot 4^n \cdot t \cdot \lambda)$

Specifically, higher-degree decoding analysis (HDDA) is only able to attack the implementation without shuffling protection and its complexity is independent with the ways to combine linear and non-linear maskings. Integrated (abbv. Intg.) attack variants amplify the correlation score by a factor of $\sqrt{\lambda}$, hence reduce the trace complexity by a factor of λ compared to the non-integrated attack according to Equation 3.1, resulting in a reduction of a factor of λ in the computation complexity. Although data-dependency based attacks do not increase the correlation score, but avoid the exponential explosion when precomputing higher-order traces, hence improve the computation complexity by a factor of t^{n-1} accordingly. Note that the complexity of DD-DCA is also impacted by the number of co-operands, which is usually small than a constant upper bound and hence is neglected here.

Chapter 8

Practical Attacks

Part of the results presented in this chapter have been published in (Goubin, Paillier, et al., 2020; Rivain and Wang, 2019; Goubin, Rivain, et al., 2020).

8.1 Introduction

In this chapter, we verify the practicability of our theoretical analyses and attack techniques exhibited from Chapter 4 to Chapter 7 by performing key-recovery attacks against several publicly available white-box AES implementations. We organize this chapter as follows:

- In Section 8.3, we successfully carry out all of three gray-box attacks, described in Chapter 4, on a white-box AES implementation protected with byte encodings which DCA has failed to break before, and on a "masked" white-box AES implementation which intends to resist DCA;
- In Section 8.4, we explain how to gradually extract the key from the winning implementation of WhibOx 2017 (WhibOx, 2017) – `Adoring Poitras` with (a) a comprehensive presentation of the performed reverse engineering, and (b) a final generic LDA attack, formalized in Chapter 5, within a highly reduced trace window;
- In Section 8.5, we apply advanced gray-box attacks and our data-dependency DCA presented in Chapter 7 together with some "lightweight" de-obfuscation to break the three winning implementations from WhibOx 2019 (WhibOx, 2019).

To the best of our knowledge, we were either the *only* or the *first* team to produce technical reports on the possibility to break those implementations by applying relevant theoretically supported attacks. To facilitate the reproduction of our results, we partially open-source our attacks against the winning submission from WhibOx 2017 in

`https://github.com/junwei-wang/WhibOx-breaking-adoring-poitras`

and those from WhibOx 2019 in

`https://github.com/CryptoExperts/breaking-winning-challenges-of-whibox2019`

respectively. While accomplishing these attacks, we also conclude a general attack methodology against obscure white-box implementations. In the following, we first depict our attack methodology in Section 8.2.

8.2 A Generic White-Box Attack Methodology

We present hereafter a general attack methodology to break obscure white-box implementations following the outline of our cryptanalysis of the winning implementations in two editions of WhibOx competition as will be shown in Section 8.4 and Section 8.5. This methodology is organized into the five following steps. Note that depending on the white-box implementation, it might not be necessary to apply all these steps.

Step 1: Initial Reverse Engineering

The targeted implementation is usually protected by several obfuscation techniques (as e.g., described in (Collberg et al., 1997)). This first step consists in removing these obfuscation layers, either manually or by using some software tools (Yadegari et al., 2015). The goal is to understand the role of each part of the code and remove any virtualization. Of course, this step is difficult to fully generalize and automate. It should somehow rely on some human handwork and intuition. The ultimate goal, for the next steps of our methodology, is to transform the implementation into an arithmetic circuit (or a Boolean circuit as a particular case). Namely, this first step must produce a straight-line program (i.e., without conditional branching) in which every instruction is of the form $v_i \leftarrow v_j * v_\ell$ for some operation $*$ lying in a defined set of operations. For instance, in the Boolean case, we would have $* \in \{\oplus, \wedge, \vee\}$. But a white-box implementation could be defined over a larger finite field (such as $\mathbb{F}_{2^n}$ or $\mathbb{F}_p$), an integer/polynomial ring, etc. An arithmetic circuit would then be composed of additions and multiplications. But some more complicated operations could occur and in all generality, which could be represented, e.g., by lookup tables, taking possibly more than two input operands.

Step 2: SSA Transformation

In the current representation of arithmetic circuit, many intermediate variables are possibly both written and read several times, which presumably hides some facts on the data flow. In compiler theory, a program in the *single static assignment* (SSA) form means that every variable is assigned (defined or written) once, but can be read for multiple times after its assignment (the memory used in an SSA formatted program is then about its number of instructions). The SSA form of a program thus looses the data dependency by reducing the meaningful interlaced dependencies introduced by variable reuse. In order to transform our Boolean circuit into SSA form, we rewrite through the few following steps:

1. Declare a global counter $c = 0$, and an empty associative map (hashmap) H.
2. For each statement, replace
   (a) each of its reading address $\texttt{addr}_r$ with $H(\texttt{addr}_r)$,
   (b) and its writing address $\texttt{addr}_w$ with c,

   then we set $H(\texttt{addr}_w) = c$ and $c = c + 1$.

After this transformation, the program is in SSA form: every memory location is written exactly once and only read after its assignment.

Step 3: Circuit Minimization

In this step, we attempt to minimize the program in several aspects. Our goal here is to decrease the computation complexity in the subsequent analysis techniques that

will then target a smaller circuit. We define a few minimization steps (described below) and we iterate over these steps several times until we cannot reduce it any more.

Detection and Removal of Constants

We execute the Boolean circuit a large number of N times (e.g., $N = 2048$) with randomly sampled inputs and record the *computation traces* (which consist of the ordered sequences of written values). Then, for each location in these computation traces, we check if the written value is always the same. Formally, denoting the i^{th} computation trace by $(v_1^{(i)}, v_2^{(i)}, \cdots, v_t^{(i)})$, where t is the size of the trace (i.e., the number of Boolean instructions), we check whether

$$v_j^{(1)} = v_j^{(2)} = \cdots = v_j^{(N)} = c \in \mathbb{F},$$

for some index j and for sufficiently large N and for the base field of the circuit. If so, we consider that the j^{th} instruction calculates a constant and we replace the corresponding variable by the constant c.

For Boolean circuit, we further propagate this constant according to the following Boolean relations:

$$\begin{aligned} v \wedge 0 &= 0, \quad v \wedge 1 = v, \\ v \vee 0 &= v, \quad v \vee 1 = 1, \\ v \oplus 0 &= v, \quad v \oplus 1 = \neg v, \end{aligned}$$

where $v \in \mathbb{F}_2$. This propagation results in the saving of further instructions.

In an idealized model where all the variables are uniformly distributed, the probability of incorrect decision is $|\mathbb{F}|^{-N}$. The complexity to perform the detection is of $\mathcal{O}(N \cdot t)$.

Detection and Removal of Duplicates

We proceed in a similar way as above to detect and remove duplicates. Namely, we observe whether for two locations in the computation traces the written values are always the same. Formally, we check whether

$$(v_{j_1}^{(1)} = v_{j_2}^{(1)}) \wedge (v_{j_1}^{(2)} = v_{j_2}^{(2)}) \wedge \cdots \wedge (v_{j_1}^{(N)} = v_{j_2}^{(N)}),$$

for some pair of indexes (j_1, j_2) and for sufficiently large N. If so, we consider that the related statements are duplicated computations and that the j_1^{th} and j_2^{th} variables are a pair of duplicates. Then we remove one of the instances and replace all its appearances in the program by the other variable.

As above, the probability of incorrect decision in an idealized model is of $|\mathbb{F}|^{-N}$. The complexity to perform the detection is of $\mathcal{O}\left(N \cdot t^2\right)$.

Detection of Inverse

The detection of Boolean inverses is similar to the detection of duplicates but instead, we check whether

$$v_{j_1}^{(1)} \oplus v_{j_2}^{(1)} = v_{j_1}^{(2)} \oplus v_{j_2}^{(2)} = \cdots = v_{j_1}^{(N)} \oplus v_{j_2}^{(N)} = 0\,,$$

for some pair of indexes (j_1, j_2) and for sufficiently large N. If so, we can replace the statement computing v_{j_2} by the inverse of v_{j_1} (assuming $j_1 < j_2$), which is likely to induce further simplifications while looping on the minimization steps.

Detection and Removal of Pseudorandomness

Here we look for *pseudorandom* variables that are used to randomize subsequent intermediate results without affecting the final result. In order to check whether an intermediate variable serves as pseudorandom variables, we try to replace its value by random values and check whether the output always matches the output in a normal execution. Formally, denoting $x^{(i)}$ and $y^{(i)}$ the input and output of the i^{th} execution, we replace the j^{th} variable by assigning a random value over $\mathbb{F}$. Then we check whether

$$(y^{(1)} = z^{(1)}) \wedge (y^{(2)} = z^{(2)}) \wedge \cdots \wedge (y^{(N)} = z^{(N)}),$$

where $z^{(i)}$ denotes the output of the execution with the flipping statement on input $x^{(i)}$. If so, we consider v_j to be a pseudorandom variable and we replace it by a constant, e.g., 0. This constant is then propagated as described above which results in the saving of further instructions.

The probability of incorrect decision is not clear but should quickly become negligible as N grows (as v_j might affect several bits of the output). The complexity to perform the detection is of $\mathcal{O}(N \cdot t)$.

Note that a variable might impact the output result and be used as pseudorandomness at the same time. In the above detection, we can only detect the variables used exclusively as pseudorandom variables. Rather than replacing an intermediate variable, a more effective way is to replace an operand in a statement. In this sense, the replaceable operand corresponds to a pseudorandom usage of the variable and it can be replaced by a constant.

Detection and Removal of Dead (Dummy) Code

A dead statement is an instruction writing a value which is never used in the subsequent computation. Dead might be introduced by the above minimization steps or by the removal of subsequent dead code. The detection and removal process is a progressive iteration procedure.

Step 4: Data Dependency Analysis

In order to extract the key from a white-box implementation, it is usual to focus on some specific early round operations, e.g., the first round s-boxes in a block cipher. The goal of this step is to reduce the size of the trace window that will be analyzed in the next step.

Observing the *data dependency graph* (DDG) is very insightful to locate a given operation depending on the structure of the target cryptographic algorithm. This operation can be partly automated through a cluster analysis (see our breaking of `Adoring Poitras` in Section 8.4). Once the target operation has been localized, we identify the corresponding set of outgoing variables which presumably constitutes an encoding of the target variable. An alternative is to rely on the approach introduce in Chapter 7. In the former case, one could probably directly locate the good window, but it involves some human analysis. The latter can be fully automated, but this approach is less effective in substantially decreasing the attack complexity compared to an accurate localization of the target operation.

Step 5: Key Extraction

This last step consists in extracting some key information by analyzing the (presumed) encoding obtained from the data dependency analysis. To this purpose, one could for instance rely on several passive gray-box attack techniques presented in the previous chapters, such as (higher-order and/or multivariate) DCA, collision attack, MIA, (higher-degree) LDA, etc. The principle is then to collect some *computation traces* for the windows (or sets of variables) indentified by the data dependency analysis. One then makes a key guess and predicts the value of some key-dependent intermediate variable. Afterward, one applies the selected gray-box distinguisher to the collected traces and the predicted values.

8.3 Attacks against Internal Encodings Protected Implementations

In this section, we successfully carry out all gray-box attacks described in Chapter 4, namely DCA attack, collision attack, and MIA attack, on an implementation protected with byte encodings –which DCA has failed to break before–and on a "masked" white-box AES implementation –which intends to resist DCA. We first introduce the two target implementations and our target variables in Section 8.3.1 and Section 8.3.2 respectively. Then the attack results from the DCA attack, the collision attack, and the MIA attack are presented in Section 8.3.3, Section 8.3.4, and Section 8.3.5 correspondingly.

8.3.1 Target Implementations

The NSC Variant. In NoSuchCon (NSC) 2013, a Windows binary embedding with a white-box implementation of AES-128 protected by external and internal encodings, was published by Vanderbéken as a challenge.[1] Due to the presence of external encodings, the authors of (Bos et al., 2016) failed to break this challenge with DCA. Besides the challenge binary, Vanderbéken also published its generator, which led to the publication of multiple variants.[2] One interesting variant, referred as *the NSC variant* in the following, is an implementation protected with internal encodings only (i.e., an implementation of the standard AES-128). However, since it makes use of byte encodings, this implementation was believed to resist DCA.

Lee's CASE 1 Implementation. Recently, Lee et al. published a white-box implementation of AES secure against the DCA attack (Lee et al., 2018) which consists in applying additional countermeasures to the original implementation of Chow et al.. Three protection cases are suggested: CASE 1 uses some masking techniques before applying internal encodings to protect the first and last round of the implementation; CASE 2 and CASE 3 work as CASE 1 but with larger internal encodings (byte instead of nibbles) applied to more variables in the outer rounds. An implementation (under the form of a binary program) was made publicly available for CASE 1[3] but not for CASE 2 nor CASE 3. In this work, we do not intend to give a full cryptanalysis of Lee's proposal, but we point out that the masking technique can be bypassed by targeting an input byte of the second round, i.e., an output byte of the first-round MixColumn, which allows us to apply a simple DCA-like attack as described below.

8.3.2 Target Variables

In all our experiments, we select one MixColumn output byte in the first round as our target. Such a byte s satisfies

$$\begin{aligned} s &= \text{MixColumn}\big(S(x_1 \oplus k_1^*), S(x_2 \oplus k_2^*), S(x_3 \oplus k_3^*), S(x_4 \oplus k_4^*)\big) \\ &= S(x_1 \oplus k_1^*) \oplus S(x_2 \oplus k_2^*) \oplus 2 \cdot S(x_3 \oplus k_3^*) \oplus 3 \cdot S(x_4 \oplus k_4^*)\,, \end{aligned}$$

where S denotes the AES s-box, (x_1, x_2, x_3, x_4) are four plaintext bytes, and the above multiplications (by 2 and 3) are in the field $\mathbb{F}_{2^8}$.

In order to reduce the key space for guessing such a byte, we select some random plaintext with fixed values for x_3 and x_4 (specifically $x_3 = x_4 = 0$). Doing so, the byte s can be rewritten as $s = S(x_1 \oplus k_1^*) \oplus S(x_2 \oplus k_2^*) \oplus c^*$ for some (secret) constant c^* and the encoded byte $\varepsilon(s)$ is identically distributed (over a random choice of ε) to the byte $\varepsilon\big(S(x_1 \oplus k_1^*) \oplus S(x_2 \oplus k_2^*)\big)$. We then make a 2-byte key guess (k_1, k_2) and

[1] See http://www.nosuchcon.org/2013/ and http://seclists.org/fulldisclosure/2013/Apr/133.
[2] See https://github.com/SideChannelMarvels/Deadpool/tree/master/wbs_aes_nsc2013_variants.
[3] See https://github.com/SideChannelMarvels/Deadpool/tree/master/wbs_aes_lee_case1.

calculate the predictions of the target byte as

$$\varphi_{k_1,k_2}(x_1, x_2) = S(x_1 \oplus k_1) \oplus S(x_2 \oplus k_2)$$

for each plaintext with bytes (x_1, x_2). Our selection function φ is hence a $(16, 8)$-VBF and the size of the key space is 2^{16}.

Although we only focus on recovering two key bytes, we could easily repeat this attack by swapping the fixed pair of bytes to recover k_3^* and k_4^* and do the same for the three other MixColumn computations. Moreover, we can fix a pair of bytes in each column while collecting the traces, so that only two sets of traces would be sufficient to recover the full key.

8.3.3 DCA Attacks

For each implementation, some preliminary analysis of the binary allowed us to obtain the addresses of the executable code segment in the virtual memory. We were then able to collect the bytes written on the stack during an execution. For such a purpose we used the SideChannelMarvels Tracer tool.[4] We thus obtained computation traces composed of 1850 and 21536 byte samples for the NSC variant and Lee's CASE 1 implementation respectively. Each of these samples was split into bits to get binary traces from which we removed the duplicated columns. We finally obtained binary computation traces composed of 6077 and 24012 samples for the NSC variant and Lee's CASE 1 implementation respectively.

For the NSC variant, we collected 1,800 traces from which we computed the correlation traces for each key guess as described in Section 3.2.1. As an illustration, Figure 8.1 plots the correlation traces obtained when the least significant bit of the target variable is used as the prediction function. The attack was conducted for the 8 prediction bits, and the correct key guess was ranked first (among the 2^{16} guesses) in 4 out of 8 attacks.

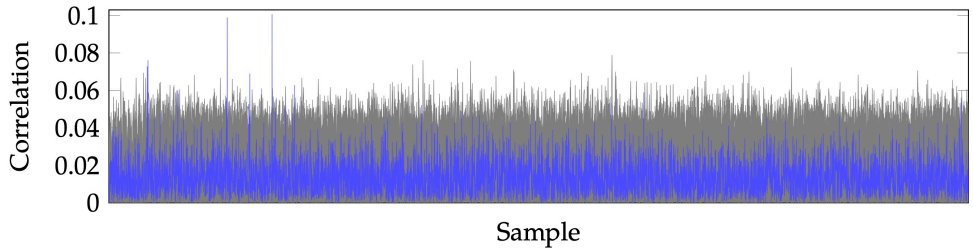

Figure 8.1: DCA correlation traces on the NSC variant for the good key guess (in blue) and 256 (out of $2^{16} - 1$) incorrect key guesses (in gray).

One can observe multiple peaks in the correlation trace for the right key guess that might correspond to different manipulations of the target variable through different encodings ε. It is worth noting that, as exhibited in our analysis, these peaks

[4] See https://github.com/SideChannelMarvels/Tracer.

converge towards correlation scores which are multiples of 2^{-m+2} i.e., of $\frac{1}{64}$ for $m = 8$. The first clearly distinguishable peak is around $\frac{5}{64} \approx 0.078$ while the two next peaks are around $\frac{6}{64} \approx 0.094$ (the match is not perfect due to the estimation error).

Using 1800 traces implies that the average noise in the correlation trace is around $\sqrt{1/1800} \approx 0.02$ (which reaches 0.06 when we take the max over 256 correlation traces as we observe in Figure 8.1). That is why only the peaks around $\frac{6}{64}$ are clearly distinguishable from the noise and that is why the maximal peak is reached for the good key guess for only 4 selection bits out of 8. Taking more traces would certainly ensure that smaller multiples of $\frac{1}{64}$ could also be distinguishable from the noise which would increase the number of prediction bits (up to 8) for which the attack works.

For Lee's CASE 1 implementation, we were able to mount a successful attack using 4000 traces. The obtained correlation traces are very similar to Figure 8.1 which all includes a few distinguishable peaks for the right key guess.

8.3.4 Collision Attacks

We also experiment our collision attack against the NSC variant and Lee's CASE 1 white-box implementation. We use the same target variable as in our DCA experiments, which is a MixColumn output in the first round, turned into a $(16, 8)$-VBF –with a key space of size 2^{16}– by fixing two input bytes (see Section 8.3.2 for details).

Our collision attack recovers the two key bytes using 60 computation traces only (which is to be compared with the 1800 traces required by DCA). As an illustration, Figure 8.2 plots the correlation traces for the correct key guess and for 256 (out of $2^{16} - 1$) incorrect key guesses. We observe some correlation peaks to 1 for the correct key. For the key guess ranked second, we observe a few peaks reaching 0.5 whereas for the other key guesses most of the peaks are around 0.25 or lower than this value.

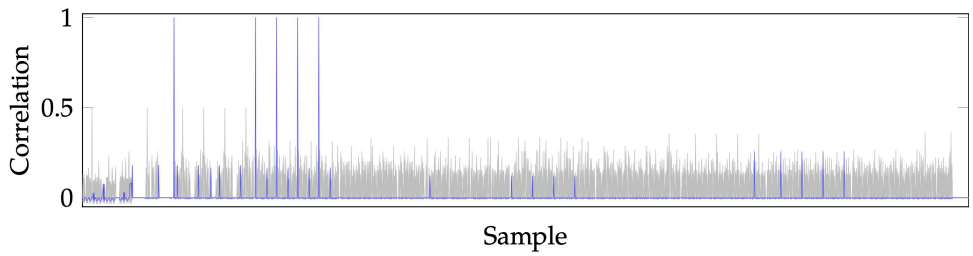

Figure 8.2: Collision attack traces on the NSC variant for the good key guess (in blue) and 256 (out of $2^{16} - 1$) incorrect key (in gray).

The same collision attack has been applied to Lee's CASE 1 implementation and could also recover the correct (two-byte) key guess using 60 traces. Although no implementations of Lee's CASE 2 & 3 are publicly available, we note that these variants (which consist in applying byte encodings to internal rounds) should not prevent our collision attack.

8.3.5 MIA Attacks

We perform practical experiments for the generic and improved MIA against the NSC variant implementation. We use the same target variable as in our DCA and collision attack experiments, which is a MixColumn output in the first round, turned into a $(16, 8)$-VBF –with a key space of size 2^{16}– by fixing two input bytes (see Section 8.3.2 for details). For each attack, we first perform some preprocessing step in order to speed up the attack.

Preprocessing. In order to save some computation, we detect all the "informationally equivalent" samples in the collected traces, namely indexes i and j for which

$$\hat{\mathrm{H}}(V_i) = \hat{\mathrm{H}}(V_j) = \hat{\mathrm{I}}(V_i\,;\,V_j)\,,$$

or equivalently for which there is a one-to-one relation between the samples $(v_i^{(\ell)})_{\ell\in[N]}$ and $(v_j^{(\ell)})_{\ell\in[N]}$. We then remove these "informational" duplicates (i.e., we either drop the column i or the column j in the set of computation traces). This can be done efficiently by grouping the indexes for which $\hat{\mathrm{H}}(V_i)$ has a given value and only computing $\hat{\mathrm{I}}(V_i\,;\,V_j)$ within each group. This preprocessing allows us to compress the computation traces by a factor 10 (from 1850 samples to 185), which is especially interesting given the large key space $|\mathcal{K}| = 2^{16}$.

Standard MIA. We could recover the two key bytes (ranked first) with standard MIA using 115 traces. As an illustration, we plot the mutual information traces for the right key guess and 256 (out of $2^{16} - 1$) in Figure 8.3. These are obtained using 150 computation traces in order to make the right guess peak clearly visible.

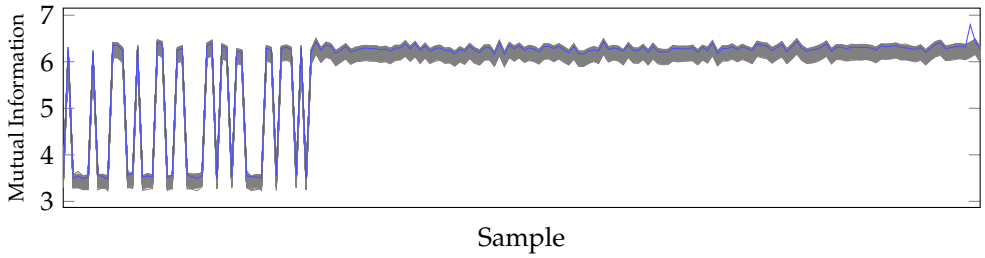

Figure 8.3: Mutual information traces for the correct key guess (in blue) and 256 (out of $2^{16} - 1$) incorrect key guesses (in gray).

Improved MIA Attack. Our experiments show that the improved MIA attack substantially decreases the number of required traces. Namely, we can recover the right key guess using 70 traces. We observe that only the correct key guess has a positive score, i.e., $\hat{\mathrm{H}}(\varphi^*) \approx 5.89$, and all the incorrect key guesses have a score close to 0. This is quite similar to our collision attack, which requires 60 traces to recover the same two key bytes on the same implementation.

8.4 Break of the Winning Challenge of WhibOx 2017

In this section, we will explain how we could gradually extract the key from the winning challenge (presumably the hardest) in the WhibOx 2017 white-box cryptography competition (WhibOx, 2017). This was achieved in several steps following the methodology described in Section 8.2, i.e., reverse engineering, SSA transformation, circuit minimization techniques, data dependency analysis, and algebraic analysis. To validate our attacks, we open-source some intermediate results and tools of our cryptanalysis on `Adoring Poitras` in the following git repository:

`https://github.com/junwei-wang/WhibOx-breaking-adoring-poitras`

8.4.1 The Winning Implementation

The winning implementation of (WhibOx, 2017), named `Adoring Poitras`,[5] was submitted by Biryukov and Udovenko from the University of Luxembourg. The challenge has been alive for 28 days which was twice more than the second one. As far as we know, it was broken only once by us to date.

The size of its source code is about 28 MB. Interestingly, as it includes two long strings with extended ASCII characters (ISO/IEC 8859-1:1998, 1998), it takes more than 30 hours for some compilers (e.g., Clang, which is not the reference compiler) to finish the compilation.[6] A summarized description of the original source code of `Adoring Poitras` is listed in Table 8.1. More specifically, it consists of 2328 lines of

Table 8.1: An overview of the source code of `Adoring Poitras`.

#lines	2328
#functions	1020
#global variables	12
`funcptrs` size	210
`pDeoW` size	2^{21} B
`JGNNvi` size	15 284 369 B

code, 1020 function definitions, and 12 global variables. Most of the global variables are pointers, but one global variable is an array of 210 function pointers (`funcptrs`) and two other global variables `pDeoW` and `JGNNvi` are large arrays with numerous extended ASCII characters.

We explain in this section how to gradually extract the key from `Adoring Poitras` which is protected by several advanced obfuscation techniques in a few steps. Firstly, we perform some reverse engineering on the source code to remove several obfuscation layers and obtain a Boolean circuit. Then, we transform the Boolean circuit into the *single static assignment* (SSA) form which enables us to minimize it by detecting

[5] The name was generated by the server. Source code available at `https://whibox-contest.github.io/show/candidate/777`.

[6] Experiments are done with Apple LLVM version 9.0.0 on macOS 10.12 and clang version 3.8.1 on Alpine Linux 3.5. The latter is the reference OS used by the contest server.

and removing many intermediate variables either constant, or redundant, or pseudorandom (with no impact on the final result). Based on this minimized Boolean circuit, we conduct a data dependency analysis to identify some specific encoded operations (e.g., first-round AES s-boxes). Finally, we perform a generic algebraic analysis based on a linear decoding assumption which turned out to be true. From the processed (encoded) data over several executions, we are able to extract the 16 AES key bytes. Overall, it took us roughly 200 man-hours (spread over 3 weeks) to break this challenge: about one third of the time was spent on reverse engineering; another third was for data dependency analysis and minimization of the circuit; and the remaining time was for seeking possible attacks and applying our algebraic analysis. Undoubtedly, we spent a lot of time investigating reverse engineering and attack strategies that turned out to be useless in the end. If we repeated our attack on an implementation from the same white-box compiler but for a different key and randomness, we could probably break it in a few hours (which could be dramatically improved with software tools). In the following sections, we will describe the above steps in detail.

8.4.2 Reverse Engineering

For some reason (e.g., in order to obscure the design ideas), the source code `Adoring Poitras` is deliberately obfuscated with several code obfuscation techniques, e.g., naming obfuscation, virtualization obfuscation (Rolles, 2009). We will go through how to unpack each obfuscation layer by reverse engineering. There is no obvious boundary between any two steps. Let us start with readability processing.

Readability Processing

The names of all the variables, functions and parameters in the original source code are obfuscated as shown below:

```
1  void xSnEq (uint UMNsVLp, uint KtFY, uint vzJZq) {
2    if (nIlajqq () == IFWBUN (UMNsVLp, KtFY))
3      EWwon (vzJZq);
4  }
5
6  void rNUiPyD (uint hFqeIO, uint jvXpt) {
7    xkpRp[hFqeIO] = MXRIWZQ (jvXpt);
8  }
9
10 void cQnB (uint QRFOf, uint CoCiI, uint aLPxnn) {
11   ooGoRv[(kIKfgI + QRFOf) & 97603] = ooGoRv[(kIKfgI + CoCiI) | 173937]
12     & ooGoRv[(kIKfgI + aLPxnn) | 39896];
13 }
14
15 uint dLJT (uint RouDUC, uint TSCaTl) {
```

```
16    return ooGoRv[763216 ul] | qscwtK (RouDUC + (kIKfgI << 17), TSCaTl);
17  }
```

Actually, only 210 of these functions listed in the `funcptrs` are invoked in the computation, in other words, nearly 80% of defined functions are never used. Besides, all these 210 useful functions are duplicate definitions of only 20 functions. With the help of the above observation, we perform a readability processing of the original code, including:

- renaming variables, functions and parameters,
- eliminating dummies and duplicates,
- rewriting constants in a meaningful way, and
- combining codes if necessary.

Technically, most of the processing here was handled manually. In the end, we acquire source code with 20 easily understood functions shown in the code listing below. With the help of some understanding (discussed in the following sections), these functions can be classified into several categories: input reading and output writing, bitwise operations, bit shifts, table lookups, assignments, control flow primitives, and dummy functions. We will refer to their names in the following if necessary.

```
1   uint a, b;              // a is used in table lookup
2                           // b is used for updating
3   const uint T[] = "..."; // 2^18 uint array
4
5   /* input reading and output writing */
6   void read_plaintext(uint addr, uint pos)
7     { assign(addr, plaintext[pos]); }
8   void write_ciphertext(uint pos, uint addr)
9     { ciphertext[pos] = lookup1(addr); }
10  void expand_bit(uint to, uint from, uint pos) {
11    // expand bit to unsigned long integer
12    T[(a + to) & 0x3ffff] = -((T[(a + from) & 0x3ffff] >> pos) & 1);
13  }
14
15  /* bitwise operations */
16  void not(uint to, uint from) {
17    T[(a + to) & 0x3ffff] = ~T[(a + from) & 0x3ffff];
18  }
19  void or(uint to, uint from1, uint from2) {
20    T[(a + to) & 0x3ffff] = T[(a + from1) & 0x3ffff]
21      | T[(a + from2) & 0x3ffff];
22  }
23  void xor(uint to, uint from1, uint from2) {
```

```
24    T[(a + to) & 0x3ffff] = T[(a + from1) & 0x3ffff]
25      ^ T[(a + from2) & 0x3ffff];
26  }
27  void and(uint to, uint from1, uint from2) {
28    T[(a + to) & 0x3ffff] = T[(a + from1) & 0x3ffff]
29      & T[(a + from2) & 0x3ffff];
30  }
31
32  /* bit shifts */
33  void right_shift_xor(uint to, uint from, uint pos) {
34    if (pos > 63) // always false
35      return;
36    T[to & 0x3ffff] ^= T[(a + from) & 0x3ffff] >> pos;
37  }
38  void left_shift_xor(uint to, uint pos, uint from) {
39    uint tmp = (T[(a + from) & 0x3ffff]) & 1;
40    T[(a + to) & 0x3ffff] ^= tmp << pos;
41  }
42
43  /* table lookups */
44  uint lookup1(uint addr)      { return T[(a + addr) & 0x3ffff]; }
45  uint lookup2(uint x, uint y) { return T[(x + y) & 0x3ffff]; }
46  void update_a() {
47    a = lookup2(1592, (b >> 6) + ((b & 63) << 12));
48  }
49  void update_b() {
50    b = 0x7fff & lookup2(522, (b >> 6) + ((b & 63) << 12));
51  }
52
53  /* assignments */
54  void assign_a(uint val)        { a = val; }
55  void assing_b(uint from)       { b = T[from] & 0x07fff; }
56  void assign(uint to, uint val) { T[(a + to) & 0x3ffff] = val; }
57  void copy(uint to, uint addr)  { assign(to, lookup1(addr - a)); }
58
59  /* control flow primitives */
60  void goto_func(uint pos) {       // ``goto'' in the virtual machine
61    pc = bop + pos;
62  }
63  void jump_if(uint x, uint y, uint pos) { // conditional jump
64    if (lookup2(2979, (b >> 6) + ((b & 63) << 12)) == lookup2(x, y))
65      goto_f(pos);
66  }
67
68  /* dummy function */
69  void mistery(uint to, uint from) {
70    T[(a + to) & 0x3ffff] =
71      T[(((~T[(a + from) & 0x3ffff]) & 0x7fff) >> 6) + 2979
```

```
72        + ((((~T[(a + from) & 0x3ffff]) & 0x7fff) & 63) << 12)];
73  }
```

De-Virtualization

After the readability processing, the source code is much easier to understand, and we can observe that the overall program relies on a virtual machine as illustrated in the code listing hereafter, which is a common obfuscation technique in modern software protection and malware (Rolles, 2009).

```
1   uint T[] = "...";           // 2^18 uint memory, renamed from pDeoW
2   char program[] = "...";     // 15284369 bytes, renamed from JGNNvi
3   void * funcptrs = {"..."};
4
5   void interpretor() {
6     uchar *bop = (uchar *) program;
7     uchar *eop = bop + sizeof (program) / sizeof (uchar);
8     uchar *pc = bop;
9     while (pc < eop) {
10      uchar args_num = *pc++;
11      if (args_num == 0) {
12        void (*func_ptr) ();
13        func_ptr = (void *) funcptrs[*pc++];
14        uint *arg_arr = (uint *) pc;
15        pc += args_num * 8;
16        func_ptr ();
17      } else if (args_num == 1) {
18        void (*func_ptr) (uint);
19        func_ptr = (void *) funcptrs[*pc++];
20        uint *arg_arr = (uint *) pc;
21        pc += args_num * 8;
22        func_ptr (arg_arr[0]);
23      } else if (args_num == 2) {
24        void (*func_ptr) (uint, uint);
25        func_ptr = (void *) funcptrs[*pc++];
26        uint *arg_arr = (uint *) pc;
27        pc += args_num * 8;
28        func_ptr (arg_arr[0], arg_arr[1]);
29      }
30      // similar branches for ags_num = 3, 4, 5, 6
31    }
32  }
33
```

```
34 void AES_128_encrypt(uchar * ciphertext, uchar * plaintext) {
35   interpretor();
36 }
```

Specifically, the authors of the challenge implemented a virtual environment with an interpreter of a bytecode program. The program is a sequence of instructions, each of which is either a conditional jump to a previous instruction or a function call written in the following format:[7]

```
[number of arguments][function pointer index][argument list],
```

where `[number of arguments]` is one byte indicating the number of arguments, and `[function pointer index]` is one byte giving the index of the called function within the array of function pointers (i.e., the global variable `funcptrs`), and `[argument list]` is the sequence of arguments, each taking eight bytes. In the runtime, the interpreter loads an instruction, then translates it into a function call with corresponding arguments.

In order to remove this virtualization layer, we construct a new equivalent program in the C language by simulating the interpreter. In detail, after the decoding of all the instructions, we rewrite the conditional jumps as do ... while loops, and construct function calls with their arguments from the bytecode program. We thus get a C program composed of do ... while loops and some calls to the 20 useful functions with hard-coded arguments.

Simplification of the Bitwise Program

The overall structure of the bitwise program is shown in Algorithm 8.1. The default data type is unsigned 64-bit integer (`uint`). The program contains a globally-accessible table T (renamed from `pDeoW`) of 2^{18} 64-bit words (i.e., 2^{21} bytes) initialized to some hardcoded values. At the beginning of the program, each bit b_i is expanded to a full word (by the operation $-b_i \mod 2^{64}$) which is assigned to some location $\mathsf{addr}_{i,1}$ in T. Then, each expanded bit $T[\mathsf{addr}_{i,1}]$ is copied to 63 locations

$$\mathsf{addr}_{i,1}^{(1)}, \mathsf{addr}_{i,1}^{(2)}, \cdots, \mathsf{addr}_{i,1}^{(63)}$$

in the table, where

$$\mathsf{addr}_{i,1}^{(n)} = \mathsf{addr}_{i,1} + 2^{12} \cdot n \mod 2^{18}.$$

Then the program performs a sequence of 2573 bitwise operation loops, followed by one bit combination loop (pictured in Algorithm 8.3 below), then by 9 additional bitwise operation loops. The bit combination loop is the only one to involve bit

[7] The conditional jump is is also implemented as a function in the same format (see `goto_func` and `jump_if` functions above). Particularly, it is used for simulating the do ... while loop in a high-level language, where the first two arguments are used for condition checking and the third arguments is the destination.

shifts. In comparison, the bitwise operation loops only perform bitwise operations (i.e., binary operations applied in parallel to each bit slot of 64-bit operands). In the end, the program outputs each ciphertext bit from a different location $\mathsf{addr}_{2,i}$ in table T.

Algorithm 8.1 Structure of the bitwise program

Input: plaintext bits $(b_1, b_2, \cdots, b_{128})$, unsigned long integer table T of length 2^{18} with initial values

Output: ciphertext bits $(c_1, c_2, \cdots, c_{128})$

```
1: for i = 1 to 128 do
2:     T[addr_{1,i}] ← −b_i          ▷ expand b_i to unsigned long integer
3:     for j = 1 to 63 do
4:         T[addr_{1,i} + j * 2^12  mod 2^18] ← T[addr_{1,i}]
5:     end for
6: end for

7: BitwiseOperationLoop1                          ▷ see Code 8.2
8: BitwiseOperationLoop2
9: ···
10: BitwiseOperationLoop2573

11: BitCombination                                ▷ see Code 8.3

12: BitwiseOperationLoop2574
13: ···
14: BitwiseOperationLoop2582

15: for i = 1 to 128 do
16:     c_i ← T[addr_{2,i}]
17: end for
```

Loops before BitCombination. Through basic debugging methods, we observe that the bitwise operation loops are each composed of 64 iterations performing up to 504 statements (except the very last loop which has 2051 statements). The basic structure of these loops is depicted in Algorithm 8.2 hereafter. A statement simply consists of a bitwise operation (xor, or, and, not) with one or two operands picked from different locations in the table T. The result of the bitwise operation is stored at another location in T. We denote by

$$\{\mathsf{addr}_1, \mathsf{addr}_2, \cdots, \mathsf{addr}_N\}$$

the accessing address sequence, namely, the locations read and written in table T by the statements (in chronologic order) in the first round of loop.

Algorithm 8.2 Example of a bitwise operation loop

1: **for** $i = 0$ to 63 **do**
2: $\quad j \leftarrow P(i)$ $\quad \triangleright$ P is a permutation of $\{0, 1, \cdots, 63\}$ and $P(0) = 0$
3: $\quad T[\mathsf{addr}_3 + j * 2^{12} \mod 2^{18}] \leftarrow$
$\qquad T[\mathsf{addr}_1 + j * 2^{12} \mod 2^{18}] \oplus T[\mathsf{addr}_2 + j * 2^{12} \mod 2^{18}]$
4: $\quad T[\mathsf{addr}_5 + j * 2^{12} \mod 2^{18}] \leftarrow$
$\qquad T[\mathsf{addr}_3 + j * 2^{12} \mod 2^{18}] \wedge T[\mathsf{addr}_4 + j * 2^{12} \mod 2^{18}]$
5: $\quad T[\mathsf{addr}_8 + j * 2^{12} \mod 2^{18}] \leftarrow$
$\qquad T[\mathsf{addr}_6 + j * 2^{12} \mod 2^{18}] \vee T[\mathsf{addr}_7 + j * 2^{12} \mod 2^{18}]$
6: $\quad T[\mathsf{addr}_9 + j * 2^{12} \mod 2^{18}] \leftarrow \neg T[\mathsf{addr}_8 + j * 2^{12} \mod 2^{18}]$
7: $\quad \vdots$
8: **end for**

All these addresses are computed from a global variable `a` which is updated in each loop iteration using a second global variable `b` and an update mechanism as follows:

```
1  int a, b; // global variables
2
3  assign_b(219964);
4  do{
5    update_a();
6    // bitwise operations
7    // ...
8    // ...
9    update_b();
10  } while(lookup2(2979, (b >> 6) + ((b & 63) << 12)) !=
11          lookup2(815257, 237931));
```

Let us denote by $a_0, a_1, \ldots, a_{63}$, the successive values taken by the global variable `a` in the 64 iterations, so that the i^{th} instruction $\mathsf{addr}_i = a_j + c_i$ in iteration j, where c_i is constant and $0 \leq j \leq 63$. By inspecting the sequence of a_j's, we observe that it satisfies

$$a_j = a_0 + p_j \cdot 2^{12} \mod 2^{18},$$

where $p_j \in \{0, 1, \ldots, 63\}$ for every j. Moreover, a closer inspection shows that $p_j = P(j)$ for some permutation P defined over $\{0, 1, \ldots, 63\}$. We did not try to understand whether there was some underlying mathematical principle in P (beyond the fact it is a permutation).

At this point, we aim to identify some properties of these loops that would reveal some structure in the program. One interesting observation is that for some loops,

there exist $1 \leq i, j \leq N$ and $i \neq j$ such that addr_i is a reading address, and addr_j is a writing address, and $\mathsf{addr}_i \equiv \mathsf{addr}_j \mod 2^{12}$ (that is $c_i \equiv c_j \mod 2^{12}$). This implies that some memory locations are both read and written during the loop execution. Such loops are said to be *overlapping*; the other loops are said to be *non-overlapping*. There are 1020 overlapping loops and 1562 non-overlapping loops in the program. Besides, there is no isolated (non-)overlapping loop in the program. With this observation, the program is divided into 27 parts, each of which only consists of either overlapping or non-overlapping loops. In the beginning, we thought this partition was related to the AES round operations, but we did not extract any useful information out of this observation.

Afterward, by inspecting some arbitrary overlapping loop, we can observe that its inner statements simply consist in some swaps between memory locations in the table T. These swaps are implemented through different sequences of bitwise operations, as shown in the sample code listed below. The operands indicate the address in table T. The first operand is for the result, while the remaining ones are for the inputs.

```
// swapping values in T[248329] and T[178697]
// where 248329 = 178697 mod 2^12
not(225586, 248329);
not( 99382, 178697);
not(125856,  99382);
xor( 13816, 225586,  99382);
xor( 33114,  99382, 225586);
not( 20933,  13816);
not(188758, 225586);
not(180239,  33114);
 or(261865, 180239, 133397);
 or( 94096,  20933, 133397);
xor(201945, 261865, 125856);
xor(  3792,  94096, 188758);
not(248329,   3792);
not(178697, 201945);

// swapping values in T[92413] and T[22781]
// where 92413 = 22781 mod 2^12
not( 24583,  92413);
not(146257,  22781);
xor( 67653, 146257, 133397);
xor(234702,  24583, 133397);
 or(181444,  24583, 133397);
and(172013, 234702,  24583);
 or(110852, 172013, 146257);
and(248606, 110852, 181444);
 or( 79222, 146257, 133397);
and(146881,  67653, 146257);
```

```
30   or( 86050, 146881,  24583);
31  and( 44767,  86050,  79222);
32  not( 92413,  44767);
33  not( 22781, 248606);
```

Moreover, we can further observe that two swapped addresses are always equivalent modulo 2^{12}. More noteworthy, these swaps seemed useless with respect to the functional correctness of the program. We thus obtain our first simplified program by removing all overlapping loops (except for the BITCOMBINATION discussed in the next paragraph). We believe the simplified code is functionally equivalent to the original program since their outputs always match on many randomly chosen inputs. Furthermore, since the remaining loops are non-overlapping (i.e., all the written memory locations are not used during the execution of the current loop), the permutation P can be replaced with the identity function (i.e., $P(j) = j, 0 \leq j \leq 63$). Or even better, we can rewrite the do ... while loop as a for loop from 0 to 63. We again verify our conjecture by comparing the program outputs before and after modification for a large number of encryptions (of random plaintexts). Now we acquire a new simpler version in which the permutations before BITCOMBINATION are all removed.

Algorithm 8.3 BITCOMBINATION (reconstructed for comprehension)

procedure BITCOMBINATION
  for $\ell = 1$ to 129 **do**
    $T[\mathsf{addr}_{3,\ell}] \leftarrow v_\ell$ ▷ $v_\ell \in \mathbb{F}_2$ is a hardcoded constant
    for $j = 1$ to 64 **do**
      $T[\mathsf{addr}_{3,\ell}] \leftarrow T[\mathsf{addr}_{3,\ell}] \oplus \text{PARITY}(T[\mathsf{addr}_{4,\ell} + j * 2^{12} \mod 2^{18}])$
      $T[\mathsf{addr}_{3,\ell}] \leftarrow T[\mathsf{addr}_{3,\ell}] \oplus \text{PARITY}(T[\mathsf{addr}_{5,\ell} + j * 2^{12} \mod 2^{18}])$
    end for
  end for
end procedure

procedure PARITY(x) ▷ the number of 1-bits in x modulo 2
  $r \leftarrow 0$
  for $i = 0$ to 63 **do**
    $r \leftarrow r \oplus (x \gg i)\&1$
  end forreturn r
end procedure

BITCOMBINATION and the Remaining Loops. Algorithm 8.3 illustrates how BITCOMBINATION works. It first assigns 129 locations $(\mathsf{addr}_{3,\ell})_{1\leq\ell\leq129}$ in T with Boolean constants (namely either `0x00...00` or `0x00...01`). Then each of these table locations is further xor-ed with the parity bits (each of which is computed through 64

simple instructions) of 128 different values stored in $\mathsf{addr}_{4,\ell} + j * 2^{12} \mod 2^{18}$ and $\mathsf{addr}_{5,\ell} + j * 2^{12} \mod 2^{18}$, for some addresses $\mathsf{addr}_{4,\ell}$ and $\mathsf{addr}_{5,\ell}$ and for $1 \leq j \leq 64$. The 129 64-bit words output from BITCOMBINATION are hence Boolean variables. Moreover, after the remaining loops, all the ciphertext bits are the least significant bits of some specific 64-bit words in T. Therefore, we deduce that only the least significant bits of the remaining computations after BITCOMBINATION influence the outputs, i.e., everything happening after BITCOMBINATION can be seen as a Boolean circuit.

Besides, we observe that only a single iteration in the last bitwise operation loop affects the output ciphertext, which means that we can replace this loop by a single iteration (for a given value of the loop index i). Then we can reiterate this observation with the loop before, and so on until the BITCOMBINATION loop. In the end, the operations after BITCOMBINATION is simplified as a Boolean circuit made of one iteration of each former loop.

Entire Transformation to a Boolean Circuit

Similar observations and conjectures can be applied to the loops before BITCOMBINATION. Specifically, observing that all the operations are bitwise and that any two bits in different positions of the operands never communicate with each other until BITCOMBINATION, we conjecture that

(1) the i^{th} bit of the intermediate values in the j^{th} loop iteration corresponds to one independent *partial* AES computation (i.e., not complete without the operations after BITCOMBINATION),

(2) only one (or odd number of) such independent computation(s) in 64*64 of them is (are) real.

To verify this conjecture, we tried to execute BITCOMBINATION while skipping one bit index $1 \leq i \leq 64$ in the parity computation for one loop index $1 \leq j \leq 64$. For three pairs (i, j), we observed the 129 outputs of BITCOMBINATION were constant to 0 over several plaintexts. We deduced that real AES computations are performed in the i^{th} bit slot of the j^{th} iteration for $(i, j) \in \{(42, 26), (58, 32), (10, 48)\}$ before BITCOMBINATION. Therefore, we can simplify the code by picking any single separate AES computation and verify our guess in the usual way. Accordingly, the bitwise program is fully transformed into a Boolean circuit.

8.4.3 SSA Transformation and Circuit Minimization

Although we get a Boolean circuit, we still lack knowledge about how it works, e.g., where each round is computed. As in a typical unpacking story, we perform some static and dynamic analyses to acquire more information. In the current representation, many intermediate variables are both written and read several times, which

presumably hides some facts on the data flow. We hence rewrite the Boolean circuit into *single static assignment* (SSA) form according to Section 8.2.

After SSA transformation, we attempt to reduce the Boolean circuit by applying circuit minimization techniques described in Section 8.2. Specifically, we can detect removable intermediate variables, including constants, duplicates, pseudorandomness, and dummy variables, by executing the implementation with a large number of randomly sampled plaintexts (along with flipping variables for detecting pseudorandomness). Then we can replace the pseudorandom variables by 0's, remove duplicates and dummy variables and propagate the constants according to the different operations. We apply each step between 2 and 5 times except for the removal of dummy variables that is applied a dozen times. We obtain a minimized circuit of 280 thousand gates (Boolean instructions), which is about half of the original size.

8.4.4 Data Dependency Analysis

A visual way to analyze data dependency of a circuit is to plot its *data dependency graph* (DDG), a directed acyclic graph (DAG) in which a vertex stands for an intermediate variable (an address in T in our case) and a directed edge means a variable (ending vertex) is computed from another variable (starting vertex). We extract and plot data dependency graph of our minimized circuit using Mathematica.[8] Specifically, for each statement in the minimized circuit, we first generate one/two directed edges from the addresses of its operands to the address of its destination; then we get an ordered sequence of edges according to the order in which the relevant gates appear in the circuit. Then we invoke the `Graph` function of Mathematica with the sequence of edges to plot the DDG. At first, we attempt to plot a figure for the whole DDG, but fail since it is too costly to produce such a large graph for Mathematica with a standard computer. Then we try to plot some smaller part of the circuit DDG, starting with the first 20% which looks like a mess as shown in the left of Figure 8.4. Afterward, we try plotting the first 10% of the DDG as shown in the right of Figure 8.5, but we can only extract a limited amount of valuable information with the exception of some kind of symmetry as illustrated by the red line on the figure. We keep going and plot the 5% of the DDG as represented in Figure 8.6 which reveals much more structure than our previous observations. A mysterious "ball" is located in the center of the graph, which is mainly composed of the first edges (i.e., the beginning of the circuit), and 16 "branches" come out from this central ball, divided into four groups for which the four branches eventually join. The plotted circuit starts from the center and ends with flake structures. Seemingly, the beginning of the circuit has a highly complex data dependency and the variables inside are deeply mixed together and then extensively used in future computation since our minimization process cannot get rid of them.

[8] See `https://www.wolfram.com/mathematica/`.

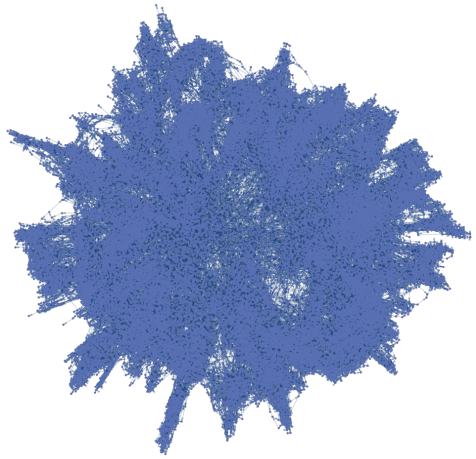

Figure 8.4: The data dependency graph for the first 20% edges plotted by Mathematica.

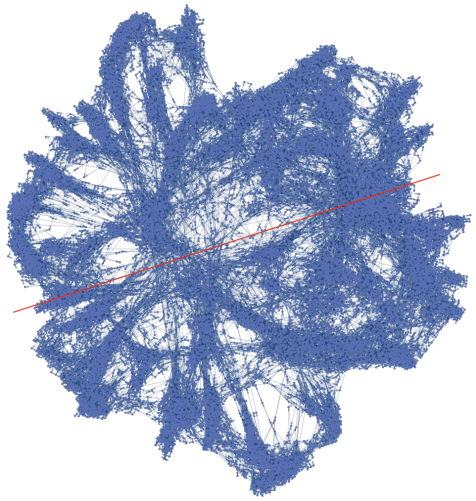

Figure 8.5: The data dependency graph for the first 10% edges plotted by Mathematica.

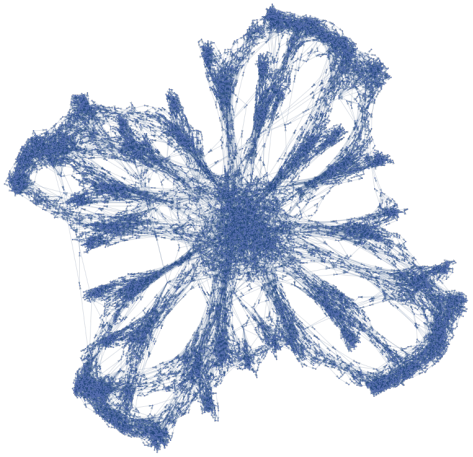

Figure 8.6: The data dependency graph for the first 5% edges plotted by Mathematica.

Extracting S-Box Encodings

Based on our knowledge of the AES structure, we make the heuristic assumption that the "branches" correspond to the 16 s-box computations in the first round of AES which are then mixed four by four through the MixColumns operations.

If our assumption is correct, the set of outgoing variables of a branch (i.e., the set of variables computed inside the branch and which are used later in the program) must be an encoding of the output s-box value. In order to extract the set of outgoing variables, we apply modularity-based clustering algorithms (Newman, 2004) to the data dependency graph. Specifically, we apply the Mathematica function `FindGraphCommunities` to the first 5% of the DDG. The graph is then divided into several communities (clusters) in a way that the vertices in the same community have a denser connection than a set of vertices from different communities. This way, we can isolate each "branch" in Figure 8.6 and obtain the corresponding set of vertices from which we extract the set of outgoing variables. Note that in practice, the clustering algorithm was not necessarily applied the first 5% of the DDG but a tuning over the search window was manually applied (see details in Table 8.2). The number of vertices in the recovered clusters is between 439 and 615 per cluster, and the number of outgoing variables scales from 29 to 57.

At this step, we have 16 sets of variables which are presumably 16 encodings of the first round s-box outputs. We now explain how we could break these encodings and recovered the corresponding secret key bytes.

8.4.5 Algebraic Analysis

Let us denote by $v_1, v_2, \ldots, v_t$, the t outgoing (binary) variables of an s-box cluster, that presumably encode an s-box output. Let us denote by x the plaintext byte and by k^* the secret key byte corresponding to this s-box computation. Then, if our data dependency analysis is correct (namely if the v_i's indeed encode the s-box output), there exists a deterministic decoding function $\mathsf{dec} : \{0,1\}^t \to \{0,1\}^8$ satisfying:

$$\mathsf{dec} : (v_1, v_2, \ldots, v_t) \mapsto \big(S_1(x \oplus k^*), \ldots, S_8(x \oplus k^*)\big)$$

where $S_j(\cdot)$ denotes the j^{th} Boolean coordinate function of the AES s-box.

Our algebraic analysis works by assuming that dec is linear (actually affine) over $\mathbb{F}_2$. As we show hereafter, this is enough to break `Adoring Poitras` but it can be generalized to higher degree decoding functions (see Section 5.4). This linear decoding assumption specifically states that for each output coordinate $j \in \{1, 2, \ldots, 8\}$, there exists a constant vector $\mathbf{a} = (a_0, a_1, a_2, \cdots, a_t) \in \mathbb{F}_2^{t+1}$ such that

$$a_0 \oplus \bigoplus_{i=1}^{t} a_i \cdot v_i = S_j(x \oplus k^*)\,. \tag{8.1}$$

(Note that the coefficients a_i are different for each output coordinate but we avoid an additional index for the sake of clarity.) In other words, the j^{th} output bit of the s-box is encoded by a simple Boolean sharing and its shares are distributed among the v_i variables according to the a_i coefficients: if $a_i = 1$ then v_i is a share of $S_j(x \oplus k^*)$ and if $a_i = 0$ then $S_j(x \oplus k^*)$ is independent of v_i .

To validate our assumption, we hereafter apply our LDA attack described in Section 5.2 to extract the subkey utilized in each s-box cluster. Specifically, we first collect a set of N *computation traces* for the presumed s-box encoding, namely, we execute the white-box implementation N times with random plaintexts and record the values $(v_1^{(i)}, v_2^{(i)}, \ldots, v_t^{(i)})$, $1 \leq i \leq N$, taken by the encoding variables for these N executions. Then we iterate over the 256 possible key guesses k for the 16 possible s-box positions and try to solve the following system of linear equations (with a_0, a_1, $\ldots$, a_t as unknowns):

$$\begin{bmatrix} 1 & v_1^{(1)} & v_2^{(1)} & \cdots & v_t^{(1)} \\ 1 & v_1^{(2)} & v_2^{(2)} & \cdots & v_t^{(2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & v_1^{(N)} & v_2^{(N)} & \cdots & v_t^{(N)} \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_t \end{bmatrix} = \begin{bmatrix} S_j(x^{(1)} \oplus k) \\ S_j(x^{(2)} \oplus k) \\ \vdots \\ S_j(x^{(N)} \oplus k) \end{bmatrix},$$

where $x^{(i)}$ denote the values taken by the plaintext byte x in the i^{th} execution. If our linear decoding assumption is true, then the above system is solvable for the right s-box position and the right key guess $k = k^*$, which directly follows from Equation 8.1, and the solution reveals the decoding function dec. On the other hand, for an incorrect key guess, the chance to solve the system quickly becomes negligible as the number N of traces increases above t, which has been formally discussed in Section 5.3.

Note that the selection of the outgoing variables v_1, v_2, $\ldots$, v_t (which are basically the fringe edges of a cluster) is crucial for this attack to work. When a single one happens to be missing then the system becomes unsolvable. This stresses the importance of a sound clustering step for the subsequent success of this attack.

Practical Results

We perform the above algebraic analysis based on our linear decoding assumption to extract the key from our minimized Boolean circuit. For each presumed s-box cluster, we extract the outgoing variables and record a set of computation traces. Thanks to the data dependency analysis (and the clustering step) described above, the number t of outgoing variables is never more than a few dozens (specifically at most 59). Moreover, we use up to $N = 100$ computation traces, which overall yields some linear systems of dimensions lower than 80×100 solvable within a few microseconds on a desktop computer.

For each cluster, we try to solve the linear systems obtained for all the pairs (k, j) (key guess and s-box coordinate) and all the 16 s-box positions. For most clusters, all the 8 systems obtained for a single s-box position and a single key guess are solvable whereas the others are unsolvable, giving a strong presumption that we have found the correct key byte. For one cluster, less than 8 systems are solvable, but still for a single s-box position and a single key byte. And for a few other clusters, no system is solvable at all. The two latter cases occur as a consequence of a wrong cluster selection. In these cases, we have to fine-tune the clustering step by varying the range of the input edges to eventually get some solvable systems (each time for a single key guess). After recovering 14 out of 16 key bytes, we exhaust the remaining ones (the 6^{th} and 12^{th}) by brute-force search (over a plaintext-ciphertext pair computed with the white-box implementation)[9] and finally recover the full AES key.

Table 8.2 depicts our practical results in detail. For each of the 16 s-boxes (but the 6^{th} and the 12^{th} for which we use exhaustive search) it gives the range of edges in the DDG used for clustering, the number of vertices (or variables) in the extracted cluster, the corresponding number of outgoing variables (parameter t), the number of Boolean shares in the encoding of each s-box output bit (i.e., the Hamming weight of the coefficient vector $\boldsymbol{a}$), and the recovered key byte. Note that for the 8^{th} s-box we cannot solve the 8 systems corresponding to the right key guess but only 3 of them (which explains '?' for the number of shares).

Table 8.2: Clustering and algebraic analysis results

s-box	**edge range**	**#cluster**	**#outgoing vars** (t)	**#shares** (HW($\boldsymbol{a}$))	**key byte**
1	9,500 - 18,000	541	30	{8,6,7,5,8,3,3,7}	0x71
2	4,000 - 18,000	543	29	{7,7,6,9,8,7,7,8}	0x3c
3	4,000 - 13,500	530	34	{8,10,8,8,6,2,6,4}	0xcf
4	9,500 - 18,000	515	38	{6,9,8,6,6,11,9,9}	0x9f
5	9,500 - 20,000	571	41	{8,6,6,4,4,9,8,10}	0x27
6	-	-	-	-	0x45
7	9,500 - 20,000	615	42	{2,11,5,6,7,10,3,8}	0xe5
8	9,500 - 24,000	500	59	{?,10,11,?,?,14,?,?}	0xbc
9	9,500 - 20,000	448	57	{4,6,7,8,7,6,6,12}	0x04
10	9,500 - 18,000	568	36	{8,6,6,6,6,12,6,8}	0x64
11	4,000 - 18,000	523	35	{9,5,7,9,3,3,8,7}	0xb9
12	-	-	-	-	0x07
13	9,500 - 18,000	514	30	{8,5,4,7,5,5,5,6}	0x78
14	9,500 - 20,000	454	45	{14,9,13,12,14,15,10,16}	0xf4
15	9,500 - 18,000	505	30	{8,6,6,8,8,7,4,8}	0x77
16	9,500 - 18,000	439	49	{6,8,8,8,4,6,10,6}	0x07

For instance, for the third s-box, we can extract a cluster with 530 variables in the edges ranging between 4000 and 13500 and among which 34 are outgoing variables. For this cluster, we can solve the 8 linear systems. For further illustration, Table 8.3 exhibits the solutions of these 8 systems, where the encoding coefficients are ordered chronologically. We observe that only 15 consecutive variables of the

[9] We could probably extract these bytes through the algebraic analysis as well, but it was faster to search exhaustively.

34 outgoing variables are used as Boolean shares to encode the 8 output bits of the s-box. Moreover, some of these variables are involved as shares for more than one output bit of the s-box. In other words, the decoding function is a 15-bit to 8-bit linear mapping.

Table 8.3: The solution of the system of equations for each bit in the 3^{rd} byte.

Bit	Encoding coefficients
1	0 0 0 0 0 0 1 0 1 0 1 1 1 0 0 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
2	0 0 0 0 0 0 1 0 0 1 1 0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
3	0 0 0 0 0 0 0 0 1 0 1 0 0 0 1 1 1 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
4	0 0 0 0 0 0 0 0 0 1 1 0 0 0 1 1 1 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
5	0 0 0 0 0 0 0 1 1 0 0 1 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
6	0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
7	0 0 0 0 0 0 1 0 0 0 1 0 0 1 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
8	0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

8.4.6 Attack Summary

Our study reveals that `Adoring Poitras` mixes several obfuscation layers to resist the known gray-box attacks against white-box implementations, and in particular, DCA and DFA.

The innermost obfuscation layer consists of an AES circuit in which the bits mix with pseudorandomness through binary linear applications. Higher-order DCA (see Chapter 6) exploiting multiple samples in a computation trace could defeat such an encoding scheme, but it would not be efficient on `Adoring Poitras` since the number of shares is relatively high for most of the s-box output bits and the attack complexity grows exponentially in this number. This is why all our DCA attempts failed to break `Adoring Poitras`. Besides, our efforts on DFA against `Adoring Poitras` were ineffective due to some sound fault resistance mechanism (which was not fully reverse engineered).

The middle obfuscation layer involves 4096 AES instances in parallel, three of which are identical and using the real key whereas the others are based on dummy keys. On one hand, the co-existing instances can be used as redundancy computation for error-detection; and on the other hand, the number of bit samples in the computation greatly increases, resulting in a significant slow-down for collecting and analyzing the (binary) computation traces.

The outer obfuscation layer comprises many different code obfuscation techniques to harden any in-depth understanding of the implementation.

The composition of different encoding and obfuscation techniques at different levels can make it a very hard job to break a white-box implementation. In particular, one should try to protect the implementation against known generic attacks in order to force the adversary to look for new attack paths. In this regard, the designers

of `Adoring Poitras` did a good job and we would particularly retain the idea to introduce many independent instances in parallel to increase the level of hiding and serve as an error-detection mechanism at the same time.

8.5 Break of the Winning Challenge of WhibOx 2019

This section reports practical attack experiments from white-box implementations submitted to the WhibOx 2019 competition (WhibOx, 2019). Specifically, we exhibit successful key recovery attacks against the three wining challenges due to Biryukov and Udovenko. We first describe the three implementations and explain how we partially de-obfuscated them. Then we demonstrate that our novel data-dependency higher-order DCA presented in Chapter 7 could break the three winning implementations. To reproduce our attacks, we open-source some crucial components in our attacks in the following git repository

`https://github.com/CryptoExperts/breaking-winning-challenges-of-whibox2019` .

As explained in this chapter, all three winning implementations were based on state-of-the-art white-box countermeasures including a mix of linear and non-linear masking (Biryukov and Udovenko, 2018) together with shuffling and additional obfuscation. In this chapter, we give a thorough explanation of how we managed to break all three implementations with advanced gray-box attacks (described in Section 7.3) and our novel data-dependency analytic techniques (described in Section 7.4), as well as a "lightweight" de-obfuscation. To the best of our knowledge, this is the only technical report that breaks all the three implementations.

8.5.1 The Three Winning Implementations

The three winning white-box implementations of the WhibOx 2019 competition are `#100` (`hopeful_kirch`), `#111` (`elegant_turing`), and `#115` (`goofy_lichterman`). As we will see, these three implementations are protected with a combination of linear and non-linear masking together with additional obfuscation and shuffling (for two of them). We summarize the performance achieved by the three implementations according to our (desktop computer) measurements in Table 8.4.

8.5.2 De-Obfuscation and Implementation Structures

We explain hereafter the reverse engineering effort we had to take in order to obtain implementations that we could easily target with gray box attacks.

Formatting

The three implementations are one-liner programs with an additional optimization directive for `GCC` in the comment (first line of the source file). We first write a script to turn the one-liner program into a several-line program by inserting a line break

Table 8.4: Performance of the winning implementations measured under iMac (27-inch, late 2012) with a 3.4 GHz Intel Core i7 processor and macOS Mojave Version 10.14.6

	#100	#111	#115
Source Code Size (MB)	17.42	49.10	48.42
Binary Size (MB)	16.29	8.34	11.08
RAM (MB)	16.28	9.16	11.90
Execution Time (s)	0.352	0.052	0.068
Performance Score (by us)	3.03	7.65	6.49
Performance Score (by WhibOx)	3.07	7.73	6.50

in front of each `void` and `unsigned char`, or after each semicolon (`;`) and brace (`{}`). This has to be done carefully since C language keywords are used in string variables and inserting a line break for these cases would break the integrity of the program. We believe this is used as an anti-de-obfuscation countermeasure. Then we re-indent the code for further readability. The total number of lines in `#100`, `#111`, and `#115` are about 21 thousand, 19 thousand, and 20 thousand respectively.

At this step, one can observe that `#111` and `#115` are two very similar implementations while `#100` is slightly different from them. As a matter of fact, `#111` and `#115` were submitted the same day (right before the deadline) with a two-hour gap while `#100` was submitted three days earlier.

Renaming Symbols, Removing Dummies and Duplicates

After formatting the source code, we could observe that all symbols (including variable names, function names, parameter names) were in the form of a random combination of three words connected by underscore characters. For illustration, hereafter are a few lines of the source code of `#100`:

```
1  unsigned int *suricata_stanford_hypertext;
2  unsigned int orthography_automation_parallel;
3  void associativity_differential_discrete(void) {
4    nineteen_algorithms_gardening = 0ULL;
5  }
6  void AES_128_encrypt(char *polyalphabetic_set_cyberdating,
7                       char *netzine_vocab_foundationer ) {
8    inscription_shape_writing(
9      (unsigned char *)netzine_vocab_foundationer,
10     (unsigned char *)polyalphabetic_set_cyberdating);
11   return;
12 }
```

At this point, based on our understanding of the code, we rename all symbols in a meaningful way. At the same time, we remove all dummy operations (which are never used) and we merge duplicated functionalities. As shown in Table 8.4,

both source codes of `#111` and `#115` are close to 50 MB, but their binaries are much smaller. We notice that there exist many long strings representing valid C source code in which the symbols have the same three-word format which makes it hard to distinguish between the inside and the outside of strings. Nevertheless, as we could deduce, these strings are useless and removed at compilation, so that one can safely remove them.

Virtual Machine

At this point, we have a human-readable source code and we can observe that it includes a wide array. This array is used both as a read-only bytecode interpreted by a virtual machine and as the program memory to write and read intermediate variables. The memory location in the array is dynamic and depends on the plaintext and the usage. However, we could realize the dependency of the memory location in the array on the plaintext and the usage is breakable. Hence, we isolate the memory for each usage (i.e., each piece of bytecode) from the long array.

The virtual machine of `#100` is pretty simple as shown below. Three types of instructions are implemented: group of bitwise `XOR`s, group of bitwise `AND`s (whose inputs are possibly flipped), and rewinding the memory. The operand width is fixed as 64 bits. The memory is used sequentially and it is rewound until it is fully used by the third instruction. Hence, we can easily transform the bytecode into its *single static assignment* (SSA) using a large memory.

```
1  void vm_for_100(u64 * memory, int steps) {
2    u64 * ptr = memory + 136;
3
4    while (step--) {
5      if (read_a_bit() == 0) {
6        // execute a certain number of bitwise XORs
7        u32 offset1 = read_18_bits();
8
9        int counter = 1;
10       while (read_a_bit() == 0) {
11         counter++;
12       }
13
14       while (counter--) {
15         u32 offset2 = read_18_bits();
16         *ptr++ = memory[offset1] ^ memory[offset2];
17         pos++;
18       }
19     } else {
20       u8 b2 = read_a_bit();
21       if (b2 == 0) {
22         /** execute a number of bitwise  (x & y) or (x & ~y) or
```

```
23                                          (~x & y) or (~x & ~y)  **/
24          u32 offset1 = read_18_bits();
25          u64 mask1 = read_a_bit() ? 0xffffffffffffffff : 0;
26
27          counter = 1;
28          while (read_a_bit() == 0) {
29            counter++;
30          }
31          while (counter--) {
32            u32 offset2 = read_18_bits();
33            u64 mask2  = read_a_bit() ? 0xffffffffffffffff : 0;
34            *ptr++ = (memory[offset1] ^ mask1) &
35                     (memory[offset2] ^ mask2);
36            pos++;
37          }
38        } else {
39          // rewind the memory
40          ptr = memory + read_18_bits();
41        }
42      }
43    }
44  }
```

The virtual machines in `#111` and `#115` are similar to that in `#100`, except that the operands can be either 16-bit or 32-bit depending on the bytecode. Besides, `#100` only has one bytecode whereas `#111` and `#115` have 4 different bytecodes. In the following, we will discuss these bytecodes separately.

Structure of `#100`

The bytecode in `#100` is sequentially interpreted 4 times on the same plaintext and different constants inputs. At the end of the program, the outputs of the 4 interpretations are merged. Obviously, we have four instances of a 64-bit bitslice program which makes 256 independent instances of the same Boolean circuit. This circuit takes as input a variable part obtained by applying a Boolean circuit to the plaintext at the beginning of the program and a constant part (hardcoded in the implementation). The variable part is the same for the 256 instances while the constant part is different for each instance. The output of each instance is hence of the form $f(p,i)$ for some function f where p is the input plaintext and $i \in \{0,\ldots,255\}$ represents the constant index. All the outputs are then XOR-ed together and input to a final Boolean circuit which produces

$$\mathrm{AES}_k(p) = h\left(\bigoplus_{i=0}^{255} f(p,i)\right),$$

where h denotes the function computed by the final Boolean circuit. Our intuition is that, in the i^{th} slot, the f-circuit computes $\mathrm{AES}_{k_i}(p)$ for some key k_i, as well as some

further function $\mu(p, i)$. Then for some plaintext-dependent index $i = g(p)$, the key k_i matches the right key and a selection process (at the end of the circuit) ensures

$$f(p,i) = \begin{cases} \mathrm{AES}_k(p) + \mu(p,i) & \textbf{if } i = g(p), \\ \mu(p,i) & \textbf{otherwise}. \end{cases}$$

Then XOR-ing everything, one gets something like

$$\bigoplus_{i=0}^{255} f(p,i) = \mathrm{AES}_k(p) \oplus \underbrace{\bigoplus_{i=0}^{255} \mu(p,i)}_{\mu'(p)}$$

and the h function merely removes $\mu'(p)$.

We also assume that some error detection mechanism is implemented in the f-circuit. A detectable fault injection could trigger the program to choose a wrong slot (i.e., not indexed by $g(p)$) as the final result or merge the result from many wrong slots.

A difference between `#100` and `Adoring Poitras` in Section 8.4 is that the slot chosen for correct execution in the former is fixed for any inputs while the good slot in the latter is determined pseudorandomly by the input.

Structure of `#111` and `#115`

The sketches of `#111` and `#115` are described in Algorithm 8.4 below, in which each BYTECODEX is an interpretation of a bytecode giving rise to a bitslice program with 16 or 32 slots. Specifically, BYTECODEBEGIN is only used at the beginning of the program and BYTECODEEND is only used at the end of the program, while BYTECODEMIDDLEA followed by BYTECODEMIDDLEB are sequentially used 9 times in the middle of the program afterward BYTECODEMIDDLEA is repeated twice. Only BYTECODEMIDDLEB uses 32 slots while the others use 16 slots.

Algorithm 8.4 AES(`pt`)

Input: : plaintext `pt` and constants `cst[11]`

Output: : ciphertext `ct`

1: `state` ← BYTECODEBEGIN(`pt`)
2: **for** $i \in \{1, \cdots, 9\}$ **do**
3: `state` ← BYTECODEMIDDLEA(`state`, `cst[i-1]`)
4: `state` ← BYTECODEMIDDLEB(`state`)
5: **end for**
6: `state` ← BYTECODEMIDDLEA(`state`, `cst[9]`)
7: `state` ← BYTECODEMIDDLEA(`state`, `cst[10]`)
8: `ct` ← BYTECODEEND(`state`)

Based on these observations, we assume that BYTECODEMIDDLEA performs the round key addition and the 16 s-boxes (in parallel) while BYTECODEMIDDLEB performs the linear layer (i.e., ShiftRows and MixColumns). Since the intermediate values transmitted and rearranged between two bytecode interpretations and no value merged from different slots, we also guess that each slot in BYTECODEMIDDLEA corresponds to one s-box computation so that there is no horizontal shuffling implemented.

8.5.3 Attacking `#100`

As explained in Section 8.5.2, `#100` supposedly implements a bitsliced circuit where the correct execution is carried by a single bit slot which is pseudorandomly shuffled (based on the input plaintext) among the 256 bit slots. In other words, `#100` is protected with a horizontal shuffling of degree 256. One could try to directly apply the data-dependency HO-DCA on binary samples but, according to the analysis of Section 7.3.2, the shuffling would imply a reduction of the target correlation score by a factor $\frac{1}{256}$ and hence an increase of the number of traces by a factor 2^{16}.

Locating the Slot for Correct Execution. In order to avoid paying this price, we tried to locate the good slot for each execution. To do so, we tried to locate a gate in the Boolean circuit for which flipping only one of the 256 bit slots in the output would affect the final AES ciphertext. After a few trials, we could locate such a gate, which allowed us to record a set of plaintexts for which we knew the good slot i.e., the value of $g(p)$. For this set of plaintexts, we could hence record single-slot traces and hence completely defeat the horizontal shuffling countermeasure. As a side effect of this shuffling removal, we also discarded the correlation scores corresponding to dummy keys k_i with $i \neq g(p)$.

Trace Recording. Since we have access to the (formatted) source code, we can easily record computation traces. The full computation trace is momentarily stored in RAM and directly used to derive a data-dependency computation trace (using Algorithm 7.2). This requires a preliminary detection of multipliers (through Algorithm 7.1) but this can be done a single time per implementation.

Correct Slot Attack. The results of the correct slot data-dependency HO-DCA are illustrated in Figure 8.7. The target variable is the 1^{st} bit of the 3^{rd} s-box in the initial round. The correlation trace for the good key candidate (plotted in blue) is clearly distinguishable from other candidates (in gray). This attack used 767 traces limited to the first 18% of the circuit (as we assumed the first round should occur in that range). Using this attack, we could recover 7 of the 16 key bytes.

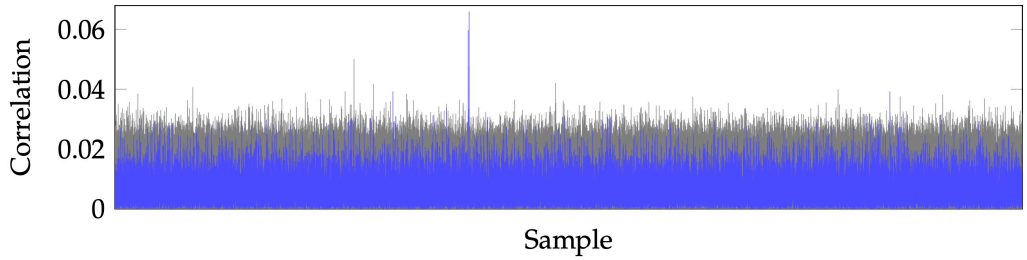

Figure 8.7: Correlation score when targeting the 1st bit in the 3rd s-box when fixing attacking correct slot, in which the correlation trace for the good key candidate `0xb3` is plotted in blue, and the wrong key candidates are plotted in gray.

We further applied our data-dependency attack for subsets of the set of multipliers. Specifically, for some target subset cardinality n (presumably corresponding to the linear masking order), we derive a sample (by XOR-ing all the elements) for each n-cardinality subset of each set of multipliers. Using this attack we could recover 8 more key bytes. The last key byte could then be recovered by exhaustive search. Table 8.5 hereafter summarizes the key byte that could be recovered by our data-dependency attack with respect to the target bit and the cardinality of the multiplier (sub)set.

Table 8.5: Which bit is vulnerable to each of 16 bytes in either a **correct slot attack** (by 767 plaintexts) or an **integrated** attack (by 15 thousand plaintexts) with a **full** set of multipliers or a subset of multipliers with cardinality **2** or **3** or **4**. The underlined bit means the good key guess ranked first in the correlation score but the advantage is not significantly high. The blank cell means no bit was vulnerable in the corresponding attack.

	correct slot attack				**integrated**
	full	**2**	**3**	**4**	**full**
1		7,8	7	1	8
2		6	6	5	6,8
3	1	1,5	1,4	1	1,4,5,6
4	5,8	1,4,5,7,8	4,5,8	4,5,7	4,5,8
5		1	1		3,5, 6
6	7	1,7,8	1,7		1,2,3,7
7		5	5		
8	4,5	4,5	4,5	4,5	4,5,8
9		3,7,8			3,5,7,8
10	1	1,6,7,8	1,7	1,7	1,2,6
11	6	6,7	7	7	6
12		4,5	4,5		8
13		4,5,6	4,5		4,5
14	1	1,2	1		1,2,6
15		1,6,8	6		8
16		5,6			7

Integrated Attack. Although for our break, we managed to remove shuffling by detecting the good slot, we could alternately have used the integration attack. According to the analysis of Section 7.3.2, using integration against shuffling degree 256, we expect to increase the number of traces by a factor 256 (instead of 2^{16} without integration). We validated that we could break `#100` with data-dependency integrated HO-DCA using 15,000 traces. For this attack, the target traces are generated in two steps:

1. derive data-dependency traces (using Algorithm 7.2) made of 256-bit samples which are computed by XOR-ing the set of multipliers for each gate,
2. derive integrated traces whose samples are the Hamming weights of the original samples.

The attack results are depicted in Figure 8.8 for the 1st bit of the 3rd s-box of the initial round. We observe that we can clearly distinguish the good key guess.

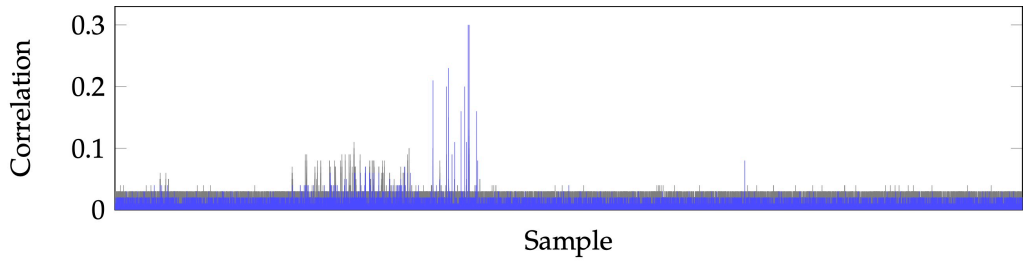

Figure 8.8: Correlation score when targeting the 1st bit in the 3rd s-box with data-dependency integrated HO-DCA. The blue line is for the correct key byte `0xb3` and the gray lines are for the incorrect key guesses.

8.5.4 Attack `#111` and `#115`

As explained above, implementations `#111` and `#115` are very similar and we could break them using the exact same attack path. In the following, we hence only present our attack results on `#115`.

Recall our hypothesis is that BYTECODEMIDDLEA implements the s-box and each bit slot corresponds to one s-box computation within one round. We target the first invocation in order to recover the first round key. However, while applying our (integrated) data-dependency attack in this context, we observe a lot of correlation peaks corresponding to many key candidates, implying that the target computation somehow includes dummy keys (probably through vertical shuffling).

In order to bypass dummy keys, a possibility is to target deeper rounds but this implies dealing with an increased computation complexity since the key space is substantially larger. We rather suggest attacking the s-box inputs in the last round which each depends on a key byte of the last round key. The target variable is hence a function of the right ciphertext and it is unlikely that dummy keys appear in this

context since that would mean that the implementation first computes the right ciphertext and then goes somehow backward to make appear e.g., s-box inverse with the right ciphertext and dummy last round key. Using our data-dependency integrated HO-DCA on the last round, we could recover the full (last round) keys of `#111` and `#115`. Figure 8.9 gives an illustration of the obtained correlation traces. We can see that the good candidate is clearly distinguishable.

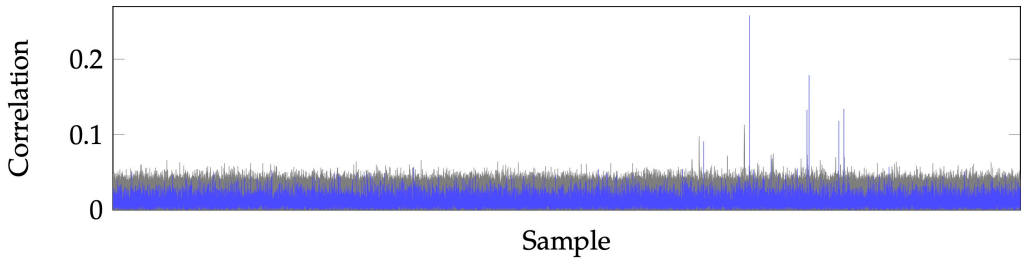

Figure 8.9: Correlation score when targeting at the 2^{nd} bit in the 1^{st} s-box in the last round with data-dependency integrated HO-DCA using 5 thousand traces. The blue curve is for the correct key guess `0x51` and the gray curves are for the incorrect key guesses.

Chapter 9

Conclusion

The existence of white-box cryptography, more generally code obfuscation is still an open problem. The industry is compelled to deploy secretly designed solutions to meet the continuously growing demand. The deployed solutions are potentially threatened by automatic passive gray-box attacks, implying that they are insecure even in a weaker setting than the one they were designed for. It is therefore important to develop implementations that are resistant to gray-box attacks, hence constraining the potential adversaries to invest costly and uncertain reverse engineering efforts. This thesis thus concentrates on the practical security of white-box cryptography, that is the analysis and improvement of gray-box attacks and the associated countermeasures in the white-box threat model.

The contribution of this thesis is summarized below.

- First of all, it formalizes the *passive gray-box adversary model* in the white-box setting and reformulates the DCA attack in this model. It also reviews the linear masking, the non-linear masking and shuffling countermeasures, as well as the source of randomness essential to implement these countermeasures. The necessity to perform shuffling both in time and memory is particularly highlighted and the possible approach to these two shuffling dimensions in bitsliced circuit is presented.

- Then, it provides an in-depth analysis of when and why DCA is capable of breaking historical internal encoded white-box designs and pinpoints the properties of the target variables and the encodings that make the attack (in)feasible. Additionally, new DCA-like attacks inspired by side-channel analysis techniques are proposed in this thesis. Specifically, a *collision attack* which particularly effective against internal encodings in terms of trace complexity and *mutual information analysis* (MIA) which naturally applicable in this context are investigated against internal encodings.

- Next, it formally describes a *linear decoding analysis* (LDA) attack to extract the key from white-box implementations, in which the target key-dependent variables are linearly encoded by a set of intermediate variables in the implementation. It also explains how this attack can be extended to break implementations protected with higher-degree encodings.

- Later, it introduces *higher-order DCA*, along with an enhanced *multivariate* version, and analyzes the security achieved by combining linear masking and shuffling against these attacks. We derive analytic expressions for the complexity of the attacks – backed up through extensive attack experiments – enabling a designer to quantify the security level of a masked and shuffled implementation in the (higher-order) DCA setting.

- After that, it considers a state-of-the-art white-box implementation in the paradigm of a randomized Boolean circuit with hardcoded key represented in software as a bitsliced program by combining linear masking, non-linear masking and shuffling in different ways. It then analyzes the different gray-box attack paths and studies their performance in terms of required traces and computation time. Most importantly, it proposes an advanced gray-box attack against white-box cryptography which exploits the data-dependency of the target implementation. This approach can efficiently break several combinations of linear and non-linear masking in the presence of shuffling and obfuscation and thus provides substantial complexity improvements over the existing attacks.

- Finally, the practicability of the theoretical analyses and attack techniques exhibited in this thesis are verified by recovering keys in several publicly available white-box AES implementations. Additionally, it summarizes a general attack methodology against obscure white-box implementations, which has been followed to analyze the winning challenges from both WhibOx contests. To best of our knowledge, we were either the *only* or the *first* team to produce technical reports on the possibility to break those implementations by applying relevant theoretically supported gray-box attacks. To facilitate the reproduction of our results, our attack tools have been partially open-sourced.

In conclusion, the theoretical analyses and practical attacks provided in this thesis demonstrate what the capabilities are of a passive gray-box adversary in the white-box context and what the approaches are to countering such an adversary. This thesis pinpoints that the adoption of a unitary and primal countermeasure tends to be ineffective to resist this kind of adversary. A truly effective solution has to combine multiple different countermeasures at the same time to resist different attack vectors. Moreover, the analyses in this thesis are a first step towards achieving provable security against passive gray-box adversaries in white-box context. In particular, they provide quantification of the different attack complexities with respect to certain parameters under certain assumptions. Typically, the adversary is assumed to have some uncertainty on the attack window within a full computation trace. Under this assumption and for some choice of the parameters, a good level of practical resistance against these attacks can be achieved. Last but not least, this thesis stresses that circuit obfuscation techniques, which prevent the adversary from learning structural knowledge of the implementation, play an essential role in achieving the considered security in this thesis.

From this point, we identify a few research directions that would be worth further investigations.

Ensuring the Uncertainty Assumption. The goal of this research direction is to make sure the adversary cannot reduce the size of the target window. This thesis has demonstrated that data-dependency based analysis is a powerful weapon of gray-box adversaries. They can be conducted both manually and automatically to reduce the computational complexity of the underlying attack by using structural information of the attached circuit. It is therefore important to thwart these attacks to be able to achieve some level of (provable) security in practice. More generally, this direction is related to circuit obfuscation techniques.

Building Formal Security Arguments in Passive Gray-Box Attack Model. Another interesting research direction would be to show that the proposed attacks in this thesis (e.g., LDA, HDDA, multi-variate HO-DCA) are somehow optimal in the passive gray-box attack model, and to show that the best an adversary can achieve under some well-defined assumptions can be made arbitrarily hard by, for instance, combining some linear/non-linear masking and obfuscation techniques.

Constructing Higher-Degree Algebraically-Secure Gadgets. If shuffling is not employed, a combination of linear masking and first-degree algebraically-secure non-linear masking in (Biryukov and Udovenko, 2018) can be simply defeated by a second-degree decoding analysis with affordable computation efforts if all encoding shares are in a small trace window. Therefore, another interesting research direction is to construct efficient gadgets secure against decoding analysis of arbitrary degree.

Bibliography

Estuardo Alpirez Bock, Alessandro Amadori, Chris Brzuska, and Wil Michiels (Mar. 2020). "On the Security Goals of White-Box Cryptography". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2020.2, pp. 327–357. DOI: 10.13154/tches.v2020.i2.327-357. URL: https://tches.iacr.org/index.php/TCHES/article/view/8554.

Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang (Aug. 2001). "On the (Im)possibility of Obfuscating Programs". In: *Advances in Cryptology – CRYPTO 2001*. Ed. by Joe Kilian. Vol. 2139. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, pp. 1–18. DOI: 10.1007/3-540-44647-8_1.

Gilles Barthe, Sonia Belad, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini (Oct. 2016). "Strong Non-Interference and Type-Directed Higher-Order Masking". In: *ACM CCS 2016: 23rd Conference on Computer and Communications Security*. Ed. by Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi. Vienna, Austria: ACM Press, pp. 116–129. DOI: 10.1145/2976749.2978427.

Lejla Batina, Benedikt Gierlichs, Emmanuel Prouff, Matthieu Rivain, François-Xavier Standaert, and Nicolas Veyrat-Charvillon (Apr. 2011). "Mutual Information Analysis: a Comprehensive Study". In: *Journal of Cryptology* 24.2, pp. 269–291. DOI: 10.1007/s00145-010-9084-8.

Amos Beimel (2011). "Secret-Sharing Schemes: A Survey". In: *Coding and Cryptology - Third International Workshop, IWCC 2011, Qingdao, China, May 30-June 3, 2011. Proceedings*. Ed. by Yeow Meng Chee, Zhenbo Guo, San Ling, Fengjing Shao, Yuansheng Tang, Huaxiong Wang, and Chaoping Xing. Vol. 6639. Lecture Notes in Computer Science. Springer, pp. 11–46. ISBN: 978-3-642-20900-0. DOI: 10.1007/978-3-642-20901-7_2. URL: https://doi.org/10.1007/978-3-642-20901-7%5C_2.

Sonia Belad, Dahmun Goudarzi, and Matthieu Rivain (Dec. 2018). "Tight Private Circuits: Achieving Probing Security with the Least Refreshing". In: *Advances in Cryptology – ASIACRYPT 2018, Part II*. Ed. by Thomas Peyrin and Steven Galbraith. Vol. 11273. Lecture Notes in Computer Science. Brisbane, Queensland, Australia: Springer, Heidelberg, Germany, pp. 343–372. DOI: 10.1007/978-3-030-03329-3_12.

Davide Bellizia, Francesco Berti, Olivier Bronchain, Gaëtan Cassiers, Sébastien Duval, Chun Guo, Gregor Leander, Gaëtan Leurent, Itamar Levi, Charles Momin,

Olivier Pereira, Thomas Peters, François-Xavier Standaert, and Friedrich Wiemer (2019). "Spook: Sponge-Based Leakage-Resistant Authenticated Encryption with a Masked Tweakable Block Cipher". In.

Eli Biham (Jan. 1997). "A Fast New DES Implementation in Software". In: *Fast Software Encryption – FSE'97*. Ed. by Eli Biham. Vol. 1267. Lecture Notes in Computer Science. Haifa, Israel: Springer, Heidelberg, Germany, pp. 260–272. DOI: 10.1007/BFb0052352.

Eli Biham and Adi Shamir (Aug. 1997). "Differential Fault Analysis of Secret Key Cryptosystems". In: *Advances in Cryptology – CRYPTO'97*. Ed. by Burton S. Kaliski Jr. Vol. 1294. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, pp. 513–525. DOI: 10.1007/BFb0052259.

Olivier Billet, Henri Gilbert, and Charaf Ech-Chatbi (Aug. 2004). "Cryptanalysis of a White Box AES Implementation". In: *SAC 2004: 11th Annual International Workshop on Selected Areas in Cryptography*. Ed. by Helena Handschuh and Anwar Hasan. Vol. 3357. Lecture Notes in Computer Science. Waterloo, Ontario, Canada: Springer, Heidelberg, Germany, pp. 227–240. DOI: 10.1007/978-3-540-30564-4_16.

Alex Biryukov, Charles Bouillaguet, and Dmitry Khovratovich (Dec. 2014). "Cryptographic Schemes Based on the ASASA Structure: Black-Box, White-Box, and Public-Key (Extended Abstract)". In: *Advances in Cryptology – ASIACRYPT 2014, Part I*. Ed. by Palash Sarkar and Tetsu Iwata. Vol. 8873. Lecture Notes in Computer Science. Kaoshiung, Taiwan, R.O.C.: Springer, Heidelberg, Germany, pp. 63–84. DOI: 10.1007/978-3-662-45611-8_4.

Alex Biryukov and Léo Perrin (Dec. 2017). "Symmetrically and Asymmetrically Hard Cryptography". In: *Advances in Cryptology – ASIACRYPT 2017, Part III*. Ed. by Tsuyoshi Takagi and Thomas Peyrin. Vol. 10626. Lecture Notes in Computer Science. Hong Kong, China: Springer, Heidelberg, Germany, pp. 417–445. DOI: 10.1007/978-3-319-70700-6_15.

Alex Biryukov and Adi Shamir (May 2001). "Structural Cryptanalysis of SASAS". In: *Advances in Cryptology – EUROCRYPT 2001*. Ed. by Birgit Pfitzmann. Vol. 2045. Lecture Notes in Computer Science. Innsbruck, Austria: Springer, Heidelberg, Germany, pp. 394–405. DOI: 10.1007/3-540-44987-6_24.

— (Oct. 2010). "Structural Cryptanalysis of SASAS". In: *Journal of Cryptology* 23.4, pp. 505–518. DOI: 10.1007/s00145-010-9062-1.

Alex Biryukov and Aleksei Udovenko (Dec. 2018). "Attacks and Countermeasures for White-box Designs". In: *Advances in Cryptology – ASIACRYPT 2018, Part II*. Ed. by Thomas Peyrin and Steven Galbraith. Vol. 11273. Lecture Notes in Computer Science. Brisbane, Queensland, Australia: Springer, Heidelberg, Germany, pp. 373–402. DOI: 10.1007/978-3-030-03329-3_13.

Estuardo Alpirez Bock, Alessandro Amadori, Joppe W. Bos, Chris Brzuska, and Wil Michiels (Mar. 2019). "Doubly Half-Injective PRGs for Incompressible White-Box

Cryptography". In: *Topics in Cryptology – CT-RSA 2019*. Ed. by Mitsuru Matsui. Vol. 11405. Lecture Notes in Computer Science. San Francisco, CA, USA: Springer, Heidelberg, Germany, pp. 189–209. DOI: 10.1007/978-3-030-12612-4_10.

Estuardo Alpirez Bock, Joppe W. Bos, Chris Brzuska, Charles Hubain, Wil Michiels, Cristofaro Mune, Eloi Sanfelix Gonzalez, Philippe Teuwen, and Alexander Treff (Oct. 2019). "White-Box Cryptography: Don't Forget About Grey-Box Attacks". In: *Journal of Cryptology* 32.4, pp. 1095–1143. DOI: 10.1007/s00145-019-09315-1.

Estuardo Alpirez Bock, Chris Brzuska, Wil Michiels, and Alexander Treff (July 2018). "On the Ineffectiveness of Internal Encodings - Revisiting the DCA Attack on White-Box Cryptography". In: *ACNS 18: 16th International Conference on Applied Cryptography and Network Security*. Ed. by Bart Preneel and Frederik Vercauteren. Vol. 10892. Lecture Notes in Computer Science. Leuven, Belgium: Springer, Heidelberg, Germany, pp. 103–120. DOI: 10.1007/978-3-319-93387-0_6.

Estuardo Alpirez Bock and Alexander Treff (2020). *Security Assessment of White-Box Design Submissions of the CHES 2017 CTF Challenge*. Cryptology ePrint Archive, Report 2020/342. https://eprint.iacr.org/2020/342.

Andrey Bogdanov and Takanori Isobe (Oct. 2015). "White-Box Cryptography Revisited: Space-Hard Ciphers". In: *ACM CCS 2015: 22nd Conference on Computer and Communications Security*. Ed. by Indrajit Ray, Ninghui Li, and Christopher Kruegel. Denver, CO, USA: ACM Press, pp. 1058–1069. DOI: 10.1145/2810103.2813699.

Andrey Bogdanov, Matthieu Rivain, Philip S. Vejre, and Junwei Wang (Apr. 2019). "Higher-Order DCA against Standard Side-Channel Countermeasures". In: *COSADE 2019: 10th International Workshop on Constructive Side-Channel Analysis and Secure Design*. Ed. by Ilia Polian and Marc Stöttinger. Vol. 11421. Lecture Notes in Computer Science. Darmstadt, Germany: Springer, Heidelberg, Germany, pp. 118–141. DOI: 10.1007/978-3-030-16350-1_8.

Dan Boneh, Richard A. DeMillo, and Richard J. Lipton (May 1997). "On the Importance of Checking Cryptographic Protocols for Faults (Extended Abstract)". In: *Advances in Cryptology – EUROCRYPT'97*. Ed. by Walter Fumy. Vol. 1233. Lecture Notes in Computer Science. Konstanz, Germany: Springer, Heidelberg, Germany, pp. 37–51. DOI: 10.1007/3-540-69053-0_4.

Joppe W. Bos, Charles Hubain, Wil Michiels, and Philippe Teuwen (Aug. 2016). "Differential Computation Analysis: Hiding Your White-Box Designs is Not Enough". In: *Cryptographic Hardware and Embedded Systems – CHES 2016*. Ed. by Benedikt Gierlichs and Axel Y. Poschmann. Vol. 9813. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, pp. 215–236. DOI: 10.1007/978-3-662-53140-2_11.

Julien Bringer, Herve Chabanne, and Emmanuelle Dottax (2006). *White Box Cryptography: Another Attempt*. Cryptology ePrint Archive, Report 2006/468. http://eprint.iacr.org/2006/468.

Julien Bringer, Hervé Chabanne, and Emmanuelle Dottax (2006). "Perturbing and Protecting a Traceable Block Cipher". In: *Communications and Multimedia Security, 10th IFIP TC-6 TC-11 International Conference, CMS 2006, Heraklion, Crete, Greece, October 19-21, 2006, Proceedings*. Ed. by Herbert Leitold and Evangelos P. Markatos. Vol. 4237. Lecture Notes in Computer Science. Springer, pp. 109–119. ISBN: 3-540-47820-5. DOI: 10.1007/11909033_10. URL: https://doi.org/10.1007/11909033%5C_10.

Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi (Aug. 1999). "Towards Sound Approaches to Counteract Power-Analysis Attacks". In: *Advances in Cryptology – CRYPTO'99*. Ed. by Michael J. Wiener. Vol. 1666. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, pp. 398–412. DOI: 10.1007/3-540-48405-1_26.

Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi (Aug. 2003). "Template Attacks". In: *Cryptographic Hardware and Embedded Systems – CHES 2002*. Ed. by Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar. Vol. 2523. Lecture Notes in Computer Science. Redwood Shores, CA, USA: Springer, Heidelberg, Germany, pp. 13–28. DOI: 10.1007/3-540-36400-5_3.

Stanley Chow, Philip A. Eisen, Harold Johnson, and Paul C. van Oorschot (2002). "A White-Box DES Implementation for DRM Applications". In: *Security and Privacy in Digital Rights Management, ACM CCS-9 Workshop, DRM 2002, Washington, DC, USA, November 18, 2002, Revised Papers*. Ed. by Joan Feigenbaum. Vol. 2696. Lecture Notes in Computer Science. Springer, pp. 1–15. ISBN: 3-540-40410-4. DOI: 10.1007/978-3-540-44993-5_1. URL: https://doi.org/10.1007/978-3-540-44993-5%5C_1.

Stanley Chow, Philip A. Eisen, Harold Johnson, and Paul C. van Oorschot (Aug. 2003). "White-Box Cryptography and an AES Implementation". In: *SAC 2002: 9th Annual International Workshop on Selected Areas in Cryptography*. Ed. by Kaisa Nyberg and Howard M. Heys. Vol. 2595. Lecture Notes in Computer Science. St. John's, Newfoundland, Canada: Springer, Heidelberg, Germany, pp. 250–270. DOI: 10.1007/3-540-36492-7_17.

Christophe Clavier, Jean-Sébastien Coron, and Nora Dabbous (Aug. 2000). "Differential Power Analysis in the Presence of Hardware Countermeasures". In: *Cryptographic Hardware and Embedded Systems – CHES 2000*. Ed. by Çetin Kaya Koç and Christof Paar. Vol. 1965. Lecture Notes in Computer Science. Worcester, Massachusetts, USA: Springer, Heidelberg, Germany, pp. 252–263. DOI: 10.1007/3-540-44499-8_20.

Christian Collberg, Clark Thomborson, and Douglas Low (1997). *A taxonomy of obfuscating transformations*. Tech. rep. Department of Computer Science, The University of Auckland, New Zealand.

Jean-Sébastien Coron (Aug. 1999). "Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems". In: *Cryptographic Hardware and Embedded Systems – CHES'99*. Ed. by Çetin Kaya Koç and Christof Paar. Vol. 1717. Lecture

Notes in Computer Science. Worcester, Massachusetts, USA: Springer, Heidelberg, Germany, pp. 292–302. DOI: 10.1007/3-540-48059-5_25.

— (May 2014). "Higher Order Masking of Look-Up Tables". In: *Advances in Cryptology – EUROCRYPT 2014*. Ed. by Phong Q. Nguyen and Elisabeth Oswald. Vol. 8441. Lecture Notes in Computer Science. Copenhagen, Denmark: Springer, Heidelberg, Germany, pp. 441–458. DOI: 10.1007/978-3-642-55220-5_25.

Jean-Sébastien Coron and Ilya Kizhvatov (Aug. 2010). "Analysis and Improvement of the Random Delay Countermeasure of CHES 2009". In: *Cryptographic Hardware and Embedded Systems – CHES 2010*. Ed. by Stefan Mangard and François-Xavier Standaert. Vol. 6225. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, pp. 95–109. DOI: 10.1007/978-3-642-15031-9_7.

Jean-Sébastien Coron, Emmanuel Prouff, Matthieu Rivain, and Thomas Roche (Mar. 2014). "Higher-Order Side Channel Security and Mask Refreshing". In: *Fast Software Encryption – FSE 2013*. Ed. by Shiho Moriai. Vol. 8424. Lecture Notes in Computer Science. Singapore: Springer, Heidelberg, Germany, pp. 410–424. DOI: 10.1007/978-3-662-43933-3_21.

Joan Daemen and Vincent Rijmen (2013). *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media.

Yoni De Mulder, Peter Roelse, and Bart Preneel (2013a). *Revisiting the BGE Attack on a White-Box AES Implementation*. Cryptology ePrint Archive, Report 2013/450. http://eprint.iacr.org/2013/450.

— (Aug. 2013b). "Cryptanalysis of the Xiao-Lai White-Box AES Implementation". In: *SAC 2012: 19th Annual International Workshop on Selected Areas in Cryptography*. Ed. by Lars R. Knudsen and Huapeng Wu. Vol. 7707. Lecture Notes in Computer Science. Windsor, Ontario, Canada: Springer, Heidelberg, Germany, pp. 34–49. DOI: 10.1007/978-3-642-35999-6_3.

Yoni De Mulder, Brecht Wyseur, and Bart Preneel (Dec. 2010). "Cryptanalysis of a Perturbated White-Box AES Implementation". In: *Progress in Cryptology - INDOCRYPT 2010: 11th International Conference in Cryptology in India*. Ed. by Guang Gong and Kishan Chand Gupta. Vol. 6498. Lecture Notes in Computer Science. Hyderabad, India: Springer, Heidelberg, Germany, pp. 292–310.

Cécile Delerablée, Tancrède Lepoint, Pascal Paillier, and Matthieu Rivain (Aug. 2014). "White-Box Security Notions for Symmetric Encryption Schemes". In: *SAC 2013: 20th Annual International Workshop on Selected Areas in Cryptography*. Ed. by Tanja Lange, Kristin Lauter, and Petr Lisonek. Vol. 8282. Lecture Notes in Computer Science. Burnaby, BC, Canada: Springer, Heidelberg, Germany, pp. 247–264. DOI: 10.1007/978-3-662-43414-7_13.

Patrick Derbez, Pierre-Alain Fouque, Baptiste Lambin, and Brice Minaud (2018). "On Recovering Affine Encodings in White-Box Implementations". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2018.3. https://

tches.iacr.org/index.php/TCHES/article/view/7271, pp. 121–149. ISSN: 2569-2925. DOI: 10.13154/tches.v2018.i3.121-149.

Pierre Dusart, Gilles Letourneux, and Olivier Vivolo (Oct. 2003). "Differential Fault Analysis on AES". In: *ACNS 03: 1st International Conference on Applied Cryptography and Network Security*. Ed. by Jianying Zhou, Moti Yung, and Yongfei Han. Vol. 2846. Lecture Notes in Computer Science. Kunming, China: Springer, Heidelberg, Germany, pp. 293–306. DOI: 10.1007/978-3-540-45203-4_23.

Joan Feigenbaum, ed. (2003). *Security and Privacy in Digital Rights Management, ACM CCS-9 Workshop, DRM 2002, Washington, DC, USA, November 18, 2002, Revised Papers*. Vol. 2696. Lecture Notes in Computer Science. Springer. ISBN: 3-540-40410-4. DOI: 10.1007/b11725. URL: https://doi.org/10.1007/b11725.

Ronald Aylmer Fisher, Frank Yates, et al. (1938). "Statistical tables for biological, agricultural and medical research". In: *Statistical tables for biological, agricultural and medical research.*

Pierre-Alain Fouque, Pierre Karpman, Paul Kirchner, and Brice Minaud (Dec. 2016). "Efficient and Provable White-Box Primitives". In: *Advances in Cryptology – ASIACRYPT 2016, Part I*. Ed. by Jung Hee Cheon and Tsuyoshi Takagi. Vol. 10031. Lecture Notes in Computer Science. Hanoi, Vietnam: Springer, Heidelberg, Germany, pp. 159–188. DOI: 10.1007/978-3-662-53887-6_6.

Sanjam Garg, Craig Gentry, and Shai Halevi (May 2013). "Candidate Multilinear Maps from Ideal Lattices". In: *Advances in Cryptology – EUROCRYPT 2013*. Ed. by Thomas Johansson and Phong Q. Nguyen. Vol. 7881. Lecture Notes in Computer Science. Athens, Greece: Springer, Heidelberg, Germany, pp. 1–17. DOI: 10.1007/978-3-642-38348-9_1.

Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters (Oct. 2013). "Candidate Indistinguishability Obfuscation and Functional Encryption for all Circuits". In: *54th Annual Symposium on Foundations of Computer Science*. Berkeley, CA, USA: IEEE Computer Society Press, pp. 40–49. DOI: 10.1109/FOCS.2013.13.

Benedikt Gierlichs, Lejla Batina, Pim Tuyls, and Bart Preneel (Aug. 2008). "Mutual Information Analysis". In: *Cryptographic Hardware and Embedded Systems – CHES 2008*. Ed. by Elisabeth Oswald and Pankaj Rohatgi. Vol. 5154. Lecture Notes in Computer Science. Washington, DC, USA: Springer, Heidelberg, Germany, pp. 426–442. DOI: 10.1007/978-3-540-85053-3_27.

G.H. Golub and C.F. Van Loan (1996). *Matrix Computations*. Johns Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press. ISBN: 9780801854149. URL: https://books.google.fr/books?id=mlOa7wPX6OYC.

Louis Goubin, Jean-Michel Masereel, and Michaël Quisquater (Aug. 2007). "Cryptanalysis of White Box DES Implementations". In: *SAC 2007: 14th Annual International Workshop on Selected Areas in Cryptography*. Ed. by Carlisle M. Adams, Ali

Miri, and Michael J. Wiener. Vol. 4876. Lecture Notes in Computer Science. Ottawa, Canada: Springer, Heidelberg, Germany, pp. 278–295. DOI: 10.1007/978-3-540-77360-3_18.

Louis Goubin, Pascal Paillier, Matthieu Rivain, and Junwei Wang (Apr. 2020). "How to reveal the secrets of an obscure white-box implementation". In: *Journal of Cryptographic Engineering* 10.1, pp. 49–66. DOI: 10.1007/s13389-019-00207-5.

Louis Goubin and Jacques Patarin (Aug. 1999). "DES and Differential Power Analysis (The "Duplication" Method)". In: *Cryptographic Hardware and Embedded Systems – CHES'99*. Ed. by Çetin Kaya Koç and Christof Paar. Vol. 1717. Lecture Notes in Computer Science. Worcester, Massachusetts, USA: Springer, Heidelberg, Germany, pp. 158–172. DOI: 10.1007/3-540-48059-5_15.

Louis Goubin, Matthieu Rivain, and Junwei Wang (2020). "Defeating State-of-the-Art White-Box Countermeasures". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2020.3. https://tches.iacr.org/index.php/TCHES/article/view/8597, pp. 454–482. ISSN: 2569-2925. DOI: 10.13154/tches.v2020.i3.454-482.

Dahmun Goudarzi, Jérémy Jean Jean, Kölbl Stefan, Peyrin Thomas, and Rivain Matthieu (2019). "Pyjamask". In.

Dahmun Goudarzi and Matthieu Rivain (Apr. 2017). "How Fast Can Higher-Order Masking Be in Software?" In: *Advances in Cryptology – EUROCRYPT 2017, Part I*. Ed. by Jean-Sébastien Coron and Jesper Buus Nielsen. Vol. 10210. Lecture Notes in Computer Science. Paris, France: Springer, Heidelberg, Germany, pp. 567–597. DOI: 10.1007/978-3-319-56620-7_20.

Máté Horváth (2015). *Survey on Cryptographic Obfuscation*. Cryptology ePrint Archive, Report 2015/412. http://eprint.iacr.org/2015/412.

Yuval Ishai, Amit Sahai, and David Wagner (Aug. 2003). "Private Circuits: Securing Hardware against Probing Attacks". In: *Advances in Cryptology – CRYPTO 2003*. Ed. by Dan Boneh. Vol. 2729. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, pp. 463–481. DOI: 10.1007/978-3-540-45146-4_27.

ISO/IEC 8859-1:1998 (1998). *ISO/IEC 8859-1:1998: Information technology – 8-bit single-byte coded graphic character sets – Part 1: Latin alphabet No. 1*. https://www.iso.org/standard/28245.html. Accessed: October 2017.

Matthias Jacob, Dan Boneh, and Edward W. Felten (2002). "Attacking an Obfuscated Cipher by Injecting Faults". In: *Security and Privacy in Digital Rights Management, ACM CCS-9 Workshop, DRM 2002, Washington, DC, USA, November 18, 2002, Revised Papers*. Ed. by Joan Feigenbaum. Vol. 2696. Lecture Notes in Computer Science. Springer, pp. 16–31. ISBN: 3-540-40410-4. DOI: 10.1007/978-3-540-44993-5_2. URL: https://doi.org/10.1007/978-3-540-44993-5_2.

Anthony Journault and François-Xavier Standaert (Sept. 2017). "Very High Order Masking: Efficient Implementation and Security Evaluation". In: *Cryptographic*

Hardware and Embedded Systems – CHES 2017. Ed. by Wieland Fischer and Naofumi Homma. Vol. 10529. Lecture Notes in Computer Science. Taipei, Taiwan: Springer, Heidelberg, Germany, pp. 623–643. DOI: 10.1007/978-3-319-66787-4_30.

Mohamed Karroumi (Dec. 2011). "Protecting White-Box AES with Dual Ciphers". In: *ICISC 10: 13th International Conference on Information Security and Cryptology*. Ed. by Kyung Hyune Rhee and DaeHun Nyang. Vol. 6829. Lecture Notes in Computer Science. Seoul, Korea: Springer, Heidelberg, Germany, pp. 278–291.

Auguste Kerckhoffs (1883). "La Cryptographic Militaire". In: *Journal des Sciences Militaires*, pp. 5–38.

Paul C. Kocher (Aug. 1996). "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems". In: *Advances in Cryptology – CRYPTO'96*. Ed. by Neal Koblitz. Vol. 1109. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, pp. 104–113. DOI: 10.1007/3-540-68697-5_9.

Paul C. Kocher, Joshua Jaffe, and Benjamin Jun (Aug. 1999). "Differential Power Analysis". In: *Advances in Cryptology – CRYPTO'99*. Ed. by Michael J. Wiener. Vol. 1666. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, pp. 388–397. DOI: 10.1007/3-540-48405-1_25.

Seungkwang Lee, Taesung Kim, and Yousung Kang (2018). "A Masked White-Box Cryptographic Implementation for Protecting Against Differential Computation Analysis". In: *IEEE Trans. Information Forensics and Security* 13.10, pp. 2602–2615. DOI: 10.1109/TIFS.2018.2825939. URL: https://doi.org/10.1109/TIFS.2018.2825939.

Tancrède Lepoint and Matthieu Rivain (2013). *Another Nail in the Coffin of White-Box AES Implementations*. Cryptology ePrint Archive, Report 2013/455. http://eprint.iacr.org/2013/455.

Tancrède Lepoint, Matthieu Rivain, Yoni De Mulder, Peter Roelse, and Bart Preneel (Aug. 2014). "Two Attacks on a White-Box AES Implementation". In: *SAC 2013: 20th Annual International Workshop on Selected Areas in Cryptography*. Ed. by Tanja Lange, Kristin Lauter, and Petr Lisonek. Vol. 8282. Lecture Notes in Computer Science. Burnaby, BC, Canada: Springer, Heidelberg, Germany, pp. 265–285. DOI: 10.1007/978-3-662-43414-7_14.

Huijia Lin (May 2016). "Indistinguishability Obfuscation from Constant-Degree Graded Encoding Schemes". In: *Advances in Cryptology – EUROCRYPT 2016, Part I*. Ed. by Marc Fischlin and Jean-Sébastien Coron. Vol. 9665. Lecture Notes in Computer Science. Vienna, Austria: Springer, Heidelberg, Germany, pp. 28–57. DOI: 10.1007/978-3-662-49890-3_2.

— (Aug. 2017). "Indistinguishability Obfuscation from SXDH on 5-Linear Maps and Locality-5 PRGs". In: *Advances in Cryptology – CRYPTO 2017, Part I*. Ed. by

Jonathan Katz and Hovav Shacham. Vol. 10401. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, pp. 599–629. DOI: 10.1007/978-3-319-63688-7_20.

Huijia Lin and Stefano Tessaro (Aug. 2017). "Indistinguishability Obfuscation from Trilinear Maps and Block-Wise Local PRGs". In: *Advances in Cryptology – CRYPTO 2017, Part I*. Ed. by Jonathan Katz and Hovav Shacham. Vol. 10401. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, pp. 630–660. DOI: 10.1007/978-3-319-63688-7_21.

Hamilton E. Link and William D. Neumann (2005). "Clarifying Obfuscation: Improving the Security of White-Box DES". In: *International Symposium on Information Technology: Coding and Computing (ITCC 2005), Volume 1, 4-6 April 2005, Las Vegas, Nevada, USA*. IEEE Computer Society, pp. 679–684. ISBN: 0-7695-2315-3. DOI: 10.1109/ITCC.2005.100. URL: https://doi.org/10.1109/ITCC.2005.100.

Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazelwood (2005). "Pin: building customized program analysis tools with dynamic instrumentation". In: *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*. Ed. by Vivek Sarkar and Mary W. Hall. ACM, pp. 190–200. ISBN: 1-59593-056-6. DOI: 10.1145/1065010.1065034. URL: https://doi.org/10.1145/1065010.1065034.

Stefan Mangard (Feb. 2004). "Hardware Countermeasures against DPA – A Statistical Analysis of Their Effectiveness". In: *Topics in Cryptology – CT-RSA 2004*. Ed. by Tatsuaki Okamoto. Vol. 2964. Lecture Notes in Computer Science. San Francisco, CA, USA: Springer, Heidelberg, Germany, pp. 222–235. DOI: 10.1007/978-3-540-24660-2_18.

Stefan Mangard, Elisabeth Oswald, and Thomas Popp (2007). *Power analysis attacks - revealing the secrets of smart cards*. Springer. ISBN: 978-0-387-30857-9.

Wil Michiels, Paul Gorissen, and Henk D. L. Hollmann (Aug. 2009). "Cryptanalysis of a Generic Class of White-Box Implementations". In: *SAC 2008: 15th Annual International Workshop on Selected Areas in Cryptography*. Ed. by Roberto Maria Avanzi, Liam Keliher, and Francesco Sica. Vol. 5381. Lecture Notes in Computer Science. Sackville, New Brunswick, Canada: Springer, Heidelberg, Germany, pp. 414–428. DOI: 10.1007/978-3-642-04159-4_27.

Brice Minaud, Patrick Derbez, Pierre-Alain Fouque, and Pierre Karpman (Nov. 2015). "Key-Recovery Attacks on ASASA". In: *Advances in Cryptology – ASIACRYPT 2015, Part II*. Ed. by Tetsu Iwata and Jung Hee Cheon. Vol. 9453. Lecture Notes in Computer Science. Auckland, New Zealand: Springer, Heidelberg, Germany, pp. 3–27. DOI: 10.1007/978-3-662-48800-3_1.

— (July 2018). "Key-Recovery Attacks on ASASA". In: *Journal of Cryptology* 31.3, pp. 845–884. DOI: 10.1007/s00145-017-9272-x.

Amir Moradi, Oliver Mischke, and Thomas Eisenbarth (Aug. 2010). "Correlation-Enhanced Power Analysis Collision Attack". In: *Cryptographic Hardware and Embedded Systems – CHES 2010*. Ed. by Stefan Mangard and François-Xavier Standaert. Vol. 6225. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, pp. 125–139. DOI: 10.1007/978-3-642-15031-9_9.

James A. Muir (2013). *A Tutorial on White-box AES*. Cryptology ePrint Archive, Report 2013/104. http://eprint.iacr.org/2013/104.

Nicholas Nethercote and Julian Seward (2007). "Valgrind: a framework for heavy-weight dynamic binary instrumentation". In: *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*. Ed. by Jeanne Ferrante and Kathryn S. McKinley. ACM, pp. 89–100. ISBN: 978-1-59593-633-2. DOI: 10.1145/1250734.1250746. URL: https://doi.org/10.1145/1250734.1250746.

M. E. J. Newman (June 2004). "Fast algorithm for detecting community structure in networks". In: *Phys. Rev. E* 69 (6), p. 066133. DOI: 10.1103/PhysRevE.69.066133. URL: https://link.aps.org/doi/10.1103/PhysRevE.69.066133.

Emmanuel Prouff and Matthieu Rivain (June 2009). "Theoretical and Practical Aspects of Mutual Information Based Side Channel Analysis". In: *ACNS 09: 7th International Conference on Applied Cryptography and Network Security*. Ed. by Michel Abdalla, David Pointcheval, Pierre-Alain Fouque, and Damien Vergnaud. Vol. 5536. Lecture Notes in Computer Science. Paris-Rocquencourt, France: Springer, Heidelberg, Germany, pp. 499–518. DOI: 10.1007/978-3-642-01957-9_31.

— (May 2013). "Masking against Side-Channel Attacks: A Formal Security Proof". In: *Advances in Cryptology – EUROCRYPT 2013*. Ed. by Thomas Johansson and Phong Q. Nguyen. Vol. 7881. Lecture Notes in Computer Science. Athens, Greece: Springer, Heidelberg, Germany, pp. 142–159. DOI: 10.1007/978-3-642-38348-9_9.

Chester Rebeiro, A. David Selvakumar, and A. S. L. Devi (Dec. 2006). "Bitslice Implementation of AES". In: *CANS 06: 5th International Conference on Cryptology and Network Security*. Ed. by David Pointcheval, Yi Mu, and Kefei Chen. Vol. 4301. Lecture Notes in Computer Science. Suzhou, China: Springer, Heidelberg, Germany, pp. 203–212.

Matthieu Rivain and Emmanuel Prouff (Aug. 2010). "Provably Secure Higher-Order Masking of AES". In: *Cryptographic Hardware and Embedded Systems – CHES 2010*. Ed. by Stefan Mangard and François-Xavier Standaert. Vol. 6225. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, pp. 413–427. DOI: 10.1007/978-3-642-15031-9_28.

Matthieu Rivain, Emmanuel Prouff, and Julien Doget (Sept. 2009). "Higher-Order Masking and Shuffling for Software Implementations of Block Ciphers". In: *Cryptographic Hardware and Embedded Systems – CHES 2009*. Ed. by Christophe

Clavier and Kris Gaj. Vol. 5747. Lecture Notes in Computer Science. Lausanne, Switzerland: Springer, Heidelberg, Germany, pp. 171–188. DOI: 10.1007/978-3-642-04138-9_13.

Matthieu Rivain and Junwei Wang (2019). "Analysis and Improvement of Differential Computation Attacks against Internally-Encoded White-Box Implementations". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2019.2. https://tches.iacr.org/index.php/TCHES/article/view/7391, pp. 225–255. ISSN: 2569-2925. DOI: 10.13154/tches.v2019.i2.225-255.

Rolf Rolles (2009). "Unpacking Virtualization Obfuscators". In: *Proceedings of the 3rd USENIX Conference on Offensive Technologies*. WOOT'09. Montreal, Canada: USENIX Association, pp. 1–1. URL: http://dl.acm.org/citation.cfm?id=1855876.1855877.

Amit Sahai and Brent Waters (May 2014). "How to use indistinguishability obfuscation: deniable encryption, and more". In: *46th Annual ACM Symposium on Theory of Computing*. Ed. by David B. Shmoys. New York, NY, USA: ACM Press, pp. 475–484. DOI: 10.1145/2591796.2591825.

Eloi Sanfelix, Cristofaro Mune, and Job de Haas (2015). *Unboxing the White-Box: Practical attacks against Obfuscated Ciphers* . https://www.blackhat.com/docs/eu-15/materials/eu-15-Sanfelix-Unboxing-The-White-Box-Practical-Attacks-Against-Obfuscated-Ciphers-wp.pdf. Accessed: 2020-04-07. Black Hat Europe 2015.

Amitabh Saxena, Brecht Wyseur, and Bart Preneel (Sept. 2009). "Towards Security Notions for White-Box Cryptography". In: *ISC 2009: 12th International Conference on Information Security*. Ed. by Pierangela Samarati, Moti Yung, Fabio Martinelli, and Claudio Agostino Ardagna. Vol. 5735. Lecture Notes in Computer Science. Pisa, Italy: Springer, Heidelberg, Germany, pp. 49–58.

Kai Schramm, Gregor Leander, Patrick Felke, and Christof Paar (Aug. 2004). "A Collision-Attack on AES: Combining Side Channel- and Differential-Attack". In: *Cryptographic Hardware and Embedded Systems – CHES 2004*. Ed. by Marc Joye and Jean-Jacques Quisquater. Vol. 3156. Lecture Notes in Computer Science. Cambridge, Massachusetts, USA: Springer, Heidelberg, Germany, pp. 163–175. DOI: 10.1007/978-3-540-28632-5_12.

François-Xavier Standaert, Eric Peeters, Gaël Rouvroy, and Jean-Jacques Quisquater (2006). "An Overview of Power Analysis Attacks Against Field Programmable Gate Arrays". In: *Proceedings of the IEEE* 94.2, pp. 383–394. DOI: 10.1109/JPROC.2005.862437. URL: https://doi.org/10.1109/JPROC.2005.862437.

Volker Strassen (Aug. 1969). "Gaussian Elimination is Not Optimal". In: *Numer. Math.* 13.4, pp. 354–356.

Daehyun Strobel and Christof Paar (Nov. 2012). "An Efficient Method for Eliminating Random Delays in Power Traces of Embedded Software". In: *ICISC 11: 14th International Conference on Information Security and Cryptology*. Ed. by Howon Kim.

Vol. 7259. Lecture Notes in Computer Science. Seoul, Korea: Springer, Heidelberg, Germany, pp. 48–60.

LMG Tolhuizen (2012). "Improved cryptanalysis of an AES implementation". In: *Proceedings of the 33rd WIC Symposium on Information Theory, 2012*. WIC (Werkgemeenschap voor Inform.-en Communicatietheorie).

Nicolas Veyrat-Charvillon, Marcel Medwed, Stéphanie Kerckhof, and François-Xavier Standaert (Dec. 2012). "Shuffling against Side-Channel Attacks: A Comprehensive Study with Cautionary Note". In: *Advances in Cryptology – ASIACRYPT 2012*. Ed. by Xiaoyun Wang and Kazue Sako. Vol. 7658. Lecture Notes in Computer Science. Beijing, China: Springer, Heidelberg, Germany, pp. 740–757. DOI: 10.1007/978-3-642-34961-4_44.

WhibOx (2016). *WhibOx 2016 - White-Box Cryptography and Obfuscation*. https://www.cryptoexperts.com/whibox2016/. Accessed: October 2017.

— (2017). *CHES 2017 Capture the Flag Challenge - The WhibOx Contest, An ECRYPT White-Box Cryptography Competition*. https://whibox.cr.yp.to/. Accessed: October 2017.

— (2019). *CHES 2019 Capture the Flag Challenge - The WhibOx Contest Edition 2*. https://whibox-contest.github.io/2019/.

Brecht Wyseur, Wil Michiels, Paul Gorissen, and Bart Preneel (Aug. 2007). "Cryptanalysis of White-Box DES Implementations with Arbitrary External Encodings". In: *SAC 2007: 14th Annual International Workshop on Selected Areas in Cryptography*. Ed. by Carlisle M. Adams, Ali Miri, and Michael J. Wiener. Vol. 4876. Lecture Notes in Computer Science. Ottawa, Canada: Springer, Heidelberg, Germany, pp. 264–277. DOI: 10.1007/978-3-540-77360-3_17.

Y. Xiao and X. Lai (Dec. 2009). "A Secure Implementation of White-Box AES". In: *2009 2nd International Conference on Computer Science and its Applications*, pp. 1–6. DOI: 10.1109/CSA.2009.5404239.

Babak Yadegari, Brian Johannesmeyer, Ben Whitely, and Saumya Debray (May 2015). "A Generic Approach to Automatic Deobfuscation of Executable Code". In: *2015 IEEE Symposium on Security and Privacy*. San Jose, CA, USA: IEEE Computer Society Press, pp. 674–691. DOI: 10.1109/SP.2015.47.

Abbreviations

DCA	**D**ifferential **C**omputation **A**nalysis
DD-DCA	**D**ata-**D**ependency **DCA**
DDG	**D**ata-**D**ependency **G**raph
DFA	**D**ifferential **F**ault **A**nalysis
DPA	**D**ifferential **P**ower **A**nalysis
DRM	**D**igital **R**ight **M**anagement
HDDA	**H**igher-**D**egree **D**ecoding **A**nalysis
HO-DCA	**H**igher-**O**rder **DCA**
LDA	**L**inear **D**ecoding **A**nalysis
MIA	**M**utual **I**nformation **A**nalysis
PRNG	**P**seudo**R**andom **N**umber **G**enerator
RNG	**R**andom **N**umber **G**enerator
$i\mathcal{O}$	**i**ndistinguishability **O**bfuscation
SCA	**S**ide-**C**hannel **A**nalysis
SSA	**S**ingle **S**tatic **A**ssignment
WBC	**W**hite-**B**ox **C**ryptography

Mathematical abbreviations:

CDF	**C**umulative **D**istribution **F**unction
PMF	**P**robability **M**ass **F**unction
VBF	**V**ectorial **B**oolean **F**unction