Software protections for artificial neural networks

Linda Guiga

INSTITUT POLYTECHNIQUE DE PARIS

NNT : 2022IPPAT024

Thèse de doctorat

Software Protections for Artificial Neural Networks

Thése de doctorat de l'Institut Polytechnique de Paris
préparée á Télécom Paris

École doctorale n°626 École doctorale de l'Institut Polytechnique de Paris (EDIPP)
Spécialité de doctorat : Information, Communications, Électronique

Thèse présentée et soutenue à Paris, le 28 juin 2022, par

LINDA GUIGA

Composition du Jury :

Naofumi Homma Professor, Tohoku University (Systems and Software Division)	President
Lejla Batina Professor, Institute for Computing and Information Sciences, Radboud University (Digital Security Group)	Rapporteur
Shivam Bhasin Adjunct Professor, Nanyang Technological University (Temasek Labs)	Examinateur
Melek Önen Associate Professor, EURECOM (Digital Security)	Examinatrice
Bill Roscoe Professor, Oxford University (Department of Computer Science)	Examinateur
Hervé Chabanne Professor, Télécom Paris, Institut Polytechnique de Paris (LTCI - Laboratoire de Traitement et Communication de l'Information)	Directeur de thèse
Jean-Luc Danger Professor, Télécom Paris, Institut Polytechnique de Paris (LTCI - Laboratoire de Traitement et Communication de l'Information)	Co-directeur de thèse
Ulrich Kühne Associate Professor, Télécom Paris, Institut Polytechnique de Paris (LTCI - Laboratoire de Traitement et Communication de l'Information)	Invité

Acknowledgments

First of all, I would like to thank the members of the jury, Pr. Batina, Pr. Homma, Pr. Roscoe, Dr. Önen and Dr. Bhasin, for agreeing to assess my work despite the time investment it implies. More particularly, I would like to thank the reviewers, Pr. Lejla Batina and Pr. Naofumi Homma, for attentively reading my manuscript, as well as for their comments and suggestions to improve it.

I would also like to thank IDEMIA for their support of my thesis and welcoming me in their offices.

A thesis is about pushing yourself to your limit, working as hard as you can to try to achieve research publications. All along, Hervé has accompanied me in this journey, fostering new ideas and encouraging me daily to try my best. Hervé, thank you for your guidance, and your TV show recommendations! I was lucky enough to not have only one, but three advisors! Jean-Luc and Ulrich's ideas and constant encouragement were crucial in the making of this thesis. Their kind advice and comments enabled me to always go forward and do better.

I could not have achieved much without the supportive and resourceful URT team. I would especially like to thank Vincent, who, despite his busy schedule, made sure he was available to provide the help I needed, always with a smile. I am very grateful for all the time invested in answering my interrogations. It was a pleasure working with you!

Paul has always tried his best to provide optimal working conditions, and always had kind words to offer.

Even though I have not had the pleasure of working with the cryptography team so far, they have helped me in other ways. While the whole team has always responded to my technical interrogations – a special mention to Clémence, Luk and Stéphane here –, what I will remember are the coffee and lunch breaks where we could exchange our latest gossip, Switch recommendations, and Go game tips – Nathan, thank you for the games. Amaury, Aurélien, Clémence, Julien, Luk, Nathan, Rina, Roch, Simon and Stéphane, the best of teams!

Throughout this thesis, I have had to overcome various crises. I would never have been able to finish this manuscript and achieve this much in the past three years without the constant support from the people closest to me.

My parents and brother were always there when I needed them, be it to lend a compassionate ear or simply for a light break. I hope I can one day repay you all the sacrifices you have made for me.

Moufida, Azza and Loujaïn were always available to cheer me up and bring me joy. Thank you for your unwavering support!

Jean, Danièle and Éric, you truly are family to me. You have shown me on more

occasions than I deserved just how much I could rely on you.

And last but not least, I cannot thank Fabian enough for his encouragement, support and care. He is an inexhaustible source of – very rigorous – advice, as well as great – or maybe not so great – puns that made life better those past three years.

Résumé en Français

Le Deep Learning (DL) constitue, de nos jours, un élément important de notre vie quotidienne. Le Machine Learning (ML) est utilisé dans presque tous les domaines. Les modèles de ML peuvent prédire un diagnostic en se basant sur des images médicales [101], reconnaître nos visages ou nos empreintes pour nous authentifier [13], ou même prédire et résoudre des crimes [110, 37]. Une application de plus en plus présente est la reconnaissance faciale [13]. Par exemple, L'iPhone X de Apple permet à l'utilisateur de dévérouiller son téléphone grâce à son visage.

Les modèles de ML sont des algorithmes dont le but est d'apprendre une tâche spécifique à travers une phase d'amélioration de ses capacités en se basant sur des données. Cette phase est appelée *l'apprentissage*. Par exemple, en classification d'image, un tel modèle apprend à reconnaître des images et à les classifier dans différentes catégories. De ce fait, si le modèle reçoit une image de chat, il retourne l'étiquette 'chat'. Un modèle de ML est alors considéré comme précis lorsque ses sorties sont correctes avec une forte probabilité.

Pour une tâche donnée, la précision d'un modèle dépend de son architecture et de ses paramètres. Il existe une grande variété de modèles de ML, mais ils sont généralement adaptés à différentes tâches [69, 127, 1, 105]. Par exemple, des réseaux de neurones par convolution (CNN) sont souvent utilisés dans le cadre de la classification et le traitement d'images, puisqu'ils arrivent à détecter efficacement les caractéristiques d'une image [115, 48, 119].

Parmi les modèles de ML, les réseaux de neurones artificiels (NNs) – et plus particulièrement le DL – sont en hausse. Les NNs sont des algorithmes qui cherchent à accomplir leur tâche en imitant le cerveau humain. Ils sont divisés en couches composées de neurones interconnectés. Les valeurs de ces neurones sont mises à jour durant l'entraînement afin d'approcher des prédictions optimales. Malgré leur grand nombre de neurones, les techniques de DL sont en gain de popularité. Cela est dû à leur efficacité à traiter des problèmes complexes nécessitant de larges bases de données. Ils ont aussi l'avantage que la sélection de leur base de données d'entraînement ne nécessite pas d'expertise particulière, et que leur objectif n'a pas besoin d'être subdivisé en des problèmes plus simples. Cela est le cas pour d'autres techniques de ML, pour lesquelles un humain doit prétraiter les données d'entraînement manuellement, ou le problème de départ doit être réduit à plusieurs petits problèmes. Un exemple est l'identification des éléments d'une image en utilisant des Machines à Vecteur de Support (SVM): les éléments doivent d'abord être localisés, puis leur classe doit leur être assignée par la SVM.

La nouvelle capacité à traiter de grands ensembles de données grâce à l'amélioration des capacités de stockage et de calcul a également permis aux techniques

de DL de se développer. Ceci est d'autant plus vrai que plusieurs articles se sont concentrés sur l'accélération des calculs, par exemple par le biais d'accélérateurs [117, 151]. En outre, il a été constaté que les modèles DL pré-entraînés peuvent s'adapter relativement facilement à de nouvelles tâches : c'est ce qu'on appelle l'apprentissage par transfert. Ainsi, les modèles pré-entraînés peuvent être utilisés avec un réglage de précision pour réduire le besoin de grands ensembles de données d'apprentissage.

En raison de leur large utilisation dans notre vie quotidienne ainsi que dans l'industrie, leur sécurité est primordiale. Plus précisément, trois problèmes potentiels doivent être signalés.

Premièrement, les modèles ML eux-mêmes doivent être protégés. La sélection – ou la création – de la meilleure architecture pour la tâche à accomplir est difficile et prend du temps. Mais une fois l'architecture définie, l'apprentissage de ses paramètres prend encore plus de temps et nécessite des ressources de calcul intensif. En effet, pour des tâches complexes, les architectures ML peuvent avoir des millions de paramètres [115, 48]. Même les plus petites architectures, adaptées à une utilisation mobile, ont souvent plus d'un million de paramètres [107]. L'entraînement de ces paramètres pour atteindre une précision presque optimale peut parfois prendre des semaines, même en utilisant plusieurs processeurs de traitement graphique (GPU). C'est le cas d'AlphaGo, une intelligence artificielle jouant au Go, qui a nécessité trois semaines d'entraînement avec 50 GPUs pour battre le meilleur joueur de Go au monde de l'époque [114] Le temps et les ressources informatiques nécessaires pour entraîner correctement les modèles ML en font une propriété intellectuelle (IP).

Le deuxième problème de sécurité concerne la protection des données. L'entraînement des modèles ML repose généralement sur de grands ensembles de données. Ces derniers peuvent être des données sensibles, par exemple dans les domaines médical ou biométrique. Il est donc primordial d'éviter toute fuite de l'ensemble de données d'entraînement. Il est également crucial de protéger l'entrée du modèle au moment de l'exécution, encore une fois pour des raisons de confidentialité. En pratique, cela signifie que les sorties et la structure interne du modèle ne doivent pas révéler d'information sur ses entrées et ses données d'apprentissage.

Enfin, la dernière question de sécurité concerne l'intégrité des entrées et des sorties. Il est primordial que l'utilisateur puisse faire confiance à la sortie du modèle. De plus, certaines applications exigent l'exactitude de la prédiction pour une grande majorité d'entrées. La précision est par exemple critique lorsqu'il s'agit de reconnaissance faciale. Sinon, un utilisateur malveillant pourrait se faire passer pour une autre personne afin d'accéder à des données confidentielles ou d'endommager des fichiers sensibles. Pour cette raison, un tiers ne devrait pas être en mesure d'altérer les entrées ou les prédictions du modèle.

Ces trois problèmes de sécurité peuvent donc être résumés comme suit :

- Le modèle et ses paramètres constituent une propriété intellectuelle et ne doivent donc pas être divulgués.
- Les données stockées par l'utilisateur et les données utilisées pour l'entraînement peuvent être sensibles.
- Les prédictions du modèle doivent être résistantes à l'altération des entrées et des sorties, car cela compromettrait l'intégrité du modèle.

Ce grand intérêt pour les modèles DL a attiré l'attention sur leurs failles de sécurité. Parmi les attaques les plus anciennes et les plus étudiées figurent les attaques adversariales [91, 116]. Dans ce type d'attaque, une partie malveillante ajoute un petit bruit à l'entrée de façon à ce qu'il soit indétectable par un oracle - souvent considéré comme l'œil humain - mais trompe le modèle cible. De telles attaques constituent une faille de sécurité majeure, car elles peuvent être à la base d'usurpations d'identité, par exemple [64]. Les attaques par empoisonnement compromettent l'ensemble de données d'apprentissage en y ajoutant des défauts. Cela compromet une fois de plus l'intégrité du modèle car les prédictions sont alors faussées.

Les attaques par adhésion, quant à elles, s'attaquent à la vie privée de l'utilisateur, en divulguant des informations sur l'ensemble de données d'entraînement [112]. Dans les attaques de vol de modèle, l'attaquant tente de récupérer une entrée chiffrée ou cachée en se basant sur la sortie ou les calculs du modèle.

Enfin, ces dernières années, divers articles sur les attaques de rétro-ingénierie ont été publiés, mettant ainsi en péril la propriété intellectuelle. Certaines sont basées sur des équations et tentent de récupérer les paramètres internes à partir des valeurs de sortie [125]. D'autres approches mathématiques consistent à utiliser la structure interne du modèle ML, et plus particulièrement les hyperplans associés à chaque neurone, pour récupérer les paramètres [16, 103, 60, 87]. Mais les NNs ont également été la cible d'attaques par canaux cachés (SCA), qui utilisent des fuites liées à l'implémentation plutôt que des vulnérabilités mathématiques pour retrouver le secret recherché. Des exemples de vecteurs de fuites incluent le cache [141, 53, 52, 79], la puissance, les émanations électromagnétiques [10, 146, 55, 137] ou le temps écoulé [32, 56].

Par conséquent, les articles ont récemment exploité tous les problèmes de sécurité mentionnés ci-dessus. Même s'ils sont la cible de diverses attaques, les NNs ne sont souvent pas assez protégés, comme l'indique [118]. Cela montre le besoin de protection dans le domaine du ML. En réalité, les auteurs de [16] notent que les NNs n'ont pas été conçus pour être sécurisés, comme le montrent les attaques susmentionnées. Il apparaît alors que trouver de nouvelles protections pour les NNs est un sujet critique. C'est pourquoi cette thèse présente de nouvelles approches pour apporter la sécurité nécessaire.

Nous nous concentrons principalement sur la protection des paramètres et de l'architecture. La raison est triple. Elle permet non seulement de protéger l'IP, mais aussi d'atténuer certaines attaques adversariales et par adhésion, qui nécessitent l'architecture et/ou les paramètres de leurs modèles victimes [96, 113, 112]. Les auteurs de [24] notent même que la recherche adversariale basée sur le gradient - qui nécessitent une certaine connaissance du NN - sont les plus efficaces. La troisième raison est que l'accès au modèle complet entraîne une fuite d'informations sur les sorties du modèle. Comme indiqué dans [6], les entrées des NNs se situent dans un espace de faible dimension et les NNs opèrent une réduction de dimension majeure entre l'espace d'entrée et l'espace de sortie. Par exemple, dans un scénario d'appariement où un système détermine si le vecteur d'un utilisateur est proche de la sortie d'un modèle, un utilisateur malveillant a plus de chances de trouver un vecteur correspondant lorsque le modèle est connu. Pour le démontrer, nous avons

tenté (dans [160]) de faire correspondre la sortie d'un modèle sur 1 000 images de l'ensemble de données ALOI [39] avec les 13 394 images d'Imagenette [54]). Nous avons obtenu 17 correspondances parmi toutes les paires possibles, soit $1.27 \times 10^{-4}\%$ des correspondances. Cela représente une probabilité beaucoup plus élevée que si l'utilisateur avait sélectionné un vecteur de sortie au hasard ($< \frac{1}{284}\%$ dans notre cas).

Les trois raisons susmentionnées montrent la nécessité de protéger le modèle contre les attaques par rétro-ingénierie.

Notons que cette thèse se concentre également sur les défenses logicielles plutôt que matérielles. En effet, les défenses logicielles sont plus faciles à distribuer sans changer l'équipement des utilisateurs, par exemple par le biais d'une mise à jour logicielle. Nous nous concentrons également sur la reconnaissance d'images, car c'est l'exemple le plus commun de NNs que nous rencontrons dans notre vie quotidienne.

Contributions Dans cette thèse, nous commençons par introduire les NNs et leur structure, ainsi que les attaques et défenses qui ont déjà été publiées. Notons qu'au cours du doctorat, nous avons publié une enquête sur les attaques de rétro-ingénierie de l'architecture d'un modèle par canaux cachés, intitulé "Side-Channel Attacks for architecture Extraction of Neural Networks". Cette enquête a été publiée dans *CAAI Transactions on Intelligence Technology* [158]. Dans cette enquête, nous parlons également d'attaques de rétro-ingénierie sur les paramètres et évoquons certaines défenses.

La première partie de ce mémoire se place dans un contexte de boîte grise, où l'attaquant connaît tout ou partie de l'architecture cible. Dans ce cadre, nous proposons des défenses contre l'extraction des paramètres, mais également une protection face à des exemples adversariaux.

Nous commençons par introduire la notion de *modèle parasite*, un élément clé de notre première partie. Il s'agit d'un CNN entraîné à approximer une identité bruitée. Nous plaçons un (ou plusieurs) parasite(s) à l'intérieur du modèle cible afin de changer la structure interne de ce dernier. En effet, nous montrons que cet ajout de couches change les frontières de classification à travers l'introduction d'hyperplans en lien avec les nouveaux neurones. Plus particulièrement, les couches parasites que nous ajoutons en premier lieu contiennent des fonctions d'activation *ReLU* qui rendent nuls les neurones négatifs et gardent la valeur des neurones positifs. Cela introduit non seulement de nouveaux hyperplans correspodant à des neurones valant 0, mais la non linéarité de *ReLU* entraîne également un changement dans les hyperplans de départ. Certaines attaques mathématiques de rétro-ingénierie [56, 103, 16, 27] se basent sur ce type d'hyperplans afin de retrouver les paramètres du modèle cible. En changeant la structure interne du modèle de départ, nous rendons alors ce type d'attaques plus difficiles. En raison de leur fort lien avec les frontières de classifications, nous mesurons l'impact de notre contremesure sur les exemples adversariaux générés et observons un grand changement dans ces exemples: de nombreux exemples générés sur le modèle de départ (pouvant aller jusqu'à environ 40%) ne sont plus adversériaux pour le modèle protégé. Cette observation montre que notre technique a bien mené à un changement dans la structure interne du modèle de départ. Ces résultats ont été présentés à ICISSP 2021, sous le titre "A

Protection against the Extraction of Neural Network Models" [157].

Bien que ce changement de structure impacte les attaques mathématiques, il est insuffsant pour se protéger des attaques SCA physiques se basant sur la puissance ou les émanations électromagnétiques [10]. Pour pallier à cela, nous introduisons alors du dynamisme aux parasites présentés, et les plaçons à l'entrée du modèle à protéger. Ces attaques sont des attaques statistiques qui se basent sur des traces de puissance ou électromégnétiques, ainsi que sur la connaissance des données d'entrée, afin d'extraire les paramètres – et même l'architecture – du modèle. Sélectionner un modèle parasite à chaque exécution du modèle permet de cacher l'entrée du modèle et d'ainsi se défendre contre ces attaques statistiques. Nous proposons donc de sélectionner, pour chaque exécution, au moins un parasite à placer en entrée. Ainsi, nous nous sommes assurés qu'un utilisateur malicieux ne puisse voir qu'une entrée changeant constamment, l'empêchant d'extraire les paramètres précis du modèle. Notons qu'en plaçant les parasites en entrée, nous misons également sur l'effet domino: le bruit introduit en entrée et extrait par un attaquant est amplifié par les couches suivantes. Ce travail a mené à une publication à SPACE 2021, sous le nom "Parasite: Mitigating Physical Side-Channel Attacks against Neural Networks" [155].

En étudiant la protection des paramètres d'un modèle à l'aide de nos parasites, nous avons remarqué l'impact que ces parasites avaient sur les exemples adversériaux. Cela nous a menés à la troisième contribution de cette thèse: une protection contre des exemples adversariaux. En effet, bien que nos parasites ne faisaient que changer les exemples adversariaux générés, nous les adaptons afin de limiter les attaques adversériales. Pour cela, nous considérons à nouveau une introduction dynamique de modèles parasites à l'intérieur du modèle cible. Cette fois, le type de parasites que nous utilisons est différent. Nous choisissons d'entraîner des autoencodeurs, qui compressent les données d'entrée (encodage), puis leur redonnent leur forme initiale (décodage). La compression intermédiaire permet d'éliminer du bruit, et donc de potentiellement éliminer du bruit adversarial. Le décodage, quant à lui, introduit des détails différents de ceux de départ. Les autoencodeurs ont déjà été utilisés avec comme objectif l'élimination d'exemples adversariaux. Cependant, leur entraînement est différent du nôtre, et, surtout, ne considèrent pas le dynamisme que nous proposons. Le dynamisme fait en sorte qu'un attaquant n'ait accès qu'à une structure changeante. En générant des exemples sur une structure donnée, ces exemples ne sont alors pas nécessairement transférables aux autres structures, comme nous le montrons dans notre manuscrit [159].

Une autre méthode permettant la protection des paramètres d'un réseau de neurones est de protéger l'architecture directement. En effet, les attaques de rétro-ingénierie existantes peuvent soit supposer que l'attaquant connaît l'architecture de base [16], soit extraire l'architecture en même temps que les paramètres [10]. De plus, l'architecture constitue en elle-même une IP. En effet, l'architecture, et notamment la sélection de ses hyper-paramètres, est cruciale dans l'entraînement du modèle. Une architecture trop petite aura du mal à apprendre correctement ses données, tandis qu'un modèle trop gros prendra plus de temps à apprendre et risque de ne pas se généraliser à des entrées extérieures aux données d'entraînement. C'est pourquoi, dans une deuxième partie, nous nous intéressons à la protection, en boîte

noire, de l'architecture des NNs.

Plus particulièrement, nous mitigeons une attaque SCA se basant sur le cache. Etant donné que les attaques d'extraction d'architecture utilisent généralement l'exécution séquentielle des couches de NNs, nous proposons de changer l'ordre de calculs des neurones afin de les contrer. Pour cela, nous calculons des neurones par blocs et en profondeur. Afin de rendre la tâche de l'attaquant encore plus difficile, nous ajoutons de l'aléa dans la taille des blocs de calculs. Ainsi, nous générons des tailles de blocs aléatoires à la création de l'architecture, puis nous entamons des calculs dès qu'un bloc est prêt (i.e. lorsqu'assez de valeurs sont parvenues depuis la couche précédente). Notons que ce réarrangement des calculs est possible uniquement pour certains types de couches, telles que les couches convolutionnelles. Cependant, ces couches sont très répandues et nous montrons que nous arrivons à protéger une architecture connue: VGG16. Les résultats de cette contribution ont été publiés dans ACNS 2021 en tant que "Telepathic Headache: Mitigating Cache Side-Channel Attacks on Convolutional Neural Networks".

Contents

List of Figures

List of Tables

Acronyms

BN Batch Normalization.

BNN Binary Neural Network.

CEMA Correlation Electromagnetic Analysis.

CNN Convolutional Neural Network.

CPA Correlation Power Analysis.

DL Deep Learning.

DNN Deep Neural Network.

EM Electromagnetic.

FAB Fast Adaptive Boundary.

FAR False Acceptance Rate.

FC Fully Connected.

FCN Fully Connected Network.

FGSM Fast Gradient Sign Method.

FRR False Rejection Rate.

GAN Generative Adversarial Network.

GeMM General Matrix Multiply.

GPU Graphics Processing Unit.

ILSVRC ImageNet Large Scale Visual Recognition Challenge.

IP Intellectual Property.

LLC Last Level Cache.

LUT Look-Up Table.

ML Machine Learning.

MLaaS Machine Learning as a Service.

MLP Multi-Layer Perceptron.

MLR Multiclass Logistic Regression.

NN Neural Network.

PGD Projective Gradient Descent.

RAW Read-After-Write.

SCA Side-Channel Attack.

SEMA Simple Electromagnetic Analysis.

SGD Stochastic Gradient Descent.

SPA Simple Power Analysis.

SVM Support-Vector Machine.

TEE Trusted Execution Environment.

Chapter 1: Introduction

Deep Learning (DL) now forms an important part of our daily lives. Almost all fields in technology rely on Machine Learning (ML) models to achieve their goals. ML models can predict a diagnosis based on medical images [101], recognize our faces or our fingerprints for authentication [13] or predict and solve crimes [110, 37]. One application on the rise is facial recognition. For instance, Apple's iPhone X [13] enables its user to unlock their phone through facial recognition.

ML models are algorithms whose aim is to learn a specific task through a data-based automatic improvement phase called the *learning* phase. For instance, in image classification, such a model learns to recognize images and classify them into the correct categories. As such, if it receives a picture of a cat, it should return the label 'cat'. An ML model is then considered accurate when its outputs are correct with a high probability.

For a given task, a model's accuracy depends on its architecture and its parameters. There is a large variety of ML models, but they are generally adapted to different tasks [69, 127, 1, 105]. For example, Convolutional Neural Networks (CNNs) are often used in image classification and processing, as they are efficient when it comes to detecting target features [115, 48, 119].

Among Machine Learning models, Neural Networks (NNs) – and more specifically Deep Learning – have been on the rise [20, 68, 48, 115]. NNs are algorithms which aim at learning their task by imitating the brain. They are divided into layers comprised of interconnected *neurons*. The neurons' values are updated during training to ensure optimal predictions. Despite their large number of neurons, DL techniques have been gaining in popularity, due to their efficiency when dealing with complex problems requiring large datasets. They also have the advantage of necessitating little expertise when selecting training data, and the target task does not need to be broken down into smaller, easier problems. Other ML techniques do need this division of the main problem into smaller ones, or human intervention for preprocessing or selecting data. It is the case for identifying elements in an image using Support Vector Machines (SVM) for instance: the elements first need to be located, then their label is assigned by the SVM [23].

The new ability to deal with large datasets thanks to enhanced storing and computing capacities has also enabled DL techniques to surge. This is especially the case since several papers have been focusing on accelerating computations, for instance through DL accelerators [117, 151]. Moreover, it has been noted that pretrained DL models can relatively easily adapt to new tasks: this is called transfer learning. Thus, pretrained models can be used with fine-tuning to reduce the needs

for large sets of training data.

Because of their wide use in our daily lives as well as in the industry, security is paramount. More specifically, three potential issues should be flagged.

First, ML models themselves should be protected. Selecting – or creating – the best architecture for the task at hand is difficult and time consuming. But once the architecture is set, training its parameters takes even longer, and requires resources for intensive computing. Indeed, for complex tasks, ML architectures can have millions of parameters [115, 48]. Even smaller ones, adapted to mobile use, often have over a million parameters [107]. Training those parameters to achieve an almost optimal accuracy can sometimes take weeks, even using several Graphics Processing Units (GPUs). This was the case for AlphaGo, a Go playing artificial intelligence, which required three weeks of training using 50 GPUs to beat the best Go player in the world at the time [114]. The time and computational resources required to correctly train ML models make them Intellectual Property (IP).

The second security issue deals with data protection. The training of ML model generally relies on large datasets. The latter might be sensitive data, for instance in the medical or biometrics fields. It is therefore paramount to prevent leakage of the training dataset. It is also crucial to protect the model's input at run time, once again for user privacy reasons. In practice, it means that the model's outputs and internal structure should not reveal information about its inputs and training data.

Finally, the last security matter concerns the input and output integrity. It is paramount that the user can trust the model's output. Moreover, some applications require correctness of the prediction for a vast majority of inputs. It is for instance critical when it comes to facial recognition. Otherwise, a malicious user might impersonate another person to access confidential data or damage sensitive files. For this reason, a third party should not be able to tamper with either the input or predictions of the model.

The three security concerns can thus be summarized as follows:

- The model and its parameters constitute IP and should therefore not be leaked.
- The user's stored data and the data used for training might be sensitive.
- The model's predictions should be robust against tampering with inputs and outputs, as it would otherwise undermine the model's integrity.

This high interest in DL models has led to a new focus on their security failures. Among the oldest and most studied attacks are adversarial ones [91, 116]. In this type of attack, a malicious party adds a small noise to the input such that it is undetectable to an oracle – often taken as the human eye – but fools the target model. Such attacks constitute a serious security issue, as they can be the basis of impersonation attacks, for example [64]. Poisoning attacks compromise the training dataset by adding faults to it. This once again jeopardizes the model's integrity, as it falsifies the predictions.

Membership attacks, on the other hand, tackle the user's privacy, by leaking information about the training dataset [112]. In model stealing attacks, the attacker tries to recover an encrypted or hidden input based on the model's output or computations.

Finally, in recent years, various reverse-engineering attack papers have been published, thus jeopardizing the IP. Some are equation-based, trying to recover the inner parameters from the output values [125]. Other mathematical approaches include the use of the ML model's internal structure, and more specifically the hyperplanes associated to each neuron, to recover the parameters [16, 103, 60, 87]. But NNs have also been the target of side-channel attacks (SCAs), which use implementation-related leakages rather than mathematical vulnerabilities to recover the sought out secret. Examples of leakage vectors include the cache [141, 53, 52, 79], power or electromagnetic emanations [10, 146, 55, 137] or time elapsed [32, 56].

Therefore, papers have recently exploited all security matters mentioned above. Besides being the target of various attacks, NNs are also often not protected enough, as stated in [118]. This shows the need of protection in the ML field. In fact, the authors of [16] note that NNs were not designed to be secure, as shown by the aforementioned attacks. It is therefore apparent to us that finding new protections for NNs is a critical subject. This is why this thesis presents new approaches to providing the needed security.

We mainly focus on parameter and architecture protection. The reason is threefold. It not only enables the protection of IP, but it also mitigates some adversarial and membership inference attacks which require the architecture and/or parameters of their victim models [96, 113, 112]. The authors of [24] even note that gradient-based adversarial search – which does require some knowledge about the NN – are the most efficient ones. The third reason is that access to the full model leaks information about the outputs of the model. As noted in [6], NNs inputs lie in a low dimensionality manifold and NNs operate a major dimension reduction between the input space and the output space. For instance, in a matching scenario where a system determines whether a user's vector is close to the output of a model, a malicious user is more likely to find a matching vector when the model is known. To show this (in the context of our submitted paper [160]), we tried matching the output of a model on 1,000 images from the ALOI dataset [39] with the 13,394 images from Imagenette [54]. We got 17 matches amongst all possible pairs, or $1.27 \times 10^{-4}\%$ of matches, which is a much higher probability than if the user had selected an output vector at random ($< \frac{1}{2^{84}}\%$).

The three aforementioned reasons show the necessity to protect the model against reverse-engineering attacks.

Let us note that this thesis also focuses on software rather than hardware-based defenses. Even though hardware-based defenses are generally more efficient and less time consuming, they are less flexible. Indeed, software-based defenses are easier to distribute without changing the users' equipment, for instance through a software update. This is even more the case for IDEMIA, whose Machine Learning team relies on software approaches for its products. We also focus on image recognition, as it is the most intuitive instance of NNs that we come across in our daily lives.

Contributions in this Manuscript

This dissertation's contributions are the following:

- We present NNs, as well as attacks and defenses against them. In particular, we published a survey about architecture extraction SCAs [158] (Chapter 2).
- We propose to add one or several parasitic model(s) to an NN to protect its weights against some mathematical reverse-engineering attacks (Chapter 3).
- We feed an NN's input to a dynamically selected parasite before sending it to the base model. Hiding the input in this way constitutes a protection against some physical parameter extraction SCAs (Chapter 4).
- We show that we can adapt the parasites to make NN models more robust against adversarial attacks (Chapter 5).
- We protect an NN's architecture against reverse-engineering SCAs by reordering the way its neurons are computed (Chapter 6).

Overview

Due to their sheer number, an NN's parameters are difficult to protect. For all the aforementioned reasons, they are very sensitive, and ensuring their security is crucial. We seek to achieve this security in our thesis. We start by giving some background on NNs in Chapter 2. This enables us to explain the various attacks that target a model's architecture and parameters, as well as the existing defenses. Since one of the motivations of our thesis is to protect a model against adversarial attacks, we also present them, along with their defenses.

After setting the foundations of this work, we dive into a first part which focuses on the use of parasitic models as a protection against gray-box attacks in Part I. We introduce the notion of *parasites* in Chapter 3. Incorporating such models in a base model leads to a change in its internal structure. More precisely, we show that the additional layers alter the classification boundaries, as adversarial samples generated on the original model are not necessarily adversarial for the protected model.

While this modification of the structure is enough to tackle some mathematical reverse-engineering attacks, side-channel reverse-engineering attacks still manage to recover the parameters of a model protected with the first countermeasure. To extract precise weights, most side-channel attackers only need to gather power or electromagnetic (EM) traces associated to known inputs. Based on this, we add dynamism to the previously introduced parasites to hide a model's input. Placing the parasites at the input layer not only hides the input values, but it yields a domino effect, as explained in another publication [154]. Indeed, parasitic layers early in the target model impact later layers as well. Moreover, dynamism is more efficient against statistical attacks. In Chapter 4 we therefore place the parasites in a dynamic fashion at the entrance of the victim model to counter a statistical physical SCA.

While these countermeasures aim at protecting a model's parameters, we note in Chapter 5 that a similar technique also mitigates adversarial attacks. We adapt the parasitic models to procure a defense against adversarial samples.

An attacker cannot recover an NN's parameters without prior knowledge about its architecture. Another way of securing the learnt weights is therefore to focus on hiding the architecture itself. Protecting the architecture also has the advantage of preventing other types of threats, such as membership inference or adversarial attacks. Architectures are also a key element in a model's training, and therefore constitute intellectual property. This is why, in Part II, we propose a novel way of carrying out NN computations, which mitigates current black-box attacks that can recover an NN's architecture. Instead of keeping the sequential execution of NNs, we show in Chapter 6 that carrying out neuron evaluations by blocks in a randomized fashion mitigates cache-based architecture extraction attacks.

Finally, we summarize our countermeasures and consider potential future work in Chapter 7.

Chapter 2: Literature Review

This chapter details the notions related to Neural Networks, their attacks and existing defenses.

2.1 Neural Networks

Neural Networks (NNs) are algorithms trained to carry out a preset Machine Learning (ML) task. They are comprised of a set of interconnected nodes called *neurons*. Each connection between those neurons is associated to a real number, its *weight*. This network of neurons is inspired by biological brains.

Applications An NN can be assigned a variety of tasks, which can generally be categorized as follows:

- Classification. In classification tasks, an NN knows N classes and, given input I predicts the class number I belongs to. For instance, an NN might seek to differentiate between dogs and cats [97], or identify digits [72]. The medical field benefits from classifiers, as they can help identify cancerous cells, for instance. In language processing, NNs can learn to classify texts, which is essential when it comes to web browsing for example. Segmentation can also fall under this category. In segmentation, an NN is tasked with identifying a certain target within the input. Self-driving cars are an application of segmentation: identifying some objects and shapes in the environment is imperative in that context [106]. Classification and pattern recognition also come up in various aspects of robotics. For instance, classifiers enable prosthetics to have a behaviour similar to a person's missing limb [92].

- Function Approximation. Based on previous data, NNs can predict future occurrences. Such models are useful in finance, for example for stock price predictions [84]. Recent papers have also applied NNs to weather forecast. Google Research scientists managed to forecast precipitation up to twelve hours ahead [36].

- Data processing. When processing data in an unsupervised training – i.e. when the correct labels are not provided – setting, the NN learns to identify some characteristics features within a dataset. Clustering is an instance of data processing wherein the NN aims at dividing the data provided into various

groups. Biology often requires such a pattern or feature identification, for instance when it comes to analysing the genome. Clustering techniques have therefore been applied to that field [65]. Generative Adversarial Networks (GANs) train to fool another NN as a way of learning to imitate the training dataset. They can be exploited for art, for example in order to turn photos into Hayao Miyazaki-style cartoons [4]. They can also help increase the size of training dataset for more accurate NNs and detect anomalies [38], among other applications.

This non-exhaustive list of NN applications shows how vast the field is. We cannot address all applications. We restrict ourselves to image classification in this thesis.

Accuracy A model is trained by updating its weights using a large dataset to reach a high accuracy. The first question here is how to measure a model's accuracy. Several metrics exist, but we will only present the most common ones that we need for this thesis.

First, when an NN's task is to classify images among C classes, then the metric generally used is the percentage of correct labels output by the model on the considered input set. On a dataset $\mathcal{D}$ of size N and with labels Y, we compute a model f's accuracy by measuring the percentage of elements $x \in \mathcal{D}$ such that $f(x) = Y_x$ where Y_x is x's correct label. This can be written as:

$$A_f^{\mathcal{D}} = \frac{1}{N} \sum_{x \in \mathcal{D}} \mathbb{1}_{f(x)=Y_x} \times 100$$

Another classification task consists in matching two inputs whenever they correspond to the same class. This method is used in facial or biometric recognition. Let us place ourselves in the context where a user U wishes to authenticate into a system S which has a picture x_U of U. U sends a picture x to the model f. Then, the system computes $d(f(x_U), f(x))$ where $d(\cdot)$ is a preset distance. If $d(f(x_U), f(x)) \leq t$ for a preset threshold t, then the system considers that x and x_U are both pictures of user U. Otherwise, the authentication is refused. In other words, given an input x, f computes an output vector such that images x' from the same class as x have an output $f(x')$ close – under a certain norm d – to $f(x)$. On the other hand, images x'' from a different class are such that $f(x'')$ is far from $f(x)$. In this scenario, the accuracy is measured through a False Acceptance Rate (FAR) and a False Rejection Rate (FRR). The FAR is the probability of a user $U' \neq U$ being accepted when trying to authenticate as user U. On the other hand, the FRR is the probability of a user U being rejected when trying to be authenticated as themself. While a high FAR results in security issues, a high FRR results in inconvenience to the users. The goal of the NN model and the system is therefore to reach a balance between these two metrics. While both metrics are important, the accuracy is often measured by choosing an FAR and measuring the corresponding FRR. Let us note that one can set the FAR by selecting the appropriate threshold t.

Given a threshold t, a set X of 'gallery' images with one image x_U per user U, and a 'probe' set X_p divided in groups of images X_p^U per user U, then the FAR can

be computed as:

$$FAR = \frac{1}{|X|} \sum_{x_U \in X} \frac{1}{|X_p^{U'}|} \sum_{x \in X_p^{U'}; U' \neq U} \mathbb{1}_{d(x_U, x) \leq t}$$

where $|X|$ is the cardinality of X.

Similarly, the FRR can be computed as:

$$FRR = \frac{1}{|X|} \sum_{x_U \in X} \frac{1}{|X_p^{U}|} \sum_{x \in X_p^{U}} \mathbb{1}_{d(x_U, x) > t}$$

Label matching, FAR and FRR are the most common metrics in image classification, and the only ones we will use in the rest of this manuscript.

Training Phase Once an NN is assigned a task, the training phase can start. The NN is provided with a – preferably large – training dataset, and the weights are updated according to the data received over several rounds – or epochs. This updating process is called *training*.

During this phase, the weights change each round so as to optimize a carefully selected function called *loss function*. This process is generally executed by batches: the training dataset is divided into smaller datasets, and the optimization over each of the batches contributes to reducing the loss value. The testing phase is commonly called the *inference*.

The loss function $\mathcal{L}$ should be chosen so that when $\mathcal{L}$ decreases, the model's accuracy increases. The loss function and optimization algorithm both contribute greatly to the NN model's final accuracy. Indeed, optimizing an incorrect loss function would prevent the model from reaching the set goal. On the other hand, a poorly chosen optimization method can make the training too long or reach – and be stuck at – a suboptimal point.

More than the loss and optimization algorithms, an NN model's efficiency depends on the selected architecture. Various types of NNs have been introduced so far. Even though the architecture should be selected depending on the goal, there is a transferability factor that makes training much easier. Indeed, it has been noted that a model trained on a certain dataset can still have a high accuracy on a different but somewhat similar dataset [122]. For this reason, standard pretrained architectures with small tweaks are often used. Transferability is one of the reasons why NNs have risen in popularity, along with increased storing and computing abilities.

Let M be a pretrained model, on a large dataset $\mathcal{D}_1$. Then one can train a model M' on a smaller dataset $\mathcal{D}_2$ based on M. For this, one takes M's weights for the initialization. One can then add or remove a few layers at the end, to adapt the model M' to their needs. M' can then be trained on $\mathcal{D}_2$. Because the initial weights were already optimized for a large dataset, the new training is usually much shorter. This is also the reason why $\mathcal{D}_2$ can be small. Such a retraining is called *fine-tuning*.

Despite the transferability, selecting the correct pretrained model is still paramount. Not only are some architectures better adapted to some tasks, but it is also important to consider the number of parameters to (re)train during the fine-tuning.

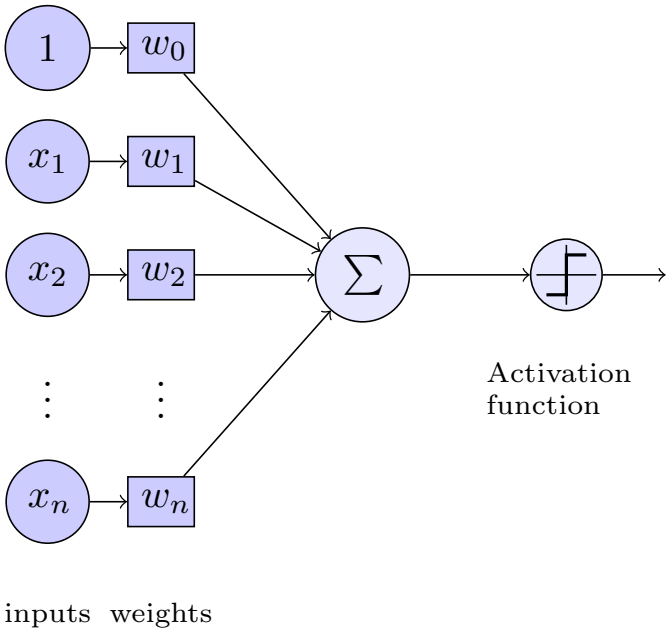


Figure 2.1: Simplest NN type: perceptron with n inputs $\{x_i\}_{i=1,\dots,n}$ and associated weights $\{w_i\}_{i=1,\dots,n}$.

2.2 Common NN Architectures

This section details some common NN architectures for image processing. They generally constitute the baseline in papers, and the implementation and weights of the more complex NNs are also available in most ML libraries.

2.2.1 General structure

Most NNs are divided into layers, where the input is the first layer and the output is the last. The layers between the input and output ones are called *hidden* layers. NNs can be represented by oriented graphs where the neurons are the nodes and the edges are labeled by the weights. Within one layer, neurons are not interconnected. The neurons within a layer can be connected to those from any of the previous layers.

For instance, in a fully connected layer, each neuron is equal to the sum of all the previous layers multiplied by their associated weights (see Figure 2.1). A bias is then added to each output neuron. For the i-th neuron in layer l, we have that:

$$x_i^l = \sum_{k=1}^{N} x_k^{l-1} \cdot w_{k,i}^{l-1} + \beta_i$$

where $\{x_i^l\}_{1,\dots,N}$ are layer l's neurons, $\{w_{i,j}^{l-1}\}_{1,\dots,N}$ are layer $(l-1)$'s weights, and β_i is the bias.

As in the fully connected case, all NN layers are linear. Having an NN only carry out linear computations would limit it to simple tasks which might be solvable in other ways. For NNs to be able to solve a great variety of tasks, some nonlinearity

is therefore required. This is why each layer is followed by a nonlinear *activation function*. They are thus called because they *activate* neurons corresponding to important features and deactivate others.

The following are the most common activation functions:

- Rectified Linear Unit function ($ReLU$): $ReLU(x) = \max(0, x)$. In this function, negative neurons are deactivated. Because it is highly likely that many neuron values will be negative and therefore set to 0 by $ReLU$, this activation functions makes computations easier. $ReLU$ also has the advantage that its hardware computation is cheap.

- Softmax: $softmax(x) = \frac{e^x}{\sum_{j=0}^{c} e^{x_j}}$ where $x \in \mathbb{R}^c$. This function turns a layer's neuron values into probabilities. It is often used in the last layer of classifiers: the NN's prediction corresponds to the class – the neuron number – with highest probability.

- Logistic function: $\sigma(x) = \frac{1}{1+e^{-x}}$. This function turns its inputs into a value between 0 and 1. Values much larger than 1 are set to a value close to 1, and large negative values are set to a value close to 0

- Tanh: $tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$. Similarly to σ, $tanh$ clips its input value between -1 and 1.

Nonlinearity is sometimes introduced via other types of layers. Some aim at reducing the dimensionality of their input. This is the case, for instance, for dropout or pooling layers. Dropout layers select a subset of their input for which the value is kept, while the other neurons are set to 0. Pooling layers reduce the size of their input by dividing their input into blocks and selecting one representative value for each block. For instance, max pooling returns the maximum value of each block, while average pooling layers take the average of each block.

Furthermore, many researchers focus on making the training phase faster and more efficient. In 2015, the authors of [58] introduce Batch Normalization (BN) layers, which normalize their input to stabilize, improve the performance and efficiency of the training. For an input I, in a batch B, the BN layer returns:

$$O_i = \gamma \times \frac{I_i - \mu_B}{\sqrt{V_B + \epsilon}} + \beta$$

where γ, β, ϵ are learnt parameters, and μ_B, V_B are the batch's expected value and variance respectively.

The architecture of an NN is defined by its hyperparameters, namely:

- The number of layers.

- The number of neurons per layer.

- The layer types, along with the layers' internal hyperparameters (for instance, block sizes in pooling layers).

- The activation functions.

- The connections between layers.

While the hyperparameters are chosen once and for all, the parameters need to be trained to reach the optimal accuracy. The most common training algorithm is backpropagation. Given a carefully selected loss function $\mathcal{L}$, the backpropagation algorithm tries to minimize $\mathcal{L}$ by efficiently computing its gradient with respect to the weights using the chain rule. Because the gradients are computed efficiently, gradient-based optimization methods, such as stochastic gradient descent (SGD), can be used to minimize $\mathcal{L}$, even for deep models.

As stated previously, the loss function selection is critical, as it measures the distance between the current model's predictions and the target ones. One such function is the *cross entropy* function, which is suitable for classification tasks. For correct predictions p where $p(i) = 1$ if i is the correct class and $p(j) = 0 \; \forall j \neq i$, and for current predictions y, cross entropy is defined as follows [62]:

$$\begin{aligned} \mathcal{L}(y) &= -\sum_{i=0}^{c-1} p_i \log(y_i) \\ &= -H(p) + D_{KL}(y||p) \end{aligned} \tag{2.1}$$

where c is the number of classes, $H(p)$ is the entropy of p, and $D_{KL}(y||p)$ is the Kullback-Leibler divergence between y and p. Once $\mathcal{L}$ has been chosen, the weights are updated through gradient computations. For instance, in the case of SGD, the update is carried out as follows:

$$w_{t+1} = w_t - l \cdot \nabla \mathcal{L}$$

where w_t are the weights of layer t, l is a scalar called the *learning rate* and $\nabla \mathcal{L}$ is the gradient of $\mathcal{L}$ with relation to w_t. Various losses and optimization algorithms can be used, depending on the task at hand [62].

An issue which might arise when training a model is *overfitting*. Overfitting means that the model does not geenralize well to inputs outside from the training set. This might occur when the architecture is too complex for the task at hand, or the model was trained for too long on too small a dataset [42]. Since the model then tends to learn unimportant details – such as background noise –, its accuracy tends to drop when given inputs from different datasets. This shows the importance of correctly selecting the architecture, training set, loss function and optimization algorithm.

Given the high – and expanding – number of hyperparameters, the set of possible NN architectures is infinite – but limited by computing and storing capacities. In the rest of this dissertation, we will focus on architectures related to image classification. In the following Section, we detail common NN architectures.

2.2.2 Multi-Layer Perceptron

The simplest NN architecture is the *perceptron*. As shown in 2.1, the perceptron is only comprised of one layer and one output neuron. The output is computed as:

$$O = step(\sum_{i=1}^{n} x_i \cdot w_i)$$

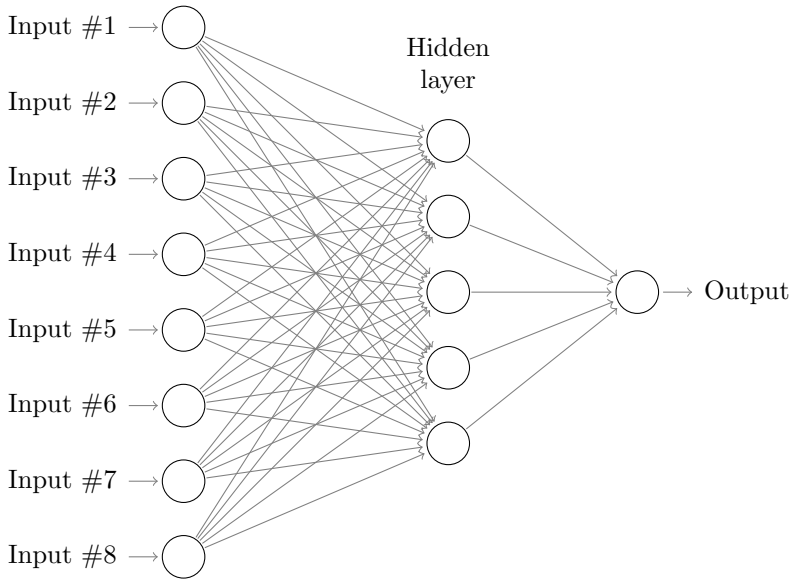


Figure 2.2: Multi-Layer Perceptron with an input, an output and a hidden layer

where *step* is the step function: $step(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ -1 & \text{otherwise} \end{cases}$.

The activation function being the step function, perceptrons are adapted to binary classification. But they can also be generalized to multiple class classification. Multiclass Logistic Regression (MLR) are such a generalization, and a suitable activation function is *softmax*.

However, the fact that perceptrons and MLRs are only comprised of one layer means that they can usually only be used for the simplest of tasks. More layers are required to carry out more complex tasks.

As their name states it, *Multi-Layer perceptrons* (MLPs) are perceptrons with several layers (see Figure 2.2). Layers which are neither the input nor the output layers are called *hidden* layers. In theory, one hidden layer suffices to solve any problem with enough neurons in the said layer [74, 150]. This result also depends on the NNs' hyperparameters. For instance, the choice of the activation functions affects the expressivity of a certain NN. Thus, some functions can be represented by deep *ReLU* networks but not shallow ones [35]. In practice, however, the number of neurons in the hidden layer – and the resulting number of weights – would have to be too large for real-life applications. This is why MLPs usually need numerous layers for complex tasks. Unfortunately, each fully connected layer introduces a high number of new parameters, but also redundancy, hence the need for new types of layers in Deep Learning (DL) architectures.

2.2.3 Convolutional Neural Networks

Despite their simplicity and expressivity, MLPs are often inefficient compared to NN models with sparsely connected layers, such as Convolutional Neural Networks (CNNs). The latter, as their name implies, mainly rely on convolutions in their layers to compute the predictions.

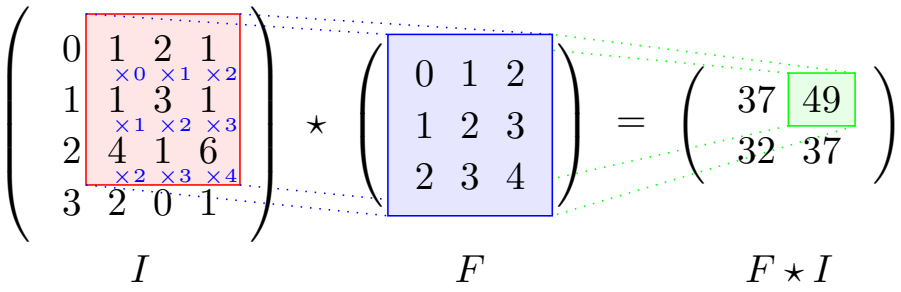


Figure 2.3: Convolution between an input I of shape (4×4) and a filter F of shape (3×3).

Convolutional layers Convolutional layers compute a convolution between their input and one or several three dimensional tensors named *filters*. Let us first consider the two dimensional case. For an input I of shape $n \times m$ and a filter F with shape $w \times h$, the convolution between I and F is defined as:

$$(F \star I)_{i,j} = \sum_{k=1}^{h} \sum_{l=1}^{w} I_{i+k,j+l} \cdot F_{k,l}$$

Figure 2.3 gives an example of a convolution between a 2-dimensional input of shape (4×4) and one 2-dimensional filter F of shape (3×3).

Here, the output has shape $(n - h + 1) \times (m - w + 1)$. This definition is valid for a stride of one: the filter 'moves' one step to the right after each multiplication. With a stride s along the height and s' along the width, the definition becomes:

$$(F \star I)_{i,j} = \sum_{k=1}^{h} \sum_{l=1}^{w} I_{i \cdot s+k,j \cdot s'+l} \cdot F_{k,l}$$

and the output has shape $(\lfloor \frac{(n-h)}{s} \rfloor + 1) \times (\lfloor \frac{(m-w)}{s'} \rfloor + 1)$, where $\lfloor \cdot \rfloor$ denotes the floor function. This can be generalized to three dimensional inputs and outputs. In this case, the first dimension is called the *channel*, while the other two are called the input *height* and *width* respectively. Let us note that this leads to a four-dimensional filter. Thus, given input I of shape $(c \times n \times m)$ and filter F of shape $(c \times h \times w \times c')$, a convolutional layer returns an output C of shape $((\lfloor \frac{(n-h)}{s} \rfloor + 1) \times (\lfloor \frac{(m-w)}{s'} \rfloor + 1) \times c')$. A neuron $C_{ch,i,j}$ is computed as:

$$C_{ch,i,j} = \sum_{ch'=1}^{c} \sum_{k=1}^{h} \sum_{l=1}^{w} I_{ch',i \cdot s+k,j \cdot s'+l} \cdot F_{ch,k,l}$$

As in the fully connected case, a bias is then added to the output neurons, but each channel has the same bias value.

Padding is another important hyperparameter of convolutional layers. Convolutional layers often zero-pad their inputs: rows and columns of zeros are added to the top and bottom, as well as the right and left of the input matrices. The reason for this padding operation is threefold [67]:

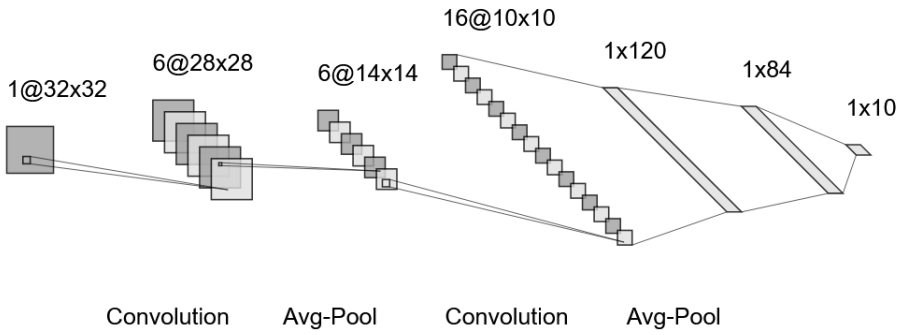


Figure 2.4: LeNet architecture. The last three layers are fully connected ones. For each layer, the input shape is indicated above it in the form $channel\_number \times input\_height \times input\_width$

- To avoid the *border effect*. Due to its definition, corner and edge input neurons occur in fewer convolutional operations than the ones in the center. Padding increases the impact of those disadvantaged neurons on the layer operations.
- Each convolutional layer returns an output with smaller dimensions than its input. A chain of such layers might therefore lead to too much information loss. Padding can be used to keep the input and output dimensions equal.
- The stride and filter size may not fit the input shape: some rows or columns may not be considered at all in the multiplications. In this case, padding can fit the input size to the layer's hyperparameters.

Pooling layers – as described in Section 2.2.1 – are a part of most common CNNs, as they usually follow a group of convolutional layers. The goal of this downsampling is to avoid overfitting on the training data. Indeed, only considering the most characteristic elements of the inputs prevents the model from learning the details in the dataset which are irrelevant to the predictions. This, in turn, enables the model to generalize and make correct predictions on inputs that are not part of the initial dataset. Dropout layers, which deactivate nonessential neurons, have the same aim.

Common CNN architectures The first CNN was introduced in 1998 by LeCun et al. [72]. It is comprised of two convolutional layers followed by an average pooling layer each, and end with three fully connected layers. It was first introduced as a classifier for a hand-written digits dataset, MNIST (Modified National Institute of Standards and Technology database) [28], and it reaches a 98.94% accuracy on it. The architecture is described in Figure 2.4. Since, the average pooling operation has been replaced by max pooling.

However, for more elaborate databases such as CIFAR10 [71], LeNet's accuracy is around 73%, which is much lower than state-of-the-art accuracy for this dataset: 99.5%. CIFAR10 is comprised of 60,000 images of shape $32 \times 32 \times 3$ (50,000 in

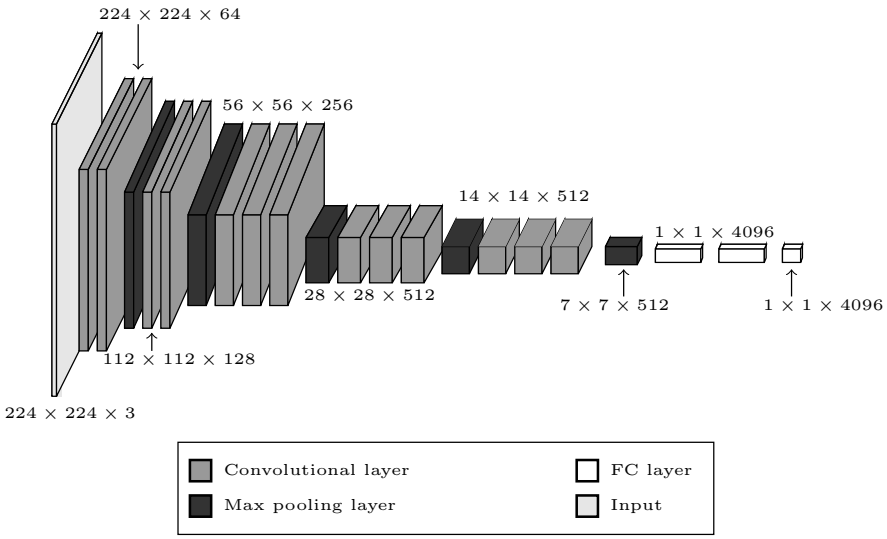


Figure 2.5: VGG16 architecture. VGG16 has 13 convolutional layers and 3 FC ones. The input shape of each block of layers is indicated on top of it, in the format: $input\_width \times input\_height \times number\_of\_channels$

the training dataset and 10,000 in the testing one), classified in 10 different classes. Another version of CIFAR includes images from 100 classes.

Since the LeNet architecture was detailed in [72], CNNs have been surging, especially when it comes to image processing. One way of improving the training of CNNs is to increase their depth and number of neurons. The depth and neurons increase their probability of identifying characteristic features. For instance, the VGG16 architecture has 13 convolutional layers, 3 fully-connected ones and 138 million parameters, as can be seen in Figure 2.5.

VGG16 reaches an accuracy of 93.56% on the CIFAR10 testing set. The authors of [115] have expanded their architecture to create VGG19, which contains 3 more convolutional layers. Although MNIST, and especially CIFAR10 can still be used in papers for early stage research or proof-of-concept, they are generally considered to be not representative enough of a model's efficiency. ImageNet Large Scale Visual Recognition Challenge (ILSVRC) – often simply called ImageNet, on the other hand, is present in most image processing papers, as it contains over 1,400,000 images divided in 1,000 classes. On ImageNet, VGG16 achieves an accuracy of 71.3%, when the state-of-the-art accuracy is over 90%. VGG19 reaches a slightly higher accuracy: 75.2%.

So far, we have explained that the deeper the network, the better the accuracy. However, this is only true up to a certain point. Indeed, very deep NNs encounter a certain issue during training: the *vanishing gradient* problem.

Most Deep Neural Network (DNN)'s training is based on backpropagation, in which gradients are computed using the chain rule, as explained in Section 2.2.1. For deep networks, gradients end up being too small after a certain point: they 'vanish'. Since parameters are updated by adding a value proportional to the computed gradient, the vanishing gradients prevent any further learning. The authors of [48] introduce ResNets (Residual Networks) in order to mitigate this effect. For

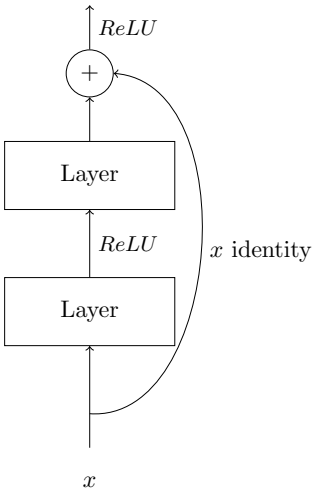


Figure 2.6: Skip connection. The output of the block is $f(x) + x$ where x is the block's input and f gives the output of the two layers at hand.

this, they consider *residual blocks*, which incorporate *identity shortcuts* – also called *skipping connections* (see Figure 2.6). In those shortcuts, earlier layers are added to deeper ones, in order to reintroduce information which would have otherwise been lost because of the vanishing gradients.

Moreover, the authors argue that the additional shortcuts enable the model to train efficiently without changing the actual training. Thanks to this new block, the ResNet architectures can be much deeper and more accurate than the VGG family. ResNet152V2, which has 152 layers, achieves a 78% accuracy on ImageNet.

2.2.4 Software Implementation of NNs

Various ML libraries enable a fast and efficient implementation of ML models. Among them are Tensorflow [83], PyTorch, Caffe [63], Theano [123] and Keras [22] (which can run on top of Tensorflow or Theano for instance). Even though their main interface is generally Python, many of them, such as Tensorflow and PyTorch, are partially based on C, C++ or CUDA for efficiency. Those libraries are heavily optimized to ensure fast computations on very large NNs.

Convolutional Implementation One of the optimization tricks was introduced by Caffe's author in 2014 [63]. If we omit the bias, then it is clear that FC layers can be written as matrix multiplications. Indeed, given an input I with n neurons, and a weight matrix W of shape (n, m), then the output has m neurons and is computed as:

$$O = I \cdot W$$

This enables the use of efficient and, once again, heavily optimized matrix multiplication algorithms, for instance GeMM [45] in the OpenBLAS library.

But in Caffe, convolutional layers are also computed as matrix multiplications. To do so, each convolutional window in the input I is flattened to form a row in the reformed input matrix I'. Each filter in F is also flattened to form a column in

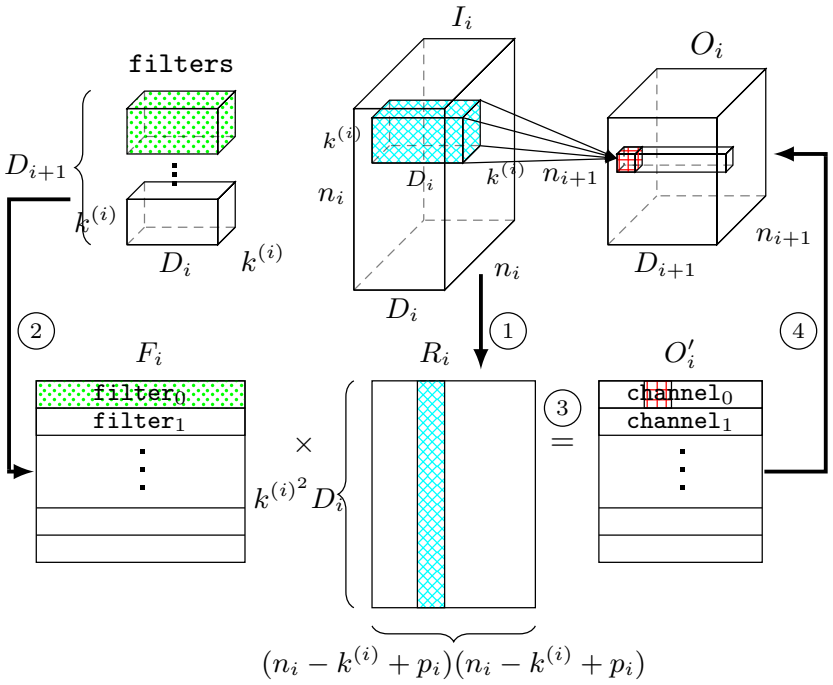


Figure 2.7: Process to turn a convolutional layer i into a matrix multiplication. Each filter and input channel is considered to be square in this depiction. The layer's filter F_i has shape $(D_i \times k^{(i)} \times k^{(i)} \times D_{i+1})$ – meaning there are D_i input channels and D_{i+1}) output ones. The reshaped filter F_i, has shape $(D_{i+1} \times (k^{(i)} \cdot k^{(i)} \cdot D_i))$. The input I has shape $(D_i \times n_i \times n_i)$, while its reshaped version F' has shape $(n_i - k^{(i)} + p_i)(n_i - k^{(i)} + p_i)$, where p_i is the layer's padding. Indeed, each column in I' corresponds to an input window on which a dot product with a filter window is carried out during the convolution. This leads to a reshaped output O' with shape $(D_{i+1} \times (n_i - k^{(i)} + p_i)(n_i - k^{(i)} + p_i)$. Each row in O' corresponds to an output channel.

the reshaped filter F'. Multiplying I' by F' yields a reshaped output O' where each column corresponds to one channel of the output O. This is described in Figure 2.7.

Despite the fact that the matrices involved can have extremely large dimensions – given the high number of neurons in a given layer –, this method has been observed to be more efficient in most cases. This is why several ML libraries such as Tensorflow have adopted this technique whenever possible – or whenever it does correspond to the most efficient implementation.

Common NN Setups The setup of an NN model matters when it comes to its security. For instance, some NN models' weights are protected during inference computations by being located in a secure environment such as SGX [85].

In this dissertation, we consider two main setups, which correspond to very common ones. In the first, the users have the model on their device. This means that they can have physical access to it – for example through probes.

The second setup we discuss is one where the model is hosted by a cloud provider.

This is the case for Machine Learning as a Service (MLaaS), such as Amazon SageMaker [3]. In this scenario, the user generally only has access to the predictions, and cannot easily – or physically – access the details of the model.

2.3 Attacks against Neural Networks

For years, NNs have been the victim of various attacks, targeting several parts of the model. NNs can either be targeted during the training phase or during runtime – also called the inference. Attack surfaces exist in both phases:

- During training:
  - The training dataset
  - The training algorithm
- During the inference:
  - The architecture
  - The testing set (or inputs)
  - The trained parameters

All the aforementioned parts can be targeted by a potential attacker, in a variety of settings. A malicious user who has access to the training set can poison it by adding crafted or mislabeled inputs, in order to change the model's predictions. This is one type of poisoning attacks. More generally, attackers using a poisoning attack wish to affect the model's predictions by targeting the training process. Besides injecting wrong data, they can also change the already existing training input or directly manipulate the algorithm itself – by changing its parameters, for example [130].

Another breach of security comes from membership inference attacks, where a malicious user can determine whether a certain input belongs to the training set [113, 112]. Thus, attackers can both change the training process and leak information about the training dataset.

But the runtime phase – also called inference –, along with the optimal architecture and parameters, are not safe either.

Like poisoning, adversarial attacks also aim at modifying the target model's input, but without requiring access to the training phase [43, 91, 80, 19, 5]. In adversarial attacks, the user crafts adversarial inputs by adding a small noise, undetectable to a certain oracle such as the human eye, but which leads the model to return the wrong predictions. This could lead to impersonation attacks, where an attacker's face can fool a model into accepting their face as the owner's one.

Model inversion attacks aim at recovering the input in a context where it is encrypted for privacy reasons [124, 136]. For instance, the input can be loaded in a secure component such as Intel SGX, and the attacker can retrieve the hidden input using another ML model trained on timing or power traces. Similarly, secret output can also be recovered.

Finally, multiple attack papers target the parameters and hyperparameters themselves. These are called reverse-engineering attacks. They can be of various types:

- Based on equation-solving or ML [125, 93, 126]: Tramèr et al. [125] train a model with the same architecture as the original one to retrieve the correct weights. Their aim is to solve the system of equations formed by $f(x) = y$ where x is an input selected at random, f is the original model and y is the model's predictions. Solving the given system can be achieved by minimizing a loss function through an optimization process, just like training. However, here, the loss function measures the difference between the extracted model's predictions and the original model's predictions rather than the true outputs. The authors of [94] also aim at training a substitute model to recover the predictive behavior of the target model. But instead of querying with random inputs, they carefully select the set of queries. Other such attacks have since then managed to make the recovery more efficient using much fewer queries [126, 66, 89].

- Mathematical attacks, which rely on the internal structure of the NN models, as described in Section 2.3.2.

- Side-channel based. Side-channel attacks rely on leakages due to the implementation of an algorithm to extract secret information. Attack vectors include the cache [141, 53, 52, 79], power or electromagnetic emanations [10, 146, 55, 136, 15, 137], memory access patterns [56] and running time [32, 82]. Some of these attacks are detailed in Section 2.3.3.

All these attacks require different threat models, depending on the attacker's needs and capabilities. We differentiate between white-box, gray-box and black-box attacks. In white-box attacks, the attacker has access to the entire NN model and knows about potential defenses. In the black-box one, the attacker has no prior information about the model and defenses. The gray-box context assumes an attacker who has partial knowledge about either the model or the applied defenses.

In this thesis, we will mainly focus on mathematical and side-channel-based reverse-engineering attacks. However, one of our proposals can also be applied to adversarial attacks (see Chapter 5).

2.3.1 Introduction to Reverse-Engineering Attacks

Threat Scenarios First of all, let us note that reverse-engineering attacks can have various threat scenarios, shown in Table 2.1.

Performance Measurement In a reverse-engineering attack the potential attacker wishes to extract a model $\hat{f}$ as close as possible to the original model f. The authors of [61] consider three ways of measuring how 'close' the two models are.

- *Functionally Equivalent Extraction*: In this case, the attacker wishes $\hat{f}$ to be such that $\hat{f}(x) = f(x) \forall x \in \mathcal{I}$ where $\mathcal{I}$ is the input set. This means that the attacker wishes to find the original model's parameters so as to return the exact same values on all inputs.

Table 2.1: Threat scenarios in reverse-engineering attacks. The attacker has query access to the model. She can then target the architecture, the parameters or both. When an element is the target, partial information about it can be assumed to be public.

Scenario	Architecture	Parameters
Scenario 1: Architecture Extraction	Target	Unknown
Scenario 2: Parameter Extraction	Known	Target
Scenario 3: Model Extraction	Target	Target

- *Fidelity Extraction*: Here, $\hat{f}$ should be such that for some similarity function S, $Pr[S(\hat{f}(x), f(x))]$ is maximal. In other words, the attacker wants to find a model $\hat{f}$ whose predictions are similar to f with a high probability. For instance, in the classification case, S can be label agreement. This means that for all inputs x, there is a high probability that $\hat{f}$ classifies x in the same class as f would.

- *Task Accuracy Extraction*: In this type of extraction, the attacker wishes to find a model that achieves the highest possible accuracy on a given task, and only uses f as a tool. She therefore aims at maximizing $Pr[\hat{f}(x) = y(x)]$ for all inputs, where $y(x)$ is x's the correct prediction.

In this dissertation, we mainly focus on Functionally Equivalent Extraction.

2.3.2 Mathematical Reverse-Engineering Attacks

Let us start by detailing the reverse-engineering attacks that rely on a model's internal structure in order to recover its parameters [61, 87, 103, 16, 27].

Threat Scenario Mathematical attacks correspond to the parameter extraction scenario (see Table 2.1). The attacker's goal is functionally equivalent extraction, as defined in Section 2.3.1. In [103, 61, 87], the following assumptions are necessary:

- The model is assumed to be piece-wise linear.

- The model is comprised of linear layers (such as FC ones) with *ReLU* activation functions.

- The attacker has query access to the victim model: they can provide crafted inputs to the model at hand. They receive the last layer's full outputs, where each neuron has a value in $\mathbb{R}$.

Although the authors of [87, 27] and [61] assume the architecture to be known, the authors of [103] assume an attacker who does not know the number of neurons per layer, and aims at recovering that secret along with the parameters.

While the authors of [16] also consider an attacker who already knows the model's architecture, they claim that the only fundamental requirement for their attack is the piece-wise linearity of the model.

This attack model can correspond to the case of online services, where the attacker is a user who can send queries and receive the output without accessing the model's parameters.

Attacks Description The authors of [16] reach their goal with fewer queries than [61, 87, 103]. The authors of [27] ensure a polynomial time exact extraction of NNs with up to three layers. However, to achieve this, they make assumptions not required in [16]. The most sensitive assumption is that the second layer's dimension must be higher than the first. This is not always the case, as in most models recovered in [16]. Moreover, even though [87] describes a strategy that applies to arbitrarily deep NNs, [16] is the only paper that proves the practicability of their attack on models with more than three layers. For these reasons, we focus on the method described in [16]. However, all three attacks follow the same pattern.

Let us start with two definitions. Let $\mathcal{V}(\eta, x)$ denote the input of neuron η, before applying the *ReLU* activation function, when the model's input is x. For a given neuron η at layer l, its critical point is defined as follows:

Definition 1. *When, for an input x, $\mathcal{V}(\eta, x) = 0$, the neuron η is said to be at a critical point. Moreover, x is called a witness of η being at a critical point.*

Finding at least one witness for a neuron η enables the attacker to compute η's critical hyperplane.

Definition 2. *A bent critical hyperplane for a neuron η is the piece-wise linear boundary $\mathcal{B}$ such that $\mathcal{V}(\eta, x) = 0$ for all $x \in \mathcal{B}$.*

Given those definitions, the parameter extraction protocol is:

1. Identify critical points and deduce the critical hyperplanes
2. Filter out critical points from later layers
3. Deduce the weights up to a multiplicative factor
4. Find the weight signs

Even though the way critical planes and hyperplanes are found differs from one paper to the next, all the attacks are based on the existence of a hyperplane for each neuron, created by the activation function. The hyperplanes' equations are the basis for the parameter recovery.

[16] covers several cases: the authors start by considering an NN with only one layer, before moving on to contractive deep layers and finally generalizing it to any deep NN. Let us note that all layers are considered to be FC. We also only consider *ReLU* activation functions for simplicity.

With one layer, the output is $f(x) = A^{(1)} \cdot x + b^{(1)}$ where A is the weight matrix and b is the vector of biases. Then $A_i^{(1)} = f(x + e_i) - f(x)$, where $e_i = (0, \ldots, 0, 1, 0, \ldots)$, and the 1 is at the i-th position.

New problems arise as soon as activation functions intervene, as the NN is no longer linear. With *ReLU*, negative neurons are set to 0. Let us now consider a

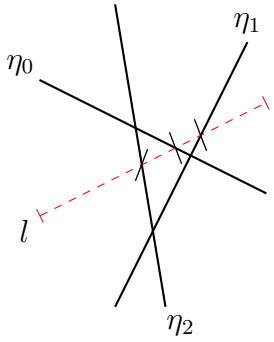


Figure 2.8: Hyperplanes for three neurons in the first layer. The dashed red line l enables the attacker to find the critical points indicated by the slashes.

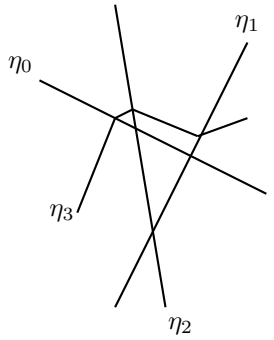


Figure 2.9: Hyperplanes are bent by boundaries from previous layers. For instance, η_3's hyperplane on the second layer is bent by the hyperplanes of η_0, η_1 and η_2 on the first layer.

2-layered NN: $f(x) = A^{(2)} \cdot ReLU(A^{(1)} \cdot x + b^{(1)}) + b^{(2)}$. The first step is then to find critical points. To achieve this, the authors of [16] suggest selecting a random line for each neuron in the input space. For each line – and therefore neuron, the attacker selects an initial point x_0 then carries out a binary search to find nonlinearities on $f(x_0+tv)$ for $t \in [-T, T]$ by checking for changes in the partial derivatives. Although the attacker does not have direct access to these derivatives, they approximate them through finite differences. Figure 2.8 displays the binary search process.

This provides a list of critical points. Let $\{e_i\}_i$ be a basis for the input space and x^* a witness for neuron η_j. For $\epsilon > 0$ small enough, η_j is the only neuron being activated – or deactivated, if $A^{(1)}$'s columns are not colinear (the authors adapt the method to the general case). Since f is differentiable everywhere except on hyperplanes, the attacker can compute:

$$\alpha_+^i = \frac{\partial f(x)}{\partial e_i}\bigg|_{x=x^*+\epsilon \cdot e_i} \text{ and } \alpha_-^i = \frac{\partial f(x)}{\partial e_i}\bigg|_{x=x^*-\epsilon \cdot e_i}$$

Either $x^* + \epsilon \cdot e_i$ is such that $\eta_j = (A^{(2)} \cdot ReLU(A^{(1)} \cdot x + b^{(1)}) + b^{(2)})_j > 0$ and $x^* - \epsilon \cdot e_i$ is such that $\eta_j = 0$, or the other way around. This holds for any other

basis vector e_k. Thus, computing:

$$\frac{A^{(1)}_{j,k}}{A^{(1)}_{j,i}} = \frac{\alpha^i_+ - \alpha^i_+}{\alpha^1_+ - \alpha^1_+}$$

gives the first layer weights, up to a multiplicative scalar. In other words, computing double partial derivatives enables the attacker to extract the first layer's weights up to a multiplicative scalar. From those weights, the value of $b^{(1)}$ up to a multiplicative scalar easily follows.

Once the first layer is recovered, getting the second layer's weights results from applying the method for a 1-layer NN.

For deeper layers, a new difficulty arises: even though the attacker can still find partial derivatives by looking for nonlinearities, she cannot determine which neurons the critical points are witnesses for. Let us also note that hyperplanes in deep layers are impacted and bent (hence the name) by previous layers' hyperplanes (see Figure 2.9). For a layer l with d_l neurons, the authors of [16] solve this problem by considering $d_l + 1$ random directions δ and taking the second partial derivatives along those those directions, on all witnesses. Given the set:

$$\{y_i\} = \left\{ \frac{\partial^2 f(x)}{\partial \delta_1 \partial \delta_i}|_{x=x^*} \right\}_{i=1}^{i=d_l+1}$$

where x^* is a witness to a critical point. Generalising from the first layer recover, the attacker buids a system of equations $ReLU(f_{l-1}(x + \delta_i)) \cdot w = y_i$ where f_{l-1} is the model restricted to its first $l - 1$ layers. The equations corresponding to a neuron x in the current layer provide the weights as the result, while the others give completely different – and seemingly random – results that can be discarded.

Solving the system of equations also gives the weights. However, it requires being able to find a preimage of the network outputs, which is not always the case. In fact, in most interesting cases, NNs are not contractive – i.e. some of their layers have higher output dimension than input dimension – and their preimage can therefore not be easily found. In those cases, a brute force attack on the sign is necessary.

Finally, the authors of [16] provide further details on how to implement this attack efficiently.

2.3.3 Side-Channel Attacks: Generalities

Side-Channel Attacks (SCAs) have been introduced in the 1990's. They take advantage of unintended leakages proceeding from the implementation of cryptographic algorithms. Processing the said leakages may involve statistical analysis or ML models.

Access Abilities Different SCAs can be carried out depending on the attacker's access abilities. We differentiate between:

- *A local access* where the attacker is located on the same machine as the target. The attacker and the victim therefore share some resources, such as the cache.

- *A remote access* where the attacker can only access the victim algorithm remotely. For instance, the victim may run on the local network. Among other possibilities, this setting enables timing attacks.

- *A physical access* where the attacker has access to the physical circuit, such as a processor or smartcard. This type of setting is necessary for power or electromagnetic-based SCAs for example.

Existing Attacks In recent years, NNs have been the target of numerous SCAs [141, 146, 55, 10, 79, 136, 124, 137, 52, 153, 53, 32]. We published a survey [158] focusing on cache-based SCAs, but presenting all SCAs at the time of publication, to the best of our knowledge. Table 2.2 and Table 2.3 summarize the characteristics of the existing architecture extraction attacks. For each architecture extraction attack, they provide the knowledge requirements for the attacker, the SCA type as well as the access type, the victim models on which the attack was tested, the results and the limitations of the attack. Among them are Cache Telepathy [141] and CSI Neural Networks [10] that we will detail in Section 2.3.4 and Section 2.3.5. To the best of our knowledge, the only memory access pattern attack for reverse-engineering an NN's architecture is [56]. The authors of [56] manage to recover a target model's architecture by monitoring memory access patterns and, more specifically, read-and-write (RAW) dependencies. They first differentiate between layers thanks to the sequential execution: one layer's output is the next layer's input. Layer boundaries therefore correspond to a memory write followed by a memory read on the same address. Similarly, they can identify most hyperparameters in a layer through the RAW dependencies. For instance, within a layer, the input neurons are only read, while output neurons are only written. This reduces the possible architectures to a very small set.

Parameter extraction SCAs are detailed in Table 2.4 in the same fashion. Xu et al. also published a survey on the security of NNs in the hardware context in 2021 [140].

Some model extraction attacks are out of scope in this dissertation. Despite their importance when it comes to correct predictions, we do not consider attackers whose only goal is to recover specific hyperparameters [129]. Indeed, we already provide a way to protect an architecture as a whole in Part II.

Moreover, new attack papers have been published since the survey's publication. Among them, some rely on queries [27] and/or are ML-based [89], others are time-based SCAs [82]. Physical leaks are also still exploited [144, 145]. While some attacks in Table 2.2, such as Cache Telepathy [141], only target CPU runs, others [53, 52] also include GPU runs. While [53, 52] are cache-based, recent attacks have managed to extract a partial architecture [21] or the entire model [15] through power and EM traces from the GPU. The authors of [135] carry out a timing SCA on GPUs to recover the architecture of a victim model during the training phase.

The Leaky Nets attack [82] is especially interesting, as it is the only one that manages to recover parameters exploiting mainly timing patterns. The authors also use Simple Power Analysis (SPA), but only to identify the interesting parts of an power consumption acquisition. Indeed, an attacker targets the timing behaviour of

Table 2.2
List of cache-based side-channel architecture extraction attacks.

Attack	Assumptions	Access type and SCA	Victims tested	Results	Limitations
Cache attacks					
Cache Telepathy [141]	Co-location, page sharing, use of GeMM, ML framework known	Local, Cache	VGG, ResNet	Set of possible architectures (including hyperparameter dimensions)	Not applicable to GPUs; the set can be too large if the architecture is complex
DeepRecon [53]	Co-location, known family of architectures, page sharing, ML framework known	Local, Cache	VGG, ResNet, DenseNet, InceptionV3, InceptionResNet, Xception, MobileNet	Number of layers, layer types, and approximation of the hyperparameter dimensions	Cannot recover novel architectures, since it relies on the knowledge of the family of NNs
How to 0wn NAS in Your Spare Time [52]	Co-location, page sharing, ML framework known, some knowledge about how NNs are constructed	Local, Cache	MalConv, ProxylessNAS	Full computational graph	
GANRED [79]	Set of possible architectures, ML framework, shared LLC	Local, Cache	AlexNet, VGGNet	Full architecture	The attacker tries all possible hyperparameters for each layer

individual multiplications, as well as of activation functions in order to fully recover the weights and biases of a given model.

The recent paper *Can One Hear the Shape of a Neural Network?* [15] show that despite the belief, so far, that SCA-based reverse-engineering attacks are impractical, very deep common architectures can be extracted through EM traces. This recent release proves NNs' weakness when faced with physical attacks.

In this dissertation, we mainly focus on two SCAs: Cache Telepathy [141] and CSI NN [10]. The first is a cache-based attack that aims at recovering a model's full architecture, while in the second, the attacker extracts the architecture and parameters simultaneously.

The authors of Cache Telepathy propose to use two SCA methods to monitor the cache: *Flush and Reload* and *Prime and Probe*.

Flush and Reload Flush and Reload [142] is a cache-based SCA used by multiple attacks targeting NNs [141, 79, 53, 52].

The goal of cache attacks is to determine whether a certain memory location has been accessed by the victim model within a certain period of time.

The cache is a small but fast memory space used to store the data that have been recently accessed in general purpose or bigger embedded processors. It is usually divided in three parts corresponding to three hierarchical levels: L1, which is close to the processor (CPU), L2, and LLC, which is the Last Level Cache. Usually data residing in L1 is also included in L2, which in turn is included in LLC. When a process tries to access an address, the CPU first looks for it in L1. If it is not there, then it looks in L2. If it still cannot find it, it goes on to search the LLC. Finally, if the address is not in the cache, it needs to access the much larger main memory.

Table 2.3
List of other side-channel architecture extraction attacks.

Attack	Assumptions	Access type and SCA	Victims tested	Results	Limitations	Number of traces
EM attacks						
DeepEM [146]	Known set of architectures, physical access	Physical, EM emanations	ConvNet, VGGNet	Recovers layer boundaries, depth and layer types. This leads to a set of possible architectures for the victim NN. An ML approach recovers the weights.	Several hyperparameters cannot be determined with EM emanations only. Training several architectures is still necessary, even if accelerated thanks to adversarial samples	10,000
DeepSniffer [55]	Only universal knowledge about how NNs are constructed	Physical, EM emanations	AlexNet, VGG, Resnet, Nasnet_large	Full computational graph		1
Can one hear the Shape of NNs? [15]	No prior knowledge	Physical, EM and ML	AlexNet, VGGNet, ResNet101	Full computational graph	Does not work on small batches, requires a training dataset, does not track dataflow	Not specified
Other attacks						
Remote timing side-channel attack [32]	Distribution of the training dataset, precise execution time measurements	Local/ Remote, Timing	LeNet, AlexNet, VGG, ResNet	Number of layers, their type and the hyperparameter dimensions	Requires precise measurements; the distribution of the training set can be private	20
Memory Access Pattern Attack [56]	Known family of architectures, ability to monitor all RAW memory pattern accesses	Local, Memory access pattern	AlexNet, SqueezeNet, LeNet, ConvNet	Smaller set of possible architectures (including hyperparameter dimensions)	Activation functions cannot be determined through this attack. The deduced set can be too large if the architecture is complex	Not specified

If a process successfully retrieves data stored in the cache, this is called a cache hit. On the other hand, if the address is not in the cache – and is therefore stored in the main memory – it is a cache miss.

Given the way the cache works, the execution time of a memory access depends on whether the target address is in the cache or not. Cache attacks are based on this induced time difference.

Let us note that while L1 and L2 caches are usually only shared among processes within the same core, the LLC is shared by all processes, independently of the core. Moreover, the LLC is often divided in sets, called cache sets.

The Flush and Reload attack consists of the three following steps:

1. Flush cache line – which contains the data – corresponding to the target address (*flush*), such that the data is evicted from the cache and can only be found in the main memory.

2. Wait for a certain period of time, leaving the victim enough time to access the target memory location.

Table 2.4: Parameter Extraction Attacks

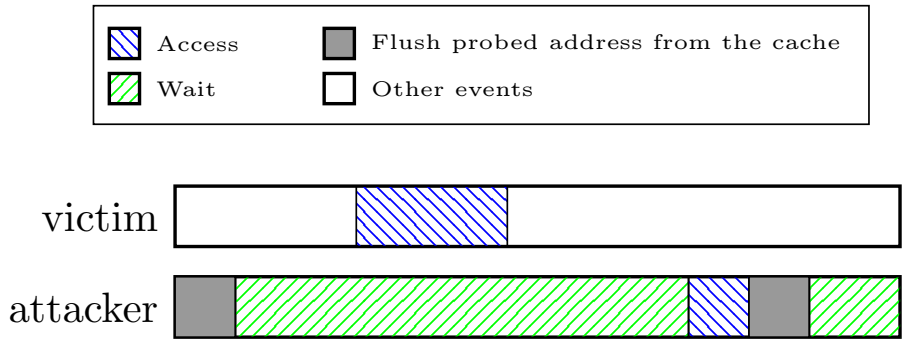
Attack	Assumptions	Side channel	Victims tested	Results	Limitations
Memory Access Pattern Attack [56]	Known architecture. Ability to query the model. Weight pruning occurs.	Memory Access Patterns	AlexNet, SqueezeNet, ConvNet	Weights as a function of the bias. If the pruning threshold can be controlled, the bias can also be recovered.	Requires the ability to control the input. The weights cannot be recovered exactly if the pruning threshold cannot be controlled.
Open DNN Box [137]	Known set of pre-trained models. Redundant weights are set to 0.	Power	AlexNet, ResNet, Inception, MobileNet	Architecture and sparsity, thus identifying the correct weights among the pretrained ones.	The assumption that the model and weights are among a set of pretrained ones is quite strong.
CSI NN [10]	Ability to collect EM emissions and power traces. The considered models are FCNs.	Power and EM	FCN, CNN	Number of layers, number of neurons per layer and precise weights.	The attack has only been tested on FCNs with few layers.
Leaky Nets [82]	Query access, ability to gather power traces	Timing and SPA	CNN	Weights and biases	Getting the timing of individual NN operations is possible on embedded microcontrollers, but more difficult in custom-designed hardware units such as FPGA

3. Access the target address and measure the access time (*reload*).

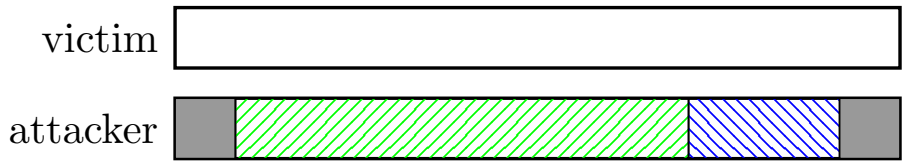
If the access time is short (i.e. below a threshold to determine), then the address is in the cache. This means that the victim has called the corresponding function (see Fig. 2.10a). A high access time, on the other hand, means that the CPU had to load the address from the memory. Thus, the victim has not accessed it between the flush and the reload (see Fig. 2.10b).

Flush and Reload works on the LLC, which is shared by all processes even from different cores. However, each process has its own virtual address space. It can therefore have different virtual addresses for the same function. Thus, this attack requires page sharing: when two processes use the same read-only memory pages, these pages are shared. This is implemented in many systems for efficiency purposes, in particular for shared libraries. Moreover, to flush a memory line from the cache, one needs the `clflush` x86 instruction or equivalent.

Prime and Probe In fact, Flush and Reload is a variant of a previous attack, Prime and Probe [95], which is slower but does not rely on any particular shared memory requirements. Once again, the attacker targets the LLC. Since there is no page sharing, the attacker does not know which cache location the target instruction



(a) The attacker flushes the probed address from the cache. Then, the victim accesses, putting the probed address back in the cache. The attacker accesses again, and the access time is reduced.



(b) The attacker flushes the probed address from the cache. The victim does not access the address. The attacker accesses the address, and the access time is high.

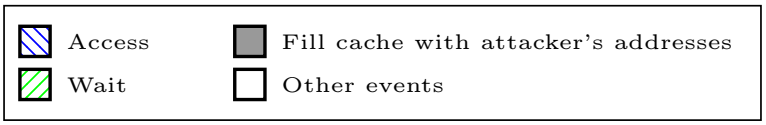
Figure 2.10: Access times in the Flush and Reload attack, with (a) and without (b) a victim access to the probed address. The x-axis is the time elapsed since the beginning of the attack.

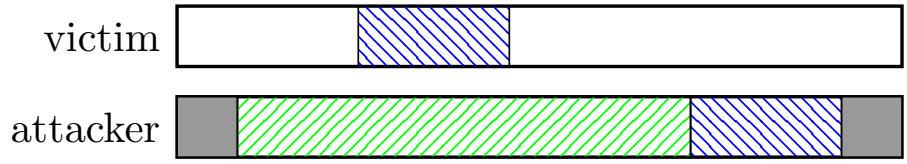
is mapped to. The first step of this attack is therefore to find the correct cache set – a set of cache lines. The details of this step can be found in [95]. Once the correct cache set has been recovered, the attacker can find out whether the victim accessed a certain address as follows:

- Find the target cache set.
- Fill the cache set with the attacker's data, thus evicting the target from the cache.
- Wait for a certain period of time, leaving the victim enough time to call the target address.
- Access the attacker's data again.

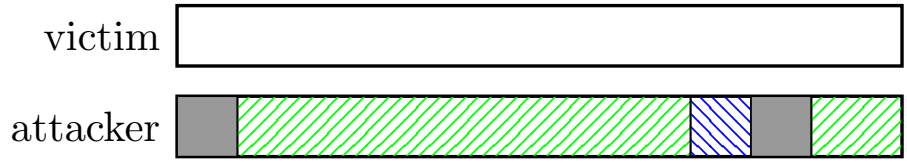
If the access time is short, then the attacker's data has not been evicted. This means that the target has not been loaded back into cache memory (see Fig. 2.11b). A high access time means that the attacker's data has been evicted from the cache, showing that the victim has accessed the target memory line (see Fig. 2.11a).

Even though Prime and Probe is more noisy – and therefore leads to less accurate results –, both can be used to carry out architecture extraction attacks, as is shown in Section 2.3.4. The main advantage of Prime and Probe is that it does not require page sharing, contrary to Flush and Reload. The authors of [53] and [52], however, give examples of how to achieve page sharing to launch an architecture extraction attack using Flush and Reload.





(a) The attacker fills the cache with her addresses, evicting the monitored address. The victim accesses the monitored address, evicting one of the attacker's addresses. The attacker accesses her addresses again, and the access time is higher.



(b) The attacker fills the cache with her addresses, evicting the monitored address. The victim does not access the monitored address. The attacker's addresses are still in the cache. The attacker accesses her addresses again, and the access time is reduced.

Figure 2.11: Access times in the Prime and Probe attack, with (a) and without (b) a victim access to the probed address. The x-axis is the time elapsed since the beginning of the attack.

2.3.4 Cache Telepathy: A Cache-Based Side-Channel Attack

Cache Telepathy [141] exploits a leak in matrix multiplications to recover most hyperparameters. Most ML frameworks, such as Tensorflow, use an efficient matrix multiplication algorithm, such as GeMM [45], to implement FC and convolutional layers. In their paper, the authors of [141] test their attack against two such algorithms: GeMM and Intel MKL's matrix multiplication [131]. While the OpenBLAS library is open source, making GeMM's code easy to analyze, the details of the implementation remain more obscure in the Intel MKL case. For this reason, let us first detail the attack methodology based on the GeMM algorithm.

Threat Scenario Cache Telepathy [141] is an architecture extraction attack (see Table 2.1) in an MLaaS context, and the quality of the extracted model can be measured through task accuracy. The goal is to retrieve a small set of possible architectures for the possible model, then train those architectures and select the one with highest accuracy. Even though this is a black box attack – in the sense that the parameters and hyperparameters are unknown to the attacker –, there still are some prerequisites:

- The attacker knows a large set of architectures the victim model belongs to, called the search space. For instance, the model could form part of the VGG (such as VGG16 or VGG19 [115]) or the ResNet (such as ResNet50 [48]) family.

- The attacker and victim processes are co-located: they share the same cache. Several papers have explained how to guarantee co-location in an MLaaS context [149, 102].

- If Flush and Reload is used, then page sharing is necessary.

The goal here is to reduce the search space to a tractable one – i.e. a search space that is small enough that an attacker can realistically train all models in it and select the one with the best accuracy.

Attack Description The authors of Cache Telepathy track calls to certain functions in Goto's implementation of GeMM [45] through the cache in order to deduce a victim model's architecture. The said algorithm divides the input matrices in smaller blocks so that the computations completely fill the cache. Let us consider the OpenBLAS library. In OpenBLAS, the algorithm makes use of three interesting functions: `itcopy`, `oncopy` and `kernel`. The pattern of appearance of these three functions is highly correlated with the dimensions of the matrices involved in the matrix multiplication. Thus, monitoring these three functions is enough for an attacker to recover the matrices' dimensions.

The Cache Telepathy attack consists in the following steps:

- Determine the number of layers thanks to the number of calls to GeMM.

- For each FC or convolutional layer, determine the input, filter and output dimensions by monitoring three of GeMM's internal functions: `itcopy`, `oncopy` and `kernel` using Flush and Reload or Prime and Probe.

- Determine the activation function thanks to Flush and Reload or Prime and Probe.

- Reduce the number of possible connections by measuring inter-GeMM latencies and using dimension constraints.

- Eliminate unfeasible stride and padding values, and reduce the possibilities for pooling and dropout layers thanks to dimension considerations.

Let us now explain the second step in this scheme. GeMM's implementation is detailed in Algorithm 1. We notice four different loops. They correspond to the subdivision of the input matrices so as to completely fill the cache, and therefore optimize the computation time. The said subdivision is shown in Figure 2.12.

Because the loops are here to implement a subdivision of the matrices into blocks, the number of iterations in the four loops is closely related to the matrices' dimensions. For two matrices of shapes $(m \times k)$ and $(k \times n)$, block sizes `P, Q, R` – selected

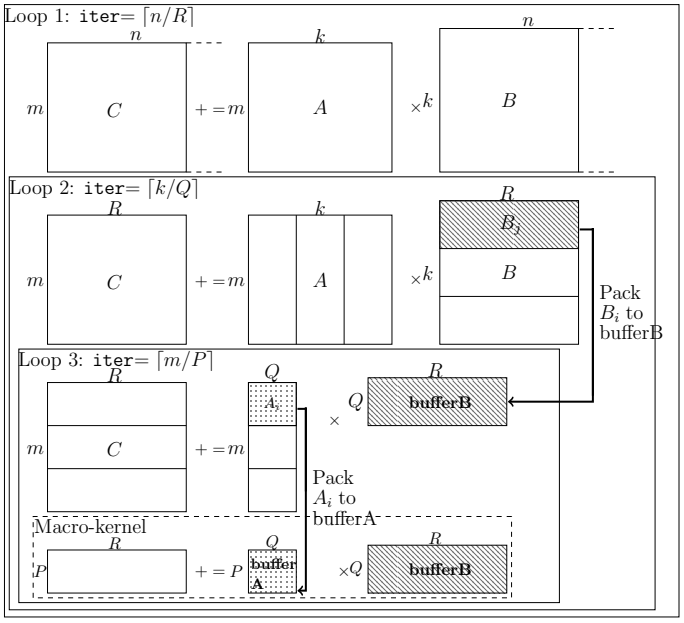


Figure 2.12: General Matrix Multiply (GeMM) algorithm implementation, which relies on machine-dependent block size to optimize matrix multiplications. The block sizes `P`, `Q` and `R` are selected depending on the user's machine, and more specifically its cache size.

based on the user's machine –, and constant `UNROLL`, the first observation is that:

$$\begin{aligned}
\texttt{iter4} &= \left\lceil \frac{R}{3\texttt{UNROLL}} \right\rceil \\
\text{or} \quad &= \left\lceil \frac{n \mod R}{3\texttt{UNROLL}} \right\rceil \\
\texttt{iter3} &= \left\lceil \frac{m}{\texttt{P}} \right\rceil \\
\texttt{iter2} &= \left\lceil \frac{k}{\texttt{Q}} \right\rceil \\
\texttt{iter1} &= \left\lceil \frac{n}{\texttt{R}} \right\rceil
\end{aligned} \tag{2.2}$$

Thus, determining the number of iterations in each loop for a given layer should give possible hyperparameters for the layer at hand. To do so, the authors of [141] monitor the three functions: `itcopy`, `oncopy` and `kernel`. If an attacker can closely monitor them and count their appearances, then they can deduce the number of iterations in the loops thanks to the observed pattern, detailed in Figure 2.13.

The scheme described enabled the authors of Cache Telepathy to reduce the set of possible architectures from 5.4×10^{12} to 16 for the VGG16 family, and from 6×10^{46} to 512 for the ResNet50 family.

Despite the fact that Intel's MKL library is not open source, the authors still manage to carry out their attack on it. Using binary analysis, they managed to identify a pattern similar to that in Figure 2.13. They can therefore apply the same technique to decrease the set of possible architectures. The attack on Intel MKL

Algorithm 1: Goto's GeMM algorithm in the OpenBlas library. It receives input matrices `A, B` and `C`, as well as scalar values α, β. It then computes α`A·B`$+\beta$`C` and stores the output in `C`. The block sizes correspond to matrix subdivision carried in the implementation for efficiency purposes (see Figure 2.12). buf_A and buf_B correspond to small buffers used to store temporary computation values.

```
   input : Matrices A, B, C; Scalars α,β; Integers (Block sizes) P, Q, R; Integer UNROLL
 1 for j ← 0 to n by R do
   |  ;                                                                    /* Loop 1 */
 2 |  for l ← 0 to k by Q do
   |  |  ;                                                                 /* Loop 2 */
   |  |  /* Loop 3, first iteration */
 3 |  |  itcopy(A[0, l], buf_A, P, Q);
 4 |  |  for jj ← j to j + R by 3UNROLL do
   |  |  |  ;                                                             /* Loop 4 */
 5 |  |  |  oncopy(B[l, jj], buf_B+(jj − j)× Q, Q, 3UNROLL); kernel(buf_A, buf_B+(jj − j)× Q,
   |  |  |    C[l, j], P, Q, 3UNROLL);
 6 |  |  endfor
 7 |  endfor
 8 |  for i ←P to m by P do
   |  |  /* Loop 3, rest of the iterations */
 9 |  |  itcopy(A[i, l], buf_A, P, Q); kernel(buf_A, buf_B, C[l, j], P, Q, 3UNROLL);
10 |  endfor
11 endfor
```

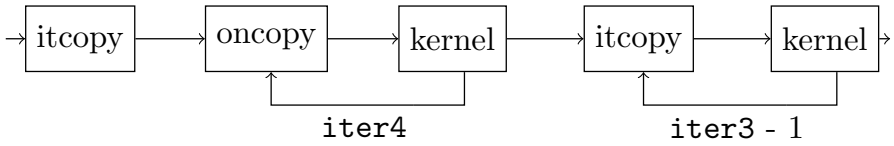


Figure 2.13: The functions `itcopy`, `oncopy` and `oncopy` form a pattern that is closely linked to the number of iterations in each loop from Goto's algorithm (see Algorithm 1).

leverages a reduced search space of 64 for VGG16 and 6144 for ResNet50. The results reported here correspond to the Flush and Reload case. Prime and Probe introduces more noise, and therefore leads to much larger reduced search spaces in the Intel MKL case. Even though the attack is less effective in the MKL and Prime and Probe cases, it still reduces the search space to a tractable one: it is then possible to train the models in the new space and select the one with the best accuracy.

2.3.5 CSI Neural Networks

Among the most powerful SCAs to date is a power or electromagnetic (EM)-based attack that successfully recovers the parameters and hyperparameters simultaneously [10]: CSI NN. The authors of [143] apply a similar approach to fully extract the architecture and secret parameters of Binary Neural Networks (BNNs). CSI NN is a model extraction attack (see Table 2.1) whose goal is functionally equivalent extraction as defined in Section 2.3.1.

Threat Scenario The attacker does not require any prior knowledge regarding the model. But she can still be considered strong, as she has a physical access to the processor running the target model. Experiments have been carried out on small FC and convolutional NNs.

Attack Description The authors of [10] statistically analyse power and/or EM traces to recover both the architecture and parameters of the victim NN. Here, we only describe the attack on FCNs.

Gathering EM and power traces can be achieved by placing a probe against the microprocessor of the attacked device [10]. EM traces can also be gathered from a distance. To do this, a software-defined radio may be used [14]. New methods relying on machine learning techniques allow the attacker to carry out side-channel analysis from a further distance (15m) [132].

The scheme in [10] relies on Correlation Power Analysis (CPA) [12] when power traces are used, and Correlation Electromagnetic Analysis (CEMA) when EM traces are gathered.

Let us start by detailing CPA. Given an algorithm requiring the use of a secret s:

- The attacker first isolates a part of the algorithm where she knows s – or part of s – intervenes. Let us denote by A the said subalgorithm.
- She gathers power traces T for inputs X to the subalgorithm A.
- She simulates the power consumption of A depending on s. This means that she finds a leakage model f such that $f(s)$ is the modeled power consumption for A. An example of such a leakage model f – used in [10] – is the Hamming weight of the output of A.
- For each guess k of the secret, she compares the simulated traces $f(k)$ for the inputs X with the gathered traces T. Let T_k denote the simulated traces for guess k. The comparison measure is the Pearson correlation coefficient ρ:

$$\rho(T, T_k) = \frac{cov(T, T_k)}{\sigma_T \cdot \sigma_{T_k}}$$

  where $cov(T, T')$ is the covariance between T and T', and σ_Y denotes the standard deviation of Y.
- The guess k^* associated with the simulated traces T_{k^*} with highest ρ is chosen as the correct guess for s. If the attack works, $k^* = s$

The only difference between CPA and CEMA is that instead of using power traces, CEMA is based on EM emanations.

Some secret key recoveries require a simpler attack. SPA (resp. SEMA), only uses one power (resp. EM) trace: in this case, the attacker can directly read the secret by observing one trace.

The authors of [10] proceed iteratively to recover the architecture and weights at the same time. The full scheme can be summarized as follows:

1. Recover the number of neurons through SPA (or SEMA).

2. Start at layer $l = 0$. For each neuron η determine whether η is in layer l or layer $l + 1$, as well as the weights by which it was multiplied thanks to CPA (or CEMA).

3. Determine layer l's activation function through time analysis. Let us note that later work [121] has established the possibility to recover activation functions through EM analysis as well.

4. If η is in $l + 1$, then $l \leftarrow l + 1$. The previous two steps are repeated until the attacker reaches the end of the gathered traces.

In the first step, the attacker uses one trace to differentiate between the various neurons, and therefore count the total number of neurons (see Figure 2.15). Neurons appear plainly because of the sequential execution of NNs. Indeed, each neuron is multiplied by all the weights it is associated to, before moving on to the next neuron. The activation function is also applied to each such neuron. On the other hand, it is much more difficult to differentiate between layers.

This is where the CPA comes into action, in the second step. In this second part, the attacker targets each individual multiplication $o = x \cdot w$ where x is a neuron and w is a weight. She then applies CPA (or CEMA) to the said multiplication, where w is the secret to be revealed. Thus, the attacker can use the Hamming weight of the output o to model the power (or EM) traces. The attacker gathers a set T of traces for $x \cdot w$ for various values of x, and computes traces T_w thanks to the leakage model for each possible weight value w. The weight w^* with modeled traces closest to the actual ones – i.e. which has highest Pearson coefficient $\rho(T, T_{w^*})$ – is taken to be the correct guess. This mechanism reveals the parameters associated to a given neuron when the architecture is known. But it does not recover a neuron's layer in the black-box case.

To overcome this issue, the authors of [10] suggest taking the layer into account in the CPA (resp. CEMA). The attacker knows the input I with n neurons. She starts with the n neurons from layer $l = 1$ and determines its weights through CPA or CEMA. She then moves on to layer $l = 2$ and proceeds layer by layer. For layer l and each neuron x, she knows x is either in layer l or in layer $l + 1$ because of the sequential execution. Thus, she applies CPA or CEMA on two hypotheses $(w, layer)$. This statistical analysis therefore gives her the two parameters at the same time. If x is in $l + 1$, she determines the activation function through a timing attack then moves on to the next layer: $l \leftarrow l + 1$. She repeats this operation until all neurons have been analyzed.

A time analysis is possible on the activation function because the various activations have a very different timing behaviour. Figure 2.14 shows the behaviour of the main activation functions depending on the input. The clear differences enable a potential attacker to determine each layer's activation once the weights are known, by trying various inputs and observing the resulting pattern.

Thus, the authors of [10] manage to fully recover small FCNs and convolutional NNs through a mix of power or EM-based SCA and timing analysis. In the case of an FCN, they recover the number of layers, the number of neurons per layer

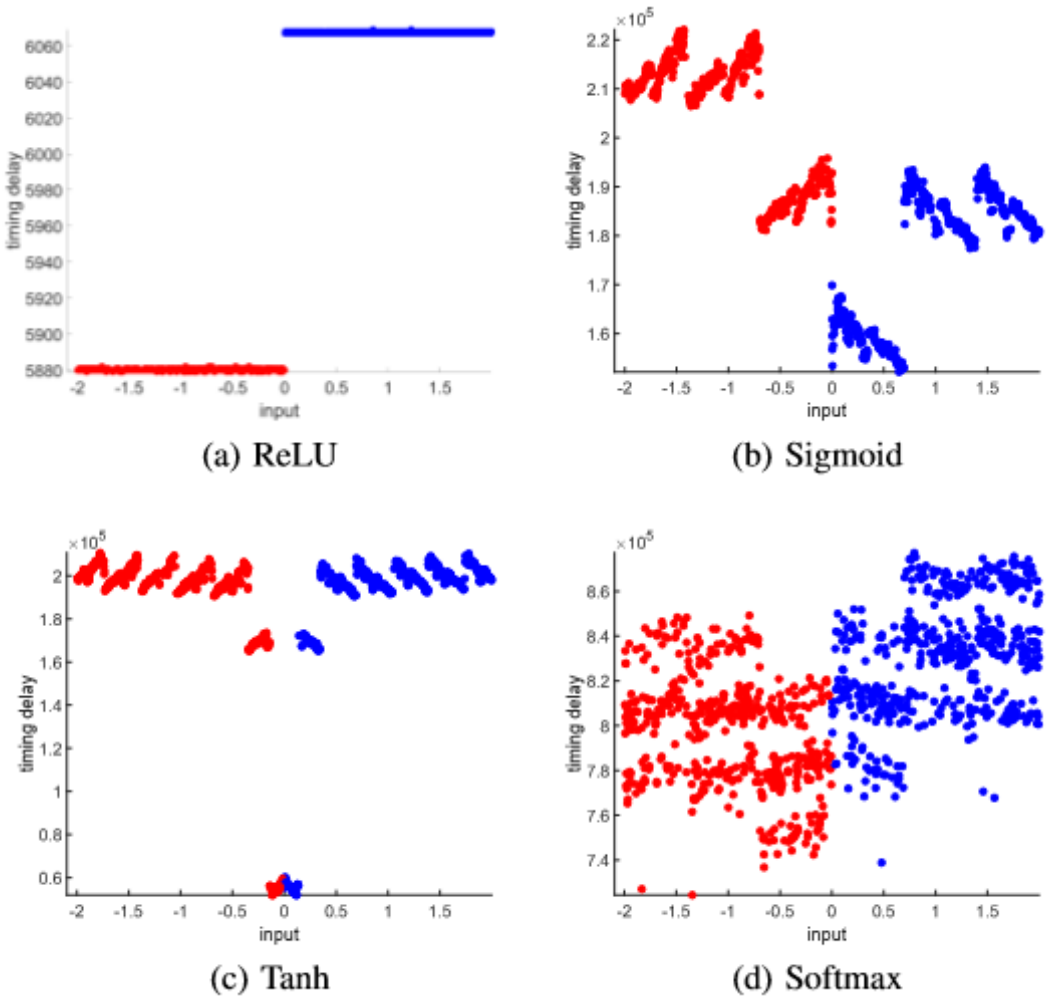


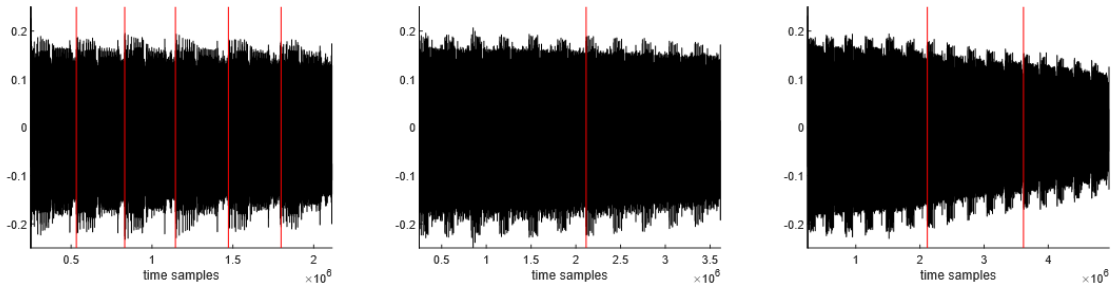
Figure 2.14: Timing behaviour for four common activation functions: (a) *ReLU*, (b) *Sigmoid*, (c) *Tanh* and (d) *Softmax* (see Section 2.2.1 for further details on the activation functions). Each graph represents the function's runtime depending on the input provided, for inputs in [-2, 2]. This image is taken from [10]

and the weights through an iterative process involving SPA (resp. SEMA) and CPA (resp. CEMA). The activation function recovery, on the other hand, relies on timing analysis. However, as mentioned, the authors of [121] have managed to launch the activation function attack using EM emanations. It is therefore possible to fully recover the architecture of models relying only on EM emanations.

In this Section, we have delved into the details of three among the numerous reverse-engineering attacks targeting NNs. We will tackle the described attacks in Chapters 3, 4 and 6 of this manuscript.

2.3.6 Adversarial Attacks

Reverse-engineering attacks have only recently become practical: most of them only target small NNs, only uncover part of the model, or they require prior knowledge. Adversarial attacks, on the other hand, can be applied to models of all sizes. While



(a) One hidden layer with 6 neurons (b) 2 hidden layers (6 and 5 neurons each) (c) 3 hidden layers (6,5,5 neurons each)

Fig. 9: SPA on hidden layers

Figure 2.15: SPA on an FCN: Observing one trace reveals the number of neurons. As is shown in (a), the neurons are easy to distinguish. However, as (b) and (c) show, differentiating between layers is much harder.

reverse-engineering attacks are quite recent, the first ones appearing in 2016, adversarial attacks have been discovered as early as 2013 [120] and have been heavily studied since.

Generalities Given a victim model f, an oracle O – often taken to be the human eye –, and an input x, the goal of an adversarial attack is to generate $x_{adv} = x + \eta$ where η is some small noise, such that the noise is undetectable to the oracle, but the target model's predictions are changed. In other words, $O(x) = O(x_{adv})$ – the oracle does not detect η – but $f(x) \neq f(x_{adv})$.

The notion of 'small noise' is relative, and it depends on the norm selected. Most often, the norms l_∞ or l_2 are taken. In the general case, the goal can be achieved by solving the following optimization problem:

$$\begin{aligned} x_{adv} = \min_{||\eta||}\{x + \eta \text{ s.t. } & f(x) = l \\ & \text{and } f(x+\eta) = l' \\ & \text{and } l \neq l' \\ & \text{and } x + \eta \in \mathcal{B}\} \end{aligned} \tag{2.3}$$

where $||\cdot||$ is the considered norm and $\mathcal{B}$ is a ball (for instance determined by $||x + \eta|| \leq 1$)

Various methods exist to achieve this optimal value, and they depend on the prior information an attacker has. In a *white-box* context, where the attacker knows the entire model, she can rely on gradients to find adversarial examples [91]. In a *black-box* context, where the attacker only gets the output of the model – either values corresponding to the last layer or labels corresponding to the classification –, she has to use approximations to reach the target [19]. For instance, one can find adversarial samples on a model similar to the victim and rely on transferability to attack the original model.

Adversarial examples can either be *targeted* or *untargeted*. If the first, then the class l' in Equation (2.3) is predetermined: the attacker does not simply aim at changing a model's predictions, but also at selecting the output class [7]. If the

second, then l' can be any label other than l [43, 91]. As an example, let us suppose that an input x's original label is 'cat'. In the untargeted case, the attacker is satisfied whether x''s label is 'tiger' or 'armchair'. In the targeted context, however, the attacker is only satisfied if the output is one specific class, say 'tiger'.

In this dissertation, we will only consider adversarial attacks in a white-box context. Let us note that an attacker's access to a model's gradient provides the best results in terms of adversarial examples. Three attacks in particular will be of interest to us.

Fast Gradient Sign Method The Fast Gradient Sign Method (FGSM) [43] was introduced by Goodfellow et al. in 2015 as a fast and simple way of generating untargeted adversarial examples. Given the model θ, input x with output y, a loss function J and a small perturbation ϵ, then the adversarial sample is computed as follows:

$$x_{adv} = x + \epsilon \times sign(\nabla_x J(\theta, x, y))$$

where $\nabla_x J(\theta, x, y)$ is the gradient of J relatively to x. The idea is to take a step in the direction of the gradient, as it enables the attacker to move away from the original class.

Projected Gradient Descent The Projected Gradient Descent algorithm (PGD) is an iterative algorithm in five main steps which consists in applying FGSM multiple times. For a given number of steps n and a – possibly varying – step size α, at iteration $i < n$:

$$x_{adv}^{i+1} = x_{adv}^{i} + \alpha \times sign(\nabla_x J(\theta, x, y))$$

If necessary, x_{adv}^{i+1} is projected to the constrained ball $\mathcal{B}$.

DeepFool DeepFool [91] is an iterative attack which generally provides a smaller perturbation than FGSM. It defines regions of input points that output the same label. For the original point x, it aims at finding the closest boundary to its label region. For this, the authors approximate the region by a polyhedron P and then approximate the distance from x to the closest of P's edges. This is the full procedure for input x and model f:

- $x_0 \leftarrow x$
- while $f(x) = f(x_i)$: Find the smallest perturbation r that brings x_i to an edge of the current polyhedron.
- $x_{i+1} \leftarrow x_i + r$
- $i \leftarrow i + 1$

Fast Adaptive Boundary Attack While DeepFool efficiently produces an adversarial example based on the classification boundaries, and often provides a sample closer to the original input than FSGM, it prioritises time efficiency over closeness to the original point. The authors of [26] consider a similar approach to DeepFool

in their Fast Adaptive Boundary (FAB) Attack. The main differences introduced in [26] are that they take the exact projection – without clipping – and introduce a bias towards x. They also apply other measures, such as random restarts, to improve the quality – i.e. decrease its distance to x – of the adversarial sample found x_{adv}.

Square Attack The attacks mentioned so far are only white-box ones. However, a potential attacker may only have access to the outputs of a model. Among other proposals, the authors of [19] and [5] introduce attacks that only rely on input-output pairs to generate samples. While [19] approximates gradients to find the closest classification boundary, the Square attack [5] bases its approach on the well-known random search optimization problem. Not only does the Square attack require no information about the original model, but it can outperform gradient-based white-box attacks. The idea is simple: it is an iterative method where at each step i, a random update δ is sampled, and it is added to the current value x_i if it helps with the optimization – i.e. if it decreases the loss value in Equation (2.3).

Despite the numerous attacks on NNs, few defenses guarantee a full protection against them as of now.

2.3.7 Protections

Reverse-Engineering Defenses Because the reverse-engineering field is still new when it comes to NNs and attacks are generally not yet considered a significant threat, few countermeasures have been proposed so far.

Many of the existing defenses are hardware-based. The authors of [133] propose an NN accelerator which mitigates timing side-channel attacks. They present a reverse-engineering attack based on instruction files. They note that while preventing leakages from intermediary data requires the encryption of intermediary featuremaps – i.e. the output channels of hidden layers –, not all featuremaps are of importance. Thus, as a defense, they propose to select a subset of *critical* featuremaps. These critical channels are chosen based on their number of nonzero elements, their absolute weight sum and energy consumption. As a defense, they suggest only encrypting the aforementioned critical featuremaps.

[31] claims to introduce the first physical side-channel countermeasure for NNs. It provides a masking and hiding hardware protection for architectures with binary weights against power-based side-channel attacks. The authors present a masking scheme for individual multiplications. As the masking is not done in modular arithmetic, it still leaks due to a bias in the sign bit. Therefore, they also consider a hiding mechanism to ensure the security of the entire model. The authors then introduce an improvement of their hardware masking, BoMaNet [30]. The new methodology eliminates all hiding mechanisms, ensuring a masking only defense. Moreover, the scheme in [30] incurs less delay and makes the method glitch-resistant.

Building on [31, 30], the authors of ModuloNet [29] propose to adapt NN architectures to masking instead of trying to introduce masking countermeasures in already existing architectures. They propose a novel BNN architecture whose layers are adapted to hardware masking in modular arithmetic. With this new BNN, the

previously noted leaks in masking [31, 30] are avoided.

[78] aims at preventing the recovery of the architecture through memory access patterns attacks. It first considers [56] as a baseline attack, but proves the security of its protection against stronger memory access pattern attacks. Since memory access patterns leak enough information to provide the entire architecture to a malicious user, the authors of [78] propose to randomize them. To achieve this, they mix three common cryptographic tools: Oblivious Shuffle [44], Address Space Layout Randomization [40] and adding dummy memory accesses.

These five papers are orthogonal to our work, since we only consider software countermeasures in this thesis.

The authors of [82] suggest some countermeasures to their proposed attacks. Changing the weight representation or making *ReLU* constant time would prevent their recovery of the weights. The authors of [124] claim that input-dependent functions – such as *ReLU* represent less than 10% of the computation time. They therefore suggest making them independent to defend a model against an input extraction. Such a method should also mitigate [82], as this approach is similar to the suggestion of making *ReLU* constant time.

Adversarial Defenses Adversarial examples have been the bane of NNs' existence ever since Szegedy et al. introduced them in 2014 [120]. Numerous protections with varying degrees of efficiency have been proposed over the years.

Most adversarial defenses fall under the following categories [100]:

- Adversarial training has been found the most effective line of defense [81, 139, 17]. It consists in incorporating adversarial samples in the training dataset. On top of being efficient against the adversarial attack used to generate the samples, it does not impact the original model's accuracy much – it can even reach state-of-the-art accuracy in some settings.

- Some papers choose randomization to counter adversarial samples [76]. Because NNs are designed to be resistant to random noise, a possible scheme consists in introducing randomness in the inference. Ways of achieving this are through a random transformation of the input, the incorporation of random noise or through the masking of some random parts of the output. Although randomization performs well in the black and gray box contexts, adversarial attacks remain efficient in the white box case.

- NNs with a higher weight sparsity rate – i.e. with a higher rate of 0-valued weights – have been observed to be more robust to adversarial attacks [46]. Following this observation, the authors of [138] use L1 regularization in order to improve the sparsity. For a given loss function $\mathcal{L}$, L1 regularization corresponds to adding a penalty term during the training phase, as follows:

$$\mathcal{L}'(\theta, x, y) = \mathcal{L}(\theta, x, y) + \lambda \sum_{w \in \theta} |w|$$

  where θ are the model's current parameters and λ is the regularization factor.

- Other papers consider eliminating adversarial noise through denoising techniques. The aim of denoising is to filter out noise by only keeping the features in an input that matter for the predictions. One such way of achieving this filtering is through the use of autoencoders [86, 8]. As for the randomizing methods, autoencoders are effective in the black and gray box scenarios, but less so in white box ones.

Other protections include defenses based on the k-nearest neighbours algorithm [134] or on Bayesian models [77]. The attacks presented so far can be applied to our main focus – image classification. But other tasks also have specific defenses [100]. While adversarial training seems to be the most efficient method so far, it is generally tailored to a specific attack. Moreover, it requires generating a large set of adversarial samples for training, as well as retraining the original model on the newly generated dataset. This necessitates time and computational resources. While an additional noise layer or a particular preprocessing generally require a specific training as well, incorporating an autoencoder for denoising purposes does not.

2.3.8 Conclusion

NNs form an important part of our daily lives. They have made major leaps of progress over the past years, even managing to beat the best Go players in the world in 2016 [114]. New optimization and learning tricks, better storage facilities, access to larger datasets have all contributed to make them efficient, fast and reliable tools. But the training data, inputs, parameters and hyperparameters all constitute sensitive information that could cause harm in the wrong hands. For all those reasons, they have been the target of multiple attacks over the years, and more recently of reverse-engineering attacks. However, so far, the number of effective countermeasures against reverse-engineering attacks remains small. This is why, in what follows, we try to tackle some of the weaknesses that have arisen. We focus on software countermeasures against mathematical, cache-based and physical reverse-engineering attacks. The mathematical and physical attacks tackled in Chapters 3 and 4 aim at recovering the parameters in a gray-box context and seek a fully equivalent extraction. The cache-based attack tackled in Chapter 6, on the other hand, determines a victim model's architecture in a black-box context. We also apply one of our countermeasures to counter white-box adversarial attacks in Chapter 5.

We start by addressing the gray-box context through the introduction of parasitic models in Part I.

Part I

Parasites as a Gray-Box Protection

Overview

As seen in Section 2.3, reverse-engineering attacks targeting NNs to recover their parameter often rely either on the full knowledge of the architecture or some prior knowledge about it. Even in the case of [10], the first step is to determine the number of neurons in the model, and the weights extraction goes hand in hand with finding a neuron's layer. The assumption that a model is known is often realistic, as common architectures and pretrained models – such as VGG16 [115] or ResNet [48] – can be fine-tuned to be adapted for various tasks. Since the use of common architectures is efficient and wide-spread for many usual tasks, it is possible that potential attackers have at least some key prior knowledge about the architecture. In fact, the authors of [16] write that "in practice, there are only a few architectures that represent most of the deployed deep learning models".

As obtaining a model's optimal parameters for a given task takes a long time and requires computational resources, they are a valuable asset that should be kept secret. This is even more the case considering that once the parameters are revealed, other attacks such as gradient-based adversarial attacks [43, 80, 91, 26] and membership inference attacks are made easier.

Most efficient adversarial attacks are white-box ones and make use of the gradient to fool the target NN. They rely on the internal structure of NNs to determine the closest input which changes the model's predictions. Indeed, a model's structure – including both parameters and hyperparameters – determines the classification boundaries. Some works, such as DeepFool [91] specifically look for those boundaries when searching for the best adversarial sample.

These reasons lead to the conclusion that changing a model's internal structure shoulf constitute a countermeasure to several reverse-engineering and gradient-based adversarial attacks. The first two Chapters in this Part propose a way to mitigate gray-box parameter extraction reverse-engineering attacks: the first focuses on mathematical attacks such as [16], while the second considers physical ones such as [10]. The third Chapter considers an application of the proposed methodology to counter adversarial examples.

Chapter 3: Parasites against Mathematical Attacks

Mathematical attacks [16, 61, 88, 87, 103] query a model f and collect input-output pairs $(x, f(x))$ to uncover its trained parameters. In Section 2.3.2, we detail one particular attack, published by Carlini et al. in 2020 [16]. It is based on hyperplanes created by the piece-wise linear activation functions. In particular, in their work, they focus on FCNs with *ReLU* activation functions. We will follow suit in this Chapter. Our aim is to mitigate the aforementioned mathematical attacks by changing the original model's internal structure.

To achieve this, we introduce the notion of *parasitic* models, that will be used in all three Chapters constituting this Part. These parasites are added to the original target model so as to change its structure and obfuscate its weights and biases.

After stating the threat scenario and giving an overview of this Chapter's protection in Section 3.1, we explain the need and simplicity of approximating identity functions with CNNs in Section 3.2. Then, we dive into the details of the proposed defense in Section 3.3. Section 3.4 discusses the theoretical security of our defense by showing that we indeed change the internal structure of the target model. We conduct experiments to demonstrate the impact of the parasites on the original model in Section 3.5. Finally, Section 3.6 concludes this Chapter.

3.1 Threat Scenario and Defense Overview

We consider the same threat model as in [16]:

- The model is a piece-wise linear function.
- The model is comprised of linear FC or convolutional layers followed by *ReLU* activation functions.
- The attacker has remote access to the model
- The attacker has query access – the attacker can send crafted inputs to the model, and receives outputs in $\mathbb{R}$.
- We also assume that the number of neurons per layer is known.

The goal of the considered attack is functionally equivalent extraction. As detailed in Section 2.3.2, the scheme uses the hyperplanes introduced by the model's neurons due to the *ReLU* functions. More specifically, it is based on the introduced irregularities in the gradients. Increasing the number of these irregularities by adding *ReLU* functions should therefore make the attacker's task more difficult. It is also interesting to note that while most steps involve a linear complexity (in time), sign recovery is, in the general case, more complex and can be exponential. This is the weakness we mean to exploit in this Chapter: adding a few hyperplanes to the structure already highly increases the difficulty of the task, because of this exponential time complexity for the last step.

In this Chapter, we consider changing the internal structure of the victim model by adding linear layers followed by *ReLU* activation functions, without changing the accuracy. Furthermore, this protection is independent from the victim model. The full scheme is as follows:

- Train a set of CNN models, called *parasite(s)* to approximate a noisy identity function. Let us note that this training is completely independent from the model under attack.
- Place the parasite(s) at one or several locations in the model.
- The inputs are run through this modified model.

The suggested countermeasure is further detailed in Section 3.3.

Related Works The authors of [49] inject normal noise during a model's training as a way of mitigating adversarial attacks. They introduce a parameter, α, trained along the original model so that $\alpha \times \mathcal{N}$ – where $\mathcal{N}$ is a fixed Gaussian noise – is added to some layers. Furthermore, they add adversarial examples to the training set to prevent α from converging to 0. Contrary to [49], our parasites are trained independently from the base model, and do not rely on adversarial training. Moreover, we do not simply seek to add noise, but also dummy hyperplanes through new *ReLU* functions. Our aims also differ: we do not wish to ensure a robustness against adversarial examples but to mitigate reverse-engineering attacks.

Shamir et al.'s paper [111] note the link between classification boundaries and adversarial examples inspired this protection: it is the intuition behind our wish to change the base model's internal structure.

3.2 Approximating the Identity using CNNs

As explained, the goal of the countermeasure is to increase the number of hyperplanes. Adding layers with *ReLU* activation functions results in the addition of dummy hyperplanes, as detailed in Section 3.4. Since we also wish to maintain the original model's accuracy, the parasitic layers we add correspond to a model approximating the identity. However, because of the way CNNs adapt to the task at hand, simply approximating the identity function allows for the hyperplanes to 'disappear' from the observable space, and thus, is not enough to thwart the attack at hand.

In order to mitigate the said attack, our parasitic CNNs approximate the identity to which we add a centered Gaussian noise. Section 3.4 details how this additional noise ensures that the introduced hyperplanes lie in the same space as the original ones.

Since CNNs are intrinsically nonlinear, approximating the identity – the simplest linear mathematical function – would appear to be a difficult learning task. However, thanks to the bias and the piecewise linearity of *ReLU*, CNNs manage to avoid the obstacle of the hyperplanes by shifting the input to a space where the activation function is linear. Therefore, CNNs manage to approximate the identity very accurately.

The simplicity of the task at hand is demonstrated in [148]. Indeed, the authors of [148] manage to approximate the identity mapping using CNNs with few layers, few channels and only one training example from the MNIST dataset [28].

First, they observe that while both CNNs and FCNs achieve a good approximation of the identity on digits when trained on three training examples from the MNIST dataset [28], only CNNs generalize to examples outside of the digits scope. Moreover, they state that this bias can still be observed when the models are trained with the whole MNIST training set.

In order to better characterize the observed bias, the authors take the worst case scenario: they only train FCNs and CNNs on a single training example. Contrary to what they expected, architectures that are not too deep manage some kind of generalization: FCNs output noisy images for inputs that are not the training example, while CNNs still manage to approximate the identity. Moreover, FCNs tend to correlate more to a constant function than to the identity. The correlation between the output of CNNs and the identity function decreases with a smaller input size and a higher filter size.

The authors of [148] consider the depth and filter sizes necessary to learn the identity in their constrained context. They show – by providing possible filter values – that in their case, if the input has n channels, $2 \cdot n$ channels suffice to approximate the identity mapping with only one training example. They also note that adding output featuremaps does help with training. Moreover, they use 5×5 filters for all their CNNs' layers. Finally, they explain that even though 20-layer CNNs can learn the identity mapping given enough training examples, shallower networks learn the task faster and provide a better approximation.

This ability of CNNs to learn the identity mapping using only one training example from the MNIST dataset and to generalize it to other datasets shows the simplicity of the task. Section 3.3 and Section 3.4 explain how this fact impairs the defense when the parasitic CNN approximates the identity mapping. They also state the necessity to approximate a noisy identity as well as to apply some constraints on the CNN's parameters.

3.3 Changing the Internal Structure

Let us consider a *ReLU* NN model – an NN with *ReLU* as its activation functions – to protect. The attack scenario described in Section 2.3.2 is based on the bent critical hyperplanes induced by the *ReLU* functions in the model. In [16], the bent

hyperplanes are especially used in the case of expansive NNs – i.e. for which a preimage does not always exist for a given value in the output space –, in order to filter out witnesses that are not useful to the studied layer. To make the attacker's task more complex, we propose to add artificial critical hyperplanes. Adding artificial hyperplanes would make the attack more complex: in a given layer, the attacker would not only have to filter out hyperplanes from other layers, but also artificial hyperplanes. The additional hyperplanes also make the sign recovery process more time consuming.

As explained in Section 3.2, CNNs can provide a very good approximation of the identity mapping. Moreover, they generalize well: with only a single training example from the MNIST dataset, CNNs up to 5 layers deep can still reach the target function.

Parasites Based on the aforementioned observations, we propose to add dummy hyperplanes through the insertion of parasitic CNNs between two layers of the model to protect. To prevent a drop in the accuracy, these CNNs approximate the identity function to which only a small centered Gaussian noise has been added. For a CNN C and input x, this can be written as:

$$C(x) \approx x + \mathcal{N}(0, \sigma)$$

where $\mathcal{N}(0, \sigma)$ is a Gaussian distribution with mean 0 and standard deviation σ.

The parasitic CNNs select a number nb of neurons from the previous layer as their input, and they return their noisy approximation.

Why add a noise? As will be further discussed in Section 3.4.2, simply adding a CNN approximating the identity function is not enough to add dummy hyperplanes. CNNs' training process has several tricks at its disposal to make those added *ReLU* functions redundant and unobservable by a potential attacker. It can ensure that the hyperplanes are either very close and almost parallel to the already existing ones, or very far from the observable input space. In both these cases, there is a high probability that the attacker would not notice the parasites, and the attack's efficiency would therefore not be impacted. Further constraints on the parasitic CNNs are required. Approximating a noisy identity instead of an exact identity mapping is a first step towards making the protection efficient. The goal of the CNN C is, given an input x, to return $C(x) \approx x + \mathcal{N}(0, \sigma)$ instead of $C(x) \approx x$. Further constraints are also applied during training to further ensure a change in the internal structure.

Other parasitic characteristics As Chapters 4 and 5 will delve into more deeply, it is also possible to consider a dynamic addition of parasitic models. Instead of considering a fixed position for the parasite(s), we can randomly select the parasites from the set of pretrained CNNs and a location in the model, at each run. For instance, in a client-server setting, each query from the client will be met with the output of a different model: the number, location and subset of the selected parasites differ from one query to the next.

Furthermore, the small CNN(s) we add do not act on all neurons. The input of a given parasite is a subset of the neurons from the previous layer. This yields two advantages:

- The added CNN(s) can be small, implying fewer computations during inference.
- We can add different CNNs to different parts of the input, to further increase the difference in behavior between neurons.

Figure 3.1 shows an example of adding such a parasitic CNN between the first and the second layer of an NN with only one hidden layer. Let us note that because we take our parasites to be CNNs, the number of its input neurons must be $nb = ch \times n_w \times n_h$ where ch is the number of input channels of the considered parasite, and n_w and n_h are two integers. When the parasitic CNN receives the set of neurons from the considered FC layer, it first reshapes it into ch images of width n_w and height n_h.

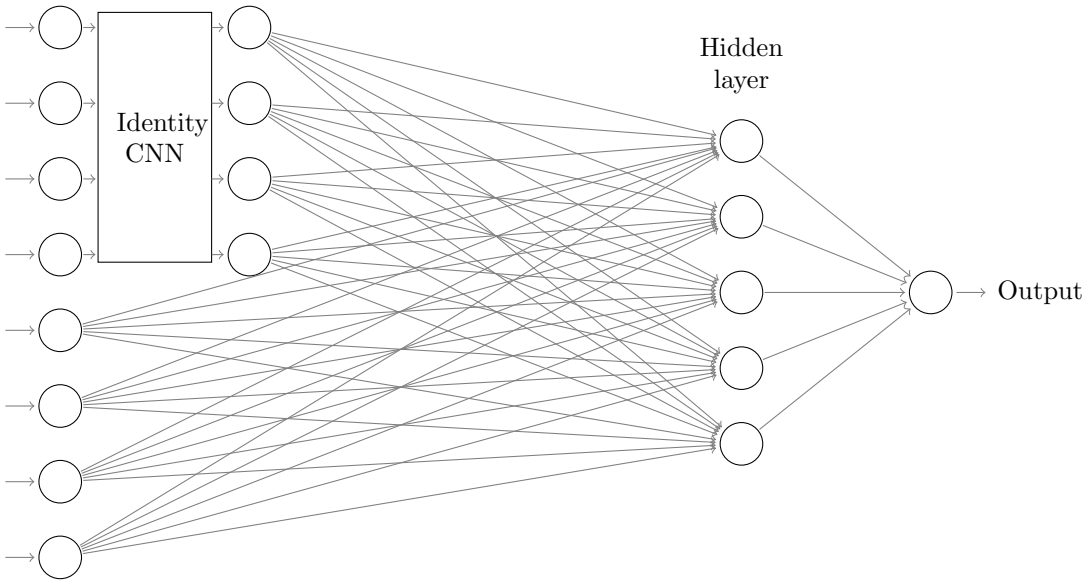


Figure 3.1: Neural Network with one hidden layer where a CNN approximating the identity has been added to approximate the first fours input neurons.

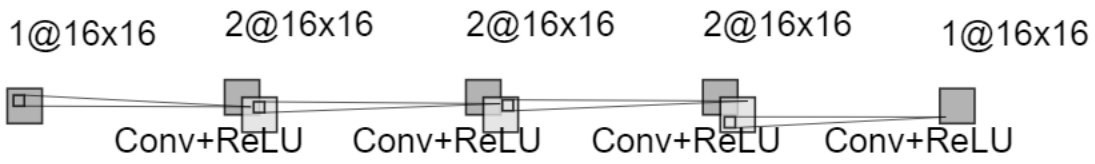


Figure 3.2: Parasitic CNN with 4 convolutional layers, with a *ReLU* activation function after each convolution. The shape of the inputs and outputs are indicated in the form `nb_channels@width×height`. Image generated with [73].

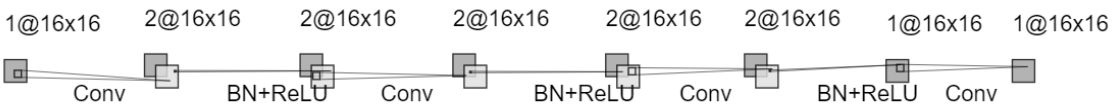


Figure 3.3: Parasitic CNN with 4 convolutional layers, with a Batch Normalization layer and a *ReLU* activation function after each convolution. The shape of the inputs and outputs are indicated in the form `nb_channels@width×height`. Image generated with [73].

Moreover, for the much harder task tackled by the authors of [148], and even though more channels improve the accuracy, $2 \cdot n$ channels in the intermediary layers are enough to get a good approximation of the identity for an input with n channels. Since we do not constrain ourselves to training our CNN with a single example, we can also limit the number of channels in the hidden layers to two – with only one input channel. This enables us to minimize the number of additional computations for the dummy layers, with only a slight drop in the original model's accuracy (see Section 3.5.3).

3.4 Complexity of Extraction in the Presence of Parasitic Layers

Adding a convolutional model with t layers as described in the previous section results in adding t layers to the architecture while keeping almost the same accuracy. If those t layers add critical hyperplanes, then the complexity of extraction increases. This section is dedicated to showing how the change in the structure impacts the attack.

We first consider a CNN approximating the identity mapping added after the first layer in the victim NN. We further assume that there are fewer neurons in the second layer than in the first. We prove that in that case, the parasitic CNN does add hyperplanes with high probability. Then, we explain the need to approximate a noisy identity mapping rather than the identity itself.

3.4.1 Proof of Additional Hyperplanes

For simplicity, we only consider CNNs with one input channel and square images, without loss of generality. Let us consider a target model with m input neurons. Let us suppose we add one parasitic layer that takes inputs of shape $n \times n \leq m$. Let also $\{F_{i,j}\}_{1 \leq i \leq k,\ 1 \leq j \leq k}$ be its associated filter of shape $k \times k$. This results in the following weight matrix C:

$$\begin{cases} C_{i \times n+j,(i+l) \times n+j+h} = F_{l,h} \ \forall (i,j) \in [\![1, n-k+1]\!]^2 \text{ and } (l,h) \in [\![1,k]\!]^2 \\ C_{i,i} = 1 \ \forall i \geq (n-k+1) \times (n-k+1) \\ C_{i,j} = 0 \text{ otherwise} \end{cases} \tag{3.1}$$

Here, without loss of generality, we assume there is no padding.

This new layer adds at most $n \times n$ bent hyperplanes. This number decreases if two neurons η_i and η_j share the same hyperplane.

Let $\mathcal{V}(\eta_i, x)$ be the value of η_i before the activation function, if the model's input is x.

We need to consider two cases:

1. η_i and η_j are in different layers. Let us suppose that η_i's layer is l and that η_j's layer is $l+1$. If the layers are not consecutive, then η_j's hyperplane is bent by $ReLUs$ from the layers in between, making the probability of the two hyperplanes matching very low.

2. η_i and η_j are in the same layer.

First case: η_i is on layer l and η_j is on layer $l+1$

Let us suppose that η_i is on the first layer, and η_j is on the second one. Let χ denote the set of inputs. The output $z(x)$ of the first layer, for $x \in \chi$ is:

$$z(x) = A^{(1)}x + \beta^{(1)} \tag{3.2}$$

In this proof, the rows of $A^{(1)}$ are assumed to be linearly independent. This is an assumption made in [88], [61] and in [27]. As stated in [61], this is likely to be the case when the input's dimension is much larger than the first layer's. The authors of [16] state that it is the case in most $ReLU$ NNs, but not necessarily the most interesting ones. The general attack in [16] for the cases where the model to protect is not contractive is more complex, and requires a layer by layer brute force attack for the sign recovery.

Since we apply the parasitic layer, the output of the second layer is:

$$Out = C \cdot ReLU(z(x)) + \beta^{(2)} \tag{3.3}$$

Since the rows of $A^{(1)}$ are supposed to be linearly independent, for a given vector V, there exists a solution x^* such that $z(x^*) = V$, by the Rouché-Capelli theorem. If we select V so that $V_i \geq 0 \; \forall i \leq m$, then V is not affected by the $ReLU$. We can therefore select a vector V such that, letting k be the convolutional layer's filter size:

$$\begin{aligned} &V_{(\lfloor \frac{j}{n} \rfloor + h) \times n + j\%n + l} = 0 \; \forall 1 \leq l, h \leq k \text{ (where } j\%n \text{ means } j \text{ modulo } n) \\ &\text{except for one value } i' \neq \eta_i, \text{ where } V_{i'} = 1 \end{aligned} \tag{3.4}$$

where j is the first element in V necessary to compute η_j.

Since this second layer is a convolutional one, $\beta_i^{(2)}$ is the same for all i on a given channel, denoted β. The window to compute η_j is zeroed out, except for one value. The filter weight associated to that value needs to be $-\beta$ to nullify η_j. Since we can repeat the process for all values of the window that are not η_i, all the filter weights except for that associated to η_i need to be $-\beta$ except for the one associated with η_i. This is not the case with high probability. In our experiments, we observe that our parasitic weights do not have this form. Thus, with high probability, $\eta_i = 0$ does not imply that $\eta_j = 0$.

For deeper layers, if z is the output of layer l, then even though we cannot select any vector V, it is unlikely for the following equation to happen:

$$z_i(x) = 0 \iff C(ReLU(z(x)) + \beta^{(l)})_j = 0 \tag{3.5}$$

where i is the index of η_i in $z(x)$ and j is the index of η_j in the parasitic layer.

When η_i is not in the window used to compute η_j, and the two neurons are therefore unrelated, it is even less likely to be the case.

Therefore, two neurons on different layers are very likely to have different critical hyperplanes.

Second case: η_i and η_j are in the same layer

Let us suppose that η_i and η_j are in layer l. Let l be the first convolutional layer. Moreover, let us suppose that the CNN is set after the first layer of the model we want to protect. Then l's input is:

$$z(x) = ReLU(A^{(1)}x + \beta^{(1)}) \tag{3.6}$$

where x is the model's input.

Let us also suppose, without loss of generality, that $j > i$. This means that the windows used to compute the two neurons are not identical. With high probability, one of the filter values associated with the disjoint window values is nonzero. For simplicity, and without loss of generality, let us suppose, in what follows, that $F_{1,1}$ is such a filter value: $F_{1,1} \neq 0$.

Case where $\beta = 0$: As explained before, we can find x^* such that:

$$z(x^*)_h = \begin{cases} 1 \text{ if } h = i \\ 0 \text{ otherwise} \end{cases}$$

Since $j > i$, $z(x^*)_i$ is not in the window used to compute η_j, but it is in η_i's window. In this case, $\eta_i \neq 0$ and $\eta_j = 0$. Thus, η_i and η_j do not share the same critical hyperplane.

Case where $\beta \neq 0$: If $\beta \neq 0$, we cannot directly apply the previous reasoning. Let x^* be a witness for η_j being at a critical point. Let us show that we can find an input x^{**} such that $\eta_j = 0$ but $\eta_i \neq 0$.

If x^* already satisfies this property, our work is done. Otherwise, x^* is such that $\eta_i = \eta_j = 0$. As explained before, there exists an input to the NN x' such that:

$$(A^{(1)} \cdot x')_h = \begin{cases} a \text{ with } a > 0 \text{ if } h = i \\ 0 \text{ otherwise} \end{cases}$$

Then, by piecewise linearity of z, we have, for a value a large enough, that:

$$z(x^* + x')_i > z(x^*)_i$$

Moreover, for all other indices h,

$$z(x^* + x')_h = z(x^*)_h$$

Let us consider $x^{**} = x^* + x'$. We have that z_i is not in η_j's window, which means that η_j remains unchanged and $\eta_j = 0$ when the NN's input is x^{**}. On the other hand, η_i's value changes since one of its window values changes and $F_{1,1} \neq 0$. Thus, $\eta_i \neq 0$. Therefore, we can indeed find x^{**} such that $\eta_j = 0$ but $\eta_i \neq 0$.

Let us now consider the case where η_i and η_j are on deeper layers, in which case the previous proof does not work. Let $i = i_1 \times n + i_2$ and $j = j_1 \times n + j_2$, where $i_1 \neq j_1$, and/or $i_2 \neq j_2$. Let also F be the filter of the considered convolutional layer, of size $k \times k$.

If η_i and η_j share the same hyperplane, then whenever z is such that $C_i z + \beta = 0$, we have that:

$$\sum_{l=1}^{k} \sum_{h=1}^{k} F_{l,h}(z_{(i_1+l)\times n+i_2+h} - z_{(j_1+l)\times n+j_2+h}) = 0 \tag{3.7}$$

Since Equation (3.7) needs to hold for all the z that are on the hyperplane, this equation is unlikely to hold.

Therefore, with a high probability, no two neurons in the same layer share the same hyperplane.

3.4.2 Approximating a Gaussian Noise for a Complex Extraction

As explained before, adding CNNs approximating the identity to a victim neural network adds hyperplanes. However, this does not necessarily lead to an increased complexity for the extraction attacks at hand. Indeed, the parasitic CNN might avoid the complexity of the task by isolating the newly introduced hyperplanes – meaning the critical points are far from the input space –, or very close and parallel to the original hyperplanes – i.e. the critical points correspond to a small translation from the original points. The first case can be achieved by increasing the bias in the convolutional layers, so that all values are made sufficiently large. This ensures that no value is zeroed out during the computations. The last layer's bias then translates the values back to their original position. In both cases, the attacker would not notice the introduced hyperplanes, thus defeating the purpose of the parasitic CNN.

Similarly to [49], we inject noise into our layers in order to avoid cases where the CNN we add is not detectable by an attacker. However, our method separates the training of the added CNN from that of the model to protect. Having to train each parasitic CNN along with the original model would result in too much overhead. This is especially the case since we wish to make our protection generic, and therefore independent of the victim model. We inject a fixed Gaussian noise to the labels during the training of our CNN approximating the identity.

The standard deviation of this added noise is selected so as to avoid a significant drop in the original model's accuracy. Let us note that even though the selected

standard deviation might depend on the victim network, several CNNs approximating the identity are trained independently from the victim network, and the victim can then select one or several CNNs adapted to the network at hand.

Since the added noise is fixed, it only constitutes a translation of the victim hyperplanes, and can be approximated by the CNN through an increase in the bias β. We avoid this case by bounding the bias to a small value ($||\beta||_2 < \epsilon$, where $||\cdot||$ is the ℓ_2 norm) or eliminating the bias ($\beta = 0$). This makes the learning task more complex, and forces the filter values themselves to change, thus preventing the introduced hyperplanes from being simple translations of the original ones.

Let us consider, for instance, the case where one convolutional layer is introduced. As before, let C be the matrix associated to the layer and $\mathcal{N}$ be the fixed Gaussian noise. Then the optimization problem becomes:

$$\sum_{1 \leq k \leq m} C_{i,k} x_k = x_i + \mathcal{N}_i \ \forall 1 \leq i \leq n \tag{3.8}$$

where n is the number of output neurons and m is the number of input neurons. The only element that is independent of the input is the noise $\mathcal{N}$. This makes this system of equations impossible to solve for all inputs x. Thus, the solution C^* provided by the CNN is such that:

$$\sum_{1 \leq k \leq m} C_{i,k} x_k = x_i + \mathcal{N}^*{}_i(x) \ \forall 1 \leq i \leq n \tag{3.9}$$

where $\mathcal{N}^*$ is a noise close to $\mathcal{N}$ that depends on the input. C^* leads to hyperplanes for the various inputs which cannot be translations of the input hyperplanes. This implies that the newly introduced hyperplanes intersect the original ones, increasing the chances of modifying the polytopes formed by all the model's boundaries. This explanation generalizes to the case of several layers. Indeed, in the general case, the optimization problem for k convolutional layers without a bias becomes:

$$f(x) = x_i + \mathcal{N}^*{}_i(x) \ \forall 1 \leq i \leq n \tag{3.10}$$

with $f(x) = ReLU(C_k(ReLU(...ReLU(C_1 x))))$, where C_j is the matrix associated to the $j-th$ layer.

In order to further prevent the introduced hyperplanes from being too far from the working space, we add Batch Normalization layers after each convolutional layer. These layers center their inputs, thus also setting the hyperplanes closer to the center – and the 'observable' space for the attacker.

To summarize, we may encounter two main problems when adding dummy hyperplanes:

- The parasitic CNNs can set the hyperplanes far from the input structure.
- The parasitic CNNs can ensure the dummy hyperplanes are almost the same as the existing ones.

To counter those possibilities, we change our parasites' training and architecture:

- The parasites approximate a noisy identity to make the hyperplane structure more complex.

- We add BN layers in between convolutional ones to center the hyperplanes.
- We bound or eliminate biases during the training phase to incentivize the hyperplanes being visible to the attacker.

Measuring the impact on the internal structure To ensure the hyperplanes have indeed changed, we measure the influence on adversarial examples. Adversarial attackers find the shortest path from one prediction class to another. This path depends on the subdivision of the space by the original model's hyperplanes [111, 91]. Thus, changes in the said subdivision lead to different adversarial samples. Conversely, if two models lead to the same subdivision of the space, then adversarial examples remain the same for both models.

This property of adversarial samples is the reason why, in Section 3.5, we measure the impact of the parasitic CNNs on both the original model's accuracy and the adversarial samples. It is important to note that in this Chapter, we do not seek to counter adversarial samples, but simply to change some of them.

3.5 Experiments

In this Section, we detail the model we want to protect as well as the added CNN. Then, we measure the impact of the added layers on the model to protect by counting the number of adversarial samples which do not generalize to the protected model.

3.5.1 Description of the NN Models Used

Parasitic architectures In our experiments, we consider two architectures for our parasites. The first is comprised of:

- One convolutional layer with 1 input channel and 2 output channels.
- Two convolutional layers with 2 input channels and 2 output channels.
- One final convolutional layer with 2 input channels and 1 output channel.

Each convolutional layer is followed by a *ReLU* activation function and has filter shape 5×5. Figure 3.2 shows this first parasite's architecture.

The second architecture is very similar, except that after each convolutional layer and before the activation function, a BN layer is added. Figure 3.3 shows this second parasite. The batch normalizations in this second model normalize their input, ensuring a mean of 0 and a standard deviation of 1. This increases the chances of the *ReLU* functions being activated.

Training the parasites We train these parasitic models over 10,000 random inputs $\{x_i \in [0,1]^n\}_{1 \leq i \leq 10{,}000}$ of size $n = 16 \times 16$. In our experiments, we select the n input neurons as the first or the last ones from the previous layer, but they can be selected at random among the previous layer's neurons. For a given training, we fix $\mathcal{N}$ as a Gaussian noise, and we set the labels to be $\{x_i + \mathcal{N}\}_{1 \leq i \leq 10{,}000}$.

Target models The target model is a LeNet architecture [72] (see Figure 2.4 in Section 2.2.3) where average pooling layers are replaced by max pooling layers. It is trained on the MNIST dataset [28]. We also consider a second model where we introduce BN layers after the convolutional layers of the LeNet architecture (see Figure 3.4). We denote VM the victim LeNet architecture and VM_{batch} the architecture where BN layers have been added.

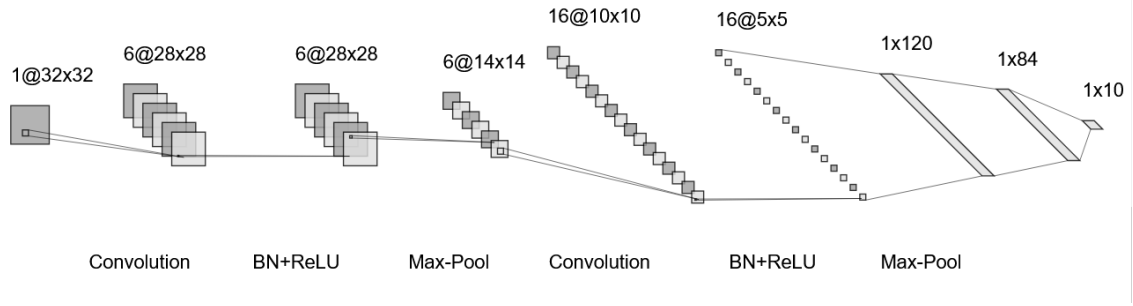


Figure 3.4: LeNet architecture, as in [72], where a Batch Normalization (BN) layer is added after each convolution. Image generated thanks to [73].

VM has an accuracy of 98.78% on the MNIST dataset, while VM_{batch}'s accuracy is of 99.11%.

3.5.2 Generating Adversarial Examples

Several methods enable an attacker to compute adversarial samples [43, 80, 91, 90]. In this Chapter, we use the Fast Gradient Sign Method introduced by Goodfellow et al. [43] (see Section 2.3.6) to determine adversarial samples for our LeNet architecture.

As previously explained, since adversarial examples are based on the subdivision of the input space by the neurons' hyperplanes [111, 91], a modification of those examples is a good indicator of whether the said subdivision has indeed been changed by the added CNN. As our protection aims at perturbating this subdivision, we compute adversarial samples for the first 200 images of the MNIST set using the FGSM method and measure the percentage of examples which do not generalize to the modified model. For the FGSM method, we start with $\epsilon = 0.05$ and increase it by steps of 0.05 until the computed x_{adv} is indeed an adversarial example for the original model.

3.5.3 Results

We test the two original models considered with the added parasitic CNNs, without a bias β or with the constraint that $||\beta||_2 < 0.05$, where $||\cdot||_2$ is the ℓ_2 norm. We consider three experiments:

- We add one parasitic CNN (with and without BN layers) after the first 16×16 neurons of the second convolutional layer in the victim model. We report the results in Table 3.1

- We add one parasitic CNN (with and without BN layers) after the entire first layer in the victim model. We report the results in Table 3.2. This experiment is only possible if the first hidden layer has a number of neurons which can be written as a multiplication of two factors greater than 1. Indeed, the parasites being CNNs, we need to make their input two dimensional.

- We add two or three parasites after the second convolutional layer in the victim model. Both parasites have input shape 16×16. Each of these parasites are fed either the first 16×16 neurons, or the last 16×16. We report the results in Table 3.3

Let us note that in all cases, we only count the number of adversarial samples for the original model that are no longer adversarial for the protected CNN. There are also examples which are adversarial for both models, but with different predictions.

In the first experiment (Table 3.1), the parasites approximate the identity to which a centered Gaussian noise with standard deviation $\sigma = 0.2$ has been added. In every case, we observe a change in the adversarial examples.

Let us denote M_{adv} the percentage of adversarial examples for the original model which are no longer adversarial for the protected model. In all cases from this first experiment, M_{adv}, is higher or equal to 12%, and the accuracy of the protected model is very close to the original one. This shows that the boundaries between classes – which are the result of the various layers' hyperplanes – have changed. Our results are summarized in Tables 3.1 and 3.2.

Table 3.1: Measurement of the accuracy, and percentage of the adversarial samples that are no longer adversarial for the protected CNN (M_{adv}). All tests are made on the MNIST dataset [28], and the parasitic CNNs approximate the identity to which a centered Gaussian noise with a standard deviation of 0.2 was added. BN denotes Batch normalization. All parasitic CNNs were added after the second convolutional layer of the original model.

Parasite Location	**Original Model**	**Original Accuracy**	**Parasite**	**Bias constraints**	**New accuracy**	M_{adv}
After BN and activation (if BN)	*VM* (Fig. 2.4)	98.78%	Without BN	$\|\|\beta\|\|_2 < 0.05$	98.69%	24.5%
				No bias	98.7 %	19 %
			With BN	$\|\|\beta\|\|_2 < 0.05$	98.50 %	28%
				No bias	98.67%	22%
	VM_{batch} (Fig. 3.4)	99.11%	Without BN	$\|\|\beta\|\|_2 < 0.05$	99.24%	17.5%
				No bias	99.14%	14%
			With BN	$\|\|\beta\|\|_2 < 0.05$	99.18%	17%
				No bias	99.15 %	12 %
Before BN and activation (if BN)	VM_{batch} (Fig. 3.4)	99.11%	Without BN	$\|\|\beta\|\|_2 < 0.05$	96.64%	37.5%
				No bias	98.13%	39%
			With BN	$\|\|\beta\|\|_2 < 0.05$	99.05%	27.5%
				No bias	99.16 %	14 %

As Table 3.1 shows, inserting a CNN trained to learn a Gaussian noise added to the identity can lead to a modification of the polytopes formed by the original model's hyperplanes, with only a slight drop in the accuracy.

Table 3.2: Measurement of the accuracy, and percentage of the adversarial samples that are no longer adversarial for the protected CNN (M_{adv}). All tests are made on the MNIST dataset [28]. BN denotes Batch normalization. All parasitic CNNs were added after the first convolutional layer of the original model, and their input is the entire output of the first layer (after the BN layer). The target model is VM_{batch}. The original model's accuracy is 99.11%.

Parasite	**Bias constraints**	**Standard Deviation**	**Accuracy**	$\boldsymbol{M_{adv}}$
With BN	No Bias	0.1	99.11%	9%
		0.2	99.09 %	34 %
		0.3	99.02 %	34%
	$\|\|\beta\|\|_2 < 0.05$	0.1	93.58%	48%
		0.2	99.05%	32%
		0.3	97.55%	46.5%
Without BN	No Bias	0.1	99.18%	16.5%
		0.2	98.28 %	42 %
		0.3	93.33 %	52%
	$\|\|\beta\|\|_2 < 0.05$	0.1	98.79%	46.5%
		0.2	98%	48%
		0.3	98.27%	45%

Adding the same parasitic CNNs but to all the neurons of the first layer, as in the second experiment, leads to higher M_{adv}, with mostly similar accuracy drops. Let us note that the CNNs we use in this case have the same number of layers and parameters. The only difference is the model's input and output sizes. The results are shown in Table 3.2. Given the increased M_{adv} with an acceptable accuracy drop, this strategy seems more interesting. This can be explained by the fact that all neurons are impacted by the change. Because this affects all neurons in the following layers as well, adding a smaller noise to all neurons in a layer seems to yield better results than adding a larger noise to a small portion of the layer's neurons.

It is interesting to note that even though the parasitic CNNs trained with no bias incur a lower M_{adv}, in most cases, they entail a lower drop in the accuracy than the CNNs learnt with a small bias. This might be explained by the fact that the CNN with no bias cannot learn a noise independent of the input, and will therefore tend to get closer to the non-noisy identity. Furthermore, the ability for the parasitic CNN to operate a translation thanks to the small bias can explain the small drop in the accuracy that we observe. However, despite this additional possibility, the CNN with a small bias still changes the slope of the hyperplanes, as the drop in the accuracy is not steep enough to justify the high M_{adv}.

One can also add several parasitic CNNs to a given victim NN, as we did in the third experiment. This might result in a higher protection, with no – or a small – drop in the accuracy. Since the parasitic CNNs are already trained, the cost of adding these CNNs remains small, and is equal to the additional computations required for inference. On VM_{batch}, we try adding parasitic CNNs in three different ways:

- Two parasites after the first layer

Table 3.3: Measurement of the accuracy, and percentage of the adversarial samples that are no longer adversarial for the protected NN (M_{adv}). Several parasitic CNNs were added to the victim NN. All tests are made on the MNIST dataset [28], and the parasitic CNNs approximate the identity to which a centered Gaussian noise with a standard deviation of 0.2 was added. BN denotes Batch normalization. The parasitic CNNs are added after the second convolutional layer of VM_{batch}. We add them before, after, or both before and after the BN layer and activation function. The original accuracy for VM_{batch} is 99.11%. *Small* means that the constraint on the bias β is $||\beta||_2 < 0.05$.

Parasitic CNNs' Locations					Accuracy and M_{adv}	
	2^{nd} layer, Before BN and activation		**2^{nd} layer, After BN and activation**		**New accuracy**	**M_{adv}**
Which neurons?	With BN?	With Bias?	With BN?	With Bias?		
First n	BN	Small	No BN	Small	99%	31%
First n	BN	Small	BN	Small	98.98%	37%
First n	BN	Small	No BN	No bias	98.93%	31.5%
First n	BN	Small	BN	No bias	98.99%	31%
First n	BN	No bias	BN	No bias	98.96%	28.5%
First n Last n	BN BN	Small No bias	-	-	99.05%	31%
First n Last n	-	-	BN BN	Small No bias	99.17%	27.5%
First n Last n	-	-	BN BN	No bias No bias	99.15%	27%
First n First n	-	-	No BN No BN	No bias Small	99.19%	25.5%
First n Last n	BN BN	Small No bias	BN	Small	98.94%	40%
First n Last n	BN -	Small -	BN BN	Small Small	99.03%	38.5%
First n Last n	BN BN	Small No bias	BN -	Small -	98.89%	43%

- One parasite after the first layer as well as one parasite after the second layer
- Two parasites after the first layer and one after the second layer

In our experiments, we observe that adding a small CNN after the first layer did not improve our results much, be it on the accuracy or on M_{adv}. Thus, Table 3.3 gives an example of accuracy and M_{adv} obtained in various cases where the parasitic CNNs are added after the second convolutional layer from VM_{batch}, either before or after the batch normalization layer and the activation function. Let us note that once again, the standard deviation of the added noise is 0.2 in all cases. Moreover, when there are two parasitic CNNs at the same location, the first is applied to the first neurons and the second is applied to the last neurons of the victim layer.

3.6 Conclusion

In this Chapter, we have introduced the notion of *parasites* to discuss mathematical attacks – and more specifically the one introduced by Carlini et al. [16]. The latter relies on the internal structure of NNs induced by piece-wise linear activation functions to determine its weights.

To increase the complexity of this attack, we introduce dummy hyperplanes through parasitic CNNs. These are trained with criteria that aim at making the hyperplanes visible to potential attackers. We can place one or several of those parasites at various locations in a model to protect, as long as the accuracy drop remains small. We evaluate the effect on the structure by measuring the impact of our protection against adversarial examples. We manage to reach a modification in over 31% of the adversarial examples on a LeNet architecture tested against MNIST, with an accuracy that remains almost unchanged (a drop of less that 0.1%).

In this defense, the balance between the parasites' hyperparameters – depth, filter size, number and location – and the drop in the protected model's accuracy is crucial. Here, we showcase the feasibility of the countermeasure. Some tailoring of the parasites' architecture depending, for instance, on the original model's family of architectures (such as VGG or ResNet) could make this method even more efficient.

The use of parasitic models is not limited to mathematical attacks. This Chapter already hinted at a link with adversarial examples, that we will further explore in Chapter 5. But it can also be used to thwart physical SCAs, as we discuss in Chapter 4.

Chapter 4: Dynamic Parasites against Physical Side-Channel Attacks

While the notion of parasitic models was used in the previous Chapter to counter mathematical attacks, we did not focus on the dynamic aspect. So far, we have only discussed fixed parasitic models, as it was enough to change the internal structure of an NN. Section 3.3 mentions that the parasites we add to the target model can be changed at each run. Here, we show how, when placed at the entrance of the base model, this dynamic selection of parasitic models at each run can help mitigate physical SCAs.

As explained in Section 2.3.3, potential attackers can use physical leakages such as power or EM emanations to recover a model's secret weights and biases. Section 2.3.5 presents one such physical SCA: CSI Neural Networks [10].

Physical SCAs generally require a physical access to a device's processor. EM emanations can also be gathered from a distance [14]. In both cases, smartphones and other everyday embedded systems are at risk. It is therefore crucial to protect models adapted to small devices, such as MobileNetV2 [108], from reverse-engineering attacks.

Contrary to cryptographic algorithms, ML tasks do not require exact outputs: it is possible to get different outputs for a very similar model accuracy. We rely on this fact to show that dynamic parasites can hide a model's input, making the extraction much more complex.

After introducing the threat scenario and giving an overview of the proposed defense in Section 4.1, we delve into the details of our proposal in Section 4.2.1. We then evaluate our methodology through simulated attacks in Section 4.3 and discuss the results and the provided security in Section 4.4. Finally, we conclude this Chapter in Section 4.5.

4.1 Threat Scenario and Defense Overview

The authors of [10] aim at recovering a functionally equivalent model relying on EM and/or power emanations (see Section 2.3.5). For small FCNs and CNNs, they manage to extract both the architecture and the parameters with high accuracy.

Threat Scenario We consider a threat scenario that is close to the one in CSI NN [10]. The CSI NN attack is detailed in Section 2.3.5. However, our scenario is

gray-box, with a stronger attacker who has access to the architecture. Thus, the attacker:

- Knows the target model's architecture.
- Knows the model's input.
- Aims at recovering the weights and biases of the said model.
- Can monitor electromagnetic emanations and use CEMA (see Section 2.3.5) to extract the parameters.

We present our approach and carry our experiments out in the case of EM-based attacks. However, we believe that other physical leaks would lead to the same results.

Protection Overview Common cryptographic methods to tackle SCAs cannot be applied to protect millions of parameters, as they incur too much overhead [158]. In our paper "Premium Access to Convolutional Neural Networks" [154], we study various ways of protecting only part of a model. We observe that slightly changing the values in one channel in the first convolutional layer leads to noticeable changes in the accuracy. We protect the sensitive weights through a secret key. However, the goal in that paper is to provide a premium access – i.e. a model with a higher accuracy – to paying customers, and a valid but not optimal model to others. The problem at hand is very different, as we wish a potential attacker to be unable to access a model with a decent accuracy. Thus, a different approach is required.

The first observation one can make is that since the attacker in [10] – described in Section 2.3.5 – proceeds layer by layer, protecting the first layer should be enough to ensure the entire model's security. Indeed, the deduced weights from the first layer are used to compute the input to the following layer and proceed with the attack.

A second observation is that the attacker requires the input values to recover the first layer's weights. An intuitive countermeasure to this attack would therefore consist in hiding the said input values.

Third, contrary to common cryptographic problems, NNs' layers do not require exact responses. The output of each layer can be approximated as long as the model's accuracy is not affected.

Fourth, a change in a model's first layer impacts the following layers. By modifying the input of a model, we can therefore hope for a domino effect: a wrong extraction in the first layer is amplified by deeper layers.

Those four observations show that hiding the input values should mitigate the attack at hand. Hence, we propose to dynamically apply one or several CNN models approximating the identity to the entrance of the model. Our aim is to make sure the modified model's input is different from the original input, with only a slight drop in the accuracy. The extracted weights a potential attacker would get for the first layer are then only an approximation of the actual weights. Even if those first weights are still close to the original ones, the error propagates to the following layers, amplifying the noise and making the extracted model's accuracy much lower than the original one.

4.2 Dynamic Parasitic Models

In the following section, we explain a novel approach to protecting against SCAs that target NN parameters.

4.2.1 Proposal Description

Let us now detail our proposal.

As explained in Section 4.1, we aim at hiding a model's input values without decreasing its accuracy, by dynamically adding small CNNs approximating a noisy identity function. Our method is designed to limit the access to the correct input, which is required by the attacker of Section 2.3.5.

Adding layers dynamically means that at each run – or after a certain number of runs –, we consider parasitic CNNs with different weights and/or a different architecture. At each run – or after a certain number of runs –, this new parasite is selected at random among a set of small pretrained CNN models.

To carry out a CEMA, an attacker requires several traces (see Section 2.3.5). If the small CNN we add to the input changes from one run to the next, the attack becomes much harder. As explained in Section 2.3.5, CSI NN [10] enables an attacker to determine the architecture of a victim NN along with its parameters, through SEMA, CEMA and time analysis. The randomization in both the weights and architecture due to the dynamic addition of parasites should mitigate the statistic attack, even when the attacker tries to recover the architecture along with the weights.

Thus, our methodology is as follows:

- Train a set S of small CNNs that approximate a noisy identity. For x in the input space I and $s \in S$: $s(x) = x + \mathcal{N}_s(x)$ where $\mathcal{N}_s$ is a Gaussian distribution
- At each run, select $s \in S$. For $x \in I$, run $x' := s(x)$.
- Feed x' to the target model.

Figure 4.1 shows the location where we add the parasitic CNN $s \in S$, to an FC model with one hidden layer. Here, s is applied to the input x and the result x' is then fed to the model at hand.

Let us analyse the consequences of adding such CNNs at the entrance of the model. Let X be the model's input, X' be the noisy input and $x_{i,j} \in X$. To recover the weight w in each multiplication $x_{i,j} \cdot w$, the attacker requires several power or EM traces for that operation. She also needs to know $x_{i,j}$. But the input of the target model is $s(X) = X'$, and the attacker does not have access to X'. Furthermore, given she requires several traces and the parasitic CNN changes from one run to the next, the attack cannot be carried out on the parasitic layers directly, knowing input X. This is why we believe that our protection thwarts first-order physical side-channel attacks such as CSI NN [10] (see Section 4.3 for our results).

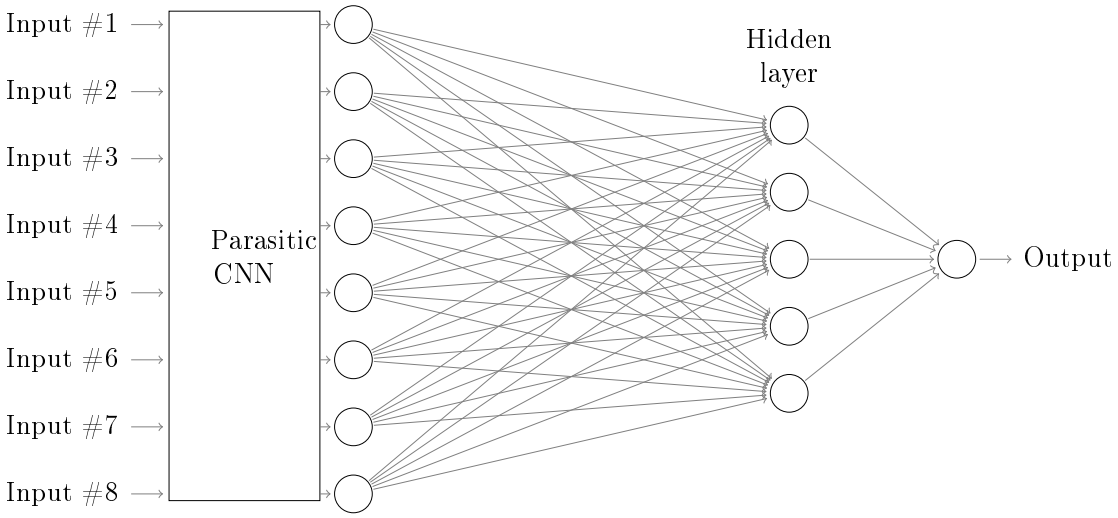


Figure 4.1: Fully Connected Neural Network with one hidden layer where a parasitic CNN approximating the identity has been added to approximate the model's input.

4.2.2 Approximating the Identity Function

To prevent the attack, our goal is to add noise to the input without decreasing the model's accuracy.

Noisy identity One possible way to achieve this is to add CNNs approximating the identity to the entrance of the original model. As in Section 3.2, if we denote M such a CNN model, then M is trained to reach $M(x) = x$. This yields the advantage of making the attack in [10] harder in the extended case where the attacker does not know the architecture.

But, as explained in Section 3.2 and as shown in [148], the simplicity of approximating the identity through a CNN model M leads to practically no difference between the input x and the model's output $M(x)$.

Because of this, simply adding a CNN approximating the identity does not have any impact on the attack. Indeed, no noise is added to the input, therefore keeping the observed values unchanged to the attacker. Thus, we follow the same training approach as in Chapter 3. The parasitic models we add are trained to approximate the identity function to which we add noise. The goal when training a parasitic CNN model M then becomes $M(x) = x + \mathcal{N}$, where $\mathcal{N}$ is a given Gaussian noise. This argument justifies our training proposal in Section 4.2.1.

Additional computations Let us now compute the complexity of adding such CNN models to the original model. Let us consider a convolutional layer with input shape (n, n, c_{in}), c_{out} output channels and a filter of shape $k \times k \times c_{out}$. Then the output of the layer has shape $(n' = n - k + 1, n' = n - k + 1, c_{out})$. Let us denote X the input of the layer, O its output and W its weights with shape (k, k, c_{in}, c_{out}).

Then, as per Section 2.2.1, we have the following:

$$O_{i,j,q} = \sum_{c=1}^{c_{in}} \sum_{l=1}^{k} \sum_{h=1}^{k} X_{i+l,j+h,c} \cdot W_{l,h,c,q}$$

This implies that for each neuron, $k \times k \times c_{in}$ multiplications are required. Therefore, each output channel adds $k \times k \times c_{in} \times (n-k+1) \times (n-k+1)$ multiplications. Since we have c_{out} channels, the total number of additional multiplications for one convolutional layer is:

$$M = c_{out} \times k \times k \times c_{in} \times (n-k+1) \times (n-k+1) = O(n^2)$$

Generally, convolutional layers include some zero-padding to preserve the input and output sizes. This increases the necessary multiplications to:

$$M = c_{out} \times k \times k \times c_{in} \times n \times n$$

To conclude, adding l identical layers leads to an increase in the number of multiplications of $l \times k \times k \times n^2 \times c_{in} \times c_{out}$.

In this Chapter, we decided to apply the model to each channel independently. Thus, the input number of channels is such that $c_{in} = 1$ here, but we multiply the total by the actual number of input channels. The new formula for l layers is then:

$$M = c_{in}^{(1)} \times \left[c_{out}^{(1)} \times {k^{(1)}}^2 \times {n^{(1)}}^2 + \sum_{i=2}^{l} c_{out}^{(i)} \times {k^{(i)}}^2 \times {n^{(i)}}^2 \times c_{in}^{(i)} \right]$$

where $c_{in}^{(i)}$ is layer i's number of input channels, $c_{out}^{(i)}$ is layer i's number of output channels, $k^{(i)}$ is layer i's filter size and $n^{(i)}$ is the layer's input size. Let us note that generally, $c_{out}^{(i-1)} = c_{in}^{(i)}$, since a layer's input is the previous layer's output.

The reason why we choose this approach is so that the parasitic CNNs can be adapted to inputs with any number of input channels. Moreover, this way, we can keep a low number of channels in the middle layers without fearing any loss of information.

4.3 Evaluation

4.3.1 Simulation

To comfort our idea, we simulate two attacks where the attacker's aim is to extract a target model's weights:

- In the first, we consider one fixed parasitic model at the entrance of the target model. The attacker does not know the parasite's architecture or weights.
- In the second, we select a different pretrained parasite at each run. The attacker knows a parasite has been added, and can therefore try to extract the parasite's weights along with those of the base model.

First simulation The first simulation corresponds to a scenario where the attacker is unaware of the protection. We proceed layer by layer, as in CSI NN [10]. We suppose that the attacker knows all the outputs of intermediate multiplications in the first layer[a].

Let X^1, W^1 and O^1 denote, respectively, the input, the weights and the output of the model's first layer. For simplicity, let us only consider the case where the first layer is a convolutional one. We remind the reader in Section 2.2.1 that:

$$O^1_{i,j} = \sum_{l=0}^{k} \sum_{h=0}^{k} X^1_{i+l,j+h} \cdot W^1_{l,h}$$

Let X'^1 denote the input with added noise. The attacker only knows X'^1 and the values $o^1_{i,j,l,h} := X^1_{i+l,j+h} \cdot W^1_{l,h}$. She proceeds as follows:

1. For each output $O^1_{i,j}$, the attacker computes $W'^1_{l,h} := \frac{o^1_{i,j,l,h}}{X^1_{i+l,j+h}}$ if $X^1_{i+l,j+h} \neq 0$. Otherwise, the attacker finds a nonzero input neuron $X^1_{i'+l,j'+h}$ to compute $W'^1_{l,h}$ in a similar fashion.

2. Once the first layer's weights are recovered, the attacker computes the second layer's weights, $W^2_{i,j}$. For this, she first computes $X'^2 := \sum_{h=0}^{k} X^1_{i+l,j+h} \cdot W'^1_{l,h}$. She takes X'^2 as the input of the second layer and extracts the second layer's weights as in the first step.

3. The attacker repeats the previous step until all the layers' weights have been recovered.

4. If there is a bias, the attacker simply recovers it through: $\beta = X'^2 - O^1$.

In fact, to make sure we introduce as little noise as possible, we average the obtained weights over 64 inputs.

Second simulation In the second simulation, the parasites are introduced dynamically, and the attacker knows about their existence. However, at each run, the parasitic CNN's weights – and possibly the architecture – change. Since, as mentioned in Section 4.2, the attacker requires several traces to recover the weights, this change of model at each run affects the results. We simulate an attacker's potential behaviour by averaging the various extracted weights over several inputs. This second simulated attack also proceeds one layer at a time, as follows:

1. Start with the first parasitic layer, and compute the weights as in the first simulation.

2. Repeat the operation over several inputs, and set the layer's weights to the average w_{av} of the extracted values.

3. The next layer's inputs is the output of the first layer with w_{av} as the weights.

[a]This is an unrealistic assumption, but since we aim to demonstrate the effectiveness of our protection, we assume perfect conditions for the attacker. Real CEMA attacks are much harder.

4. For each remaining layer in the model at hand (including the parasitic layers), repeat the previous three steps.

Let us also note that in both simulations, we discard the weights that result in a division by 0 – meaning that no nonzero input value among the provided set can be found –, by setting the extracted weights to 0.

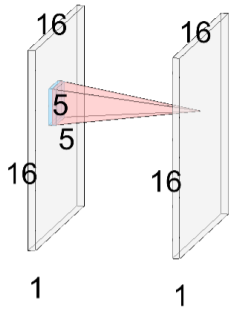
4.3.2 Models and Parasites Considered

Since we are mostly interested in models that could be used on smartphones and embedded systems in general, we consider two small models: LeNet [72] and MobileNetV2 [108], that we trained on the CIFAR-10 dataset [71]. LeNet is only comprised of two convolutional layers followed by three dense layers. MobileNetV2's more complex architecture is detailed in Table 4.1.

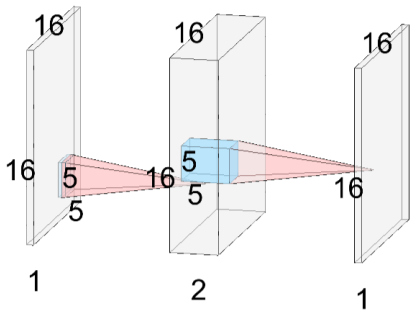
Table 4.1: Description of the MobileNetV2 architecture [108]. Each line corresponds to a group of layers, repeated n times. All layers in a given group have c output channels. The first layer of the group has stride s while the others have stride 1. t is the expansion factor: if there are c input channels and c' output channels in a block, there is an intermediate operation with $t \cdot c$ channels.

Input Shape	Operation	Expansion t	Channels c	Repetition n	Stride s
$224^2 \times 3$	Convolution (3×3)	-	32	1	2
$112^2 \times 32$	Bottleneck	1	16	1	1
$112^2 \times 16$	Bottleneck	6	24	2	2
$56^2 \times 24$	Bottleneck	6	32	3	2
$28^2 \times 32$	Bottleneck	6	64	4	2
$14^2 \times 64$	Bottleneck	6	96	3	1
$14^2 \times 96$	Bottleneck	6	160	3	2
$7^2 \times 160$	Bottleneck	6	320	1	1
$7^2 \times 320$	Convolution (1×1)	-	1,280	1	1
$7^2 \times 1,280$	Average Pooling (7×7)	-	-	1	-
$1 \times 1 \times 1,280$	Convolution (1×1)	-	k	-	-

Finally, the parasitic models we add are based on the ones used in [148], with various layer numbers and in some cases, an additional Batch Normalization layer after each convolutional one. We mainly focus on models with one or two convolutional layers, described in Figure 4.2.



(a) Parasitic model with one convolutional layer. The input and output have shape (1, 16, 16). The filter shape is (5, 5).



(b) Parasitic model with two convolutional layers. All inputs and output have shape (16, 16), but the middle layer has two channels. The filter shape is (5, 5).

Figure 4.2: Parasitic CNN models with one (a) and two (b) convolutional layers. In both cases, the convolution is followed by a *ReLU* activation function and a Batch Normalization layer. (Images generated thanks to [73].)

4.3.3 Results

The set of possible parasitic CNNs should be tailored to the user's requirements and tolerance: some tasks do not require as high an accuracy as more sensitive ones. In general, we consider in this Chapter that a couple of percents drop can be acceptable. To select the architectures that can be added to the set of possible architectures, we determine the accuracy of the protected model depending on the standard deviation of the noise of the parasitic models at hand. Figure 4.3 shows the impact of parasitic models with various standard deviations on a MobileNetV2 architecture with a 71.41% accuracy on the CIFAR10 testing set (88.16% on the CIFAR10 training set). We see that the accuracy decreases with the standard deviation, but also depends on the training. In general, the drop in accuracy seems acceptable until around $\sigma = 0.2$.

First simulation We carry out the first simulated attack described in Section 4.3.1 on a LeNet architecture trained on the CIFAR10 architecture with a 70.69% accuracy on the testing set. The accuracy of the protected model depends on the parasitic model(s) added.

We also check that when no parasite is added to the original model, our simulated attack does not result in a drop in the accuracy. We then add a parasitic model described in Section 4.3.2, approximating an identity function to which we add a Gaussian noise with a standard deviation of 0.2. The training and evaluation of the model at hand was performed on CIFAR10 images which have been normalized. Thus, the images with no protection have a mean of 0 and a standard deviation of 1, and the ratio of standard deviations is : $SNR = \frac{1}{0.2} = 5$.

This leads to an accuracy of 70.84% on the LeNet architecture, which is very similar to the original model's accuracy. The first simulated attack described in Section 4.3.1 results in an extracted model with an accuracy of 12.66%.

In order to test our protection on a more representative NN model, we applied

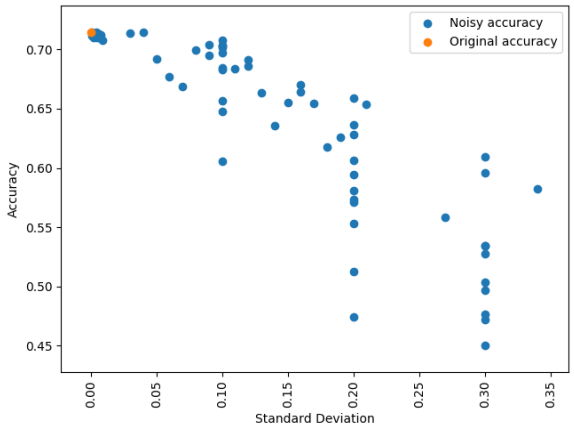


Figure 4.3: Accuracy of a protected MobileNetV2 model with relation to the standard deviation of the parasitic models. The parasitic models P are trained to return $P(x) = x + \mathcal{N}_\sigma(x)$ where $\mathcal{N}_\sigma$ is a Gaussian noise with mean 0 and standard deviation σ.

this first simulated attack on the first twelve layers of MobileNetV2 trained on the CIFAR10 dataset. The original accuracy on CIFAR10's testing set is 71.41%. In this case, the preprocessing of the training images only consists in dividing by 255, making sure that the values are in the range [0, 1]. The standard deviation of the three input channels are, respectively: $0.25, 0.24$ and 0.26. We add a parasitic CNN approximating the identity function to which we added a Gaussian noise with a 0.2 standard deviation. We therefore have an SNR approximately equal to 1 for the three input channels. Adding a two-layer parasitic model with a 0.2 standard deviation leads to a 70.23% accuracy for the protected model. We observe that with only the first twelve layers considered for the first simulated attack, the model's accuracy already drops from 71.41% for the original model to 9.97% for the extracted one. On the training set, adding the same parasite leads to an 85.85% accuracy instead of 88.16%. The accuracy of the extracted model drops to 10.40%.

We summarize our experimental results in Table 4.2. We see that increasing the standard deviation and number of layers in the parasite leads to a much lower extraction accuracy. It is therefore necessary to find a balance between a protected model's accuracy and the effectiveness of the defense. We also note that in most cases, the weights extracted from deeper layers are further from the original weights.

To further explain the way our countermeasure works, we measure the Pearson correlation coefficient ρ between the extracted weights and original ones, one output channel at a time. Figure 4.4 shows the coefficients for each extracted convolutional layer, for a 1-layer protected model with standard deviation $\sigma = 0.01$. The decrease of the correlation in deeper layers supports the domino effect: it is more difficult to extract deeper layers, as the noise introduced in the first one is amplified.

To further analyse the impact per layer, we also plot the distribution of weight differences $\delta = \frac{\hat{w}_i - w_i}{||W||_2}$ between the extracted and original weights, for each of the four recovered convolutional layers. In Figures 4.5 and 4.6, we consider a MobileNetV2 [108] model protected by a 1-layer parasite with standard deviation respectively $\sigma = 0.01$ and $\sigma = 0.1$. We can make two observations based on those

Table 4.2: First simulation results on a protected MobileNetV2 model with an original accuracy of 71.41% on the CIFAR10 testing set. The noise added in the protected case is a Gaussian distribution with various standard deviations σ. The attacker uses 128 inputs to extract the weights. We log the extracted accuracy, as well as the mean $d = \frac{E[\hat{W}-W]}{||W||_2}$ between extracted weights $\hat{W}$ and original ones W over the first four convolutions.

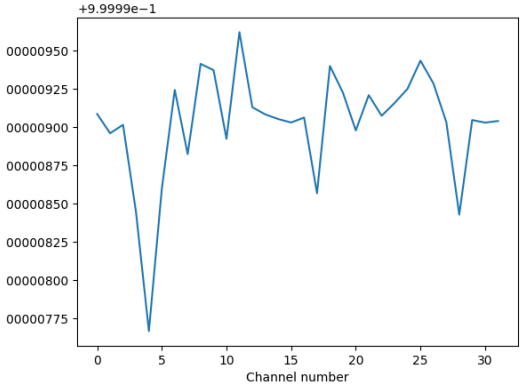
Number of parasitic layers	Standard deviation σ	New Accuracy	Extracted Accuracy
1	0.01	71.41%	71.21%
1	0.05	71.24%	69.54%
1	0.1	71.05%	12.90%
1	0.2	70.23%	18.17%
2	0.01	67.84%	11.74%
2	0.05	69.51%	10.15%
2	0.2	66.43%	9.97%

Mean Conv1	Mean Conv2	Mean Conv3	Mean Conv4
$1 \cdot 10^{-4}$	$2.1 \cdot 10^{-3}$	$5 \cdot 10^{-3}$	$3.3 \cdot 10^{-3}$
$3 \cdot 10^{-4}$	$4.9 \cdot 10^{-3}$	$6 \cdot 10^{-3}$	$2.9 \cdot 10^{-3}$
$3.1 \cdot 10^{-3}$	$1.33 \cdot 10^{-2}$	$1.79 \cdot 10^{-2}$	$1.29 \cdot 10^{-2}$
$9 \cdot 10^{-4}$	$3.6 \cdot 10^{-3}$	$1.6 \cdot 10^{-2}$	$3.6 \cdot 10^{-2}$
$1 \cdot 10^{-4}$	$1.7 \cdot 10^{-3}$	$6.9 \cdot 10^{-3}$	$2.3 \cdot 10^{-3}$
$3 \cdot 10^{-4}$	$4.8 \cdot 10^{-3}$	$9.5 \cdot 10^{-3}$	$7.6 \cdot 10^{-3}$
2.7054	$3.65 \cdot 10^{-2}$	$5.26 \cdot 10^{-2}$	$3.1 \cdot 10^{-2}$

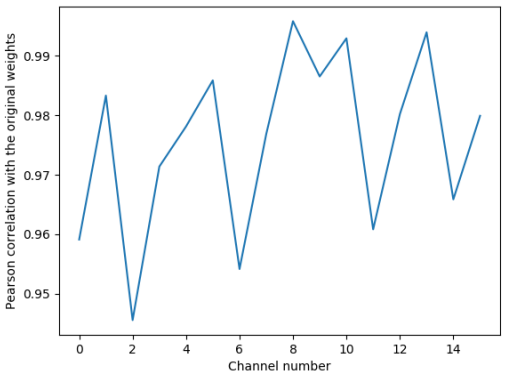
figures. First, once again, we see that recovered weights from deeper layers tend to be further away from the original ones. The second is that a higher σ leads to fewer correctly extracted weights. We can thus understand why the attack succeeds for $\sigma = 0.01$, leading to an extracted accuracy of 71.21% (almost equal to the original accuracy) but fails for $\sigma = 0.1$, with an extracted accuracy of only 12.9%.

Second simulation In Section 4.3.1, we described a second simulated attack, where the attacker tries to approximate the added parasite along with the target parameters. But in this simulation, the parasite's weights change at each run. On average, considering a set of 30 possible parasitic one-layer models, the LeNet model to which we add one parasitic layer at random has an accuracy of 70.61%. In this case, the attack on the aforementioned LeNet model leads to an extracted model with an accuracy of 11.18%.

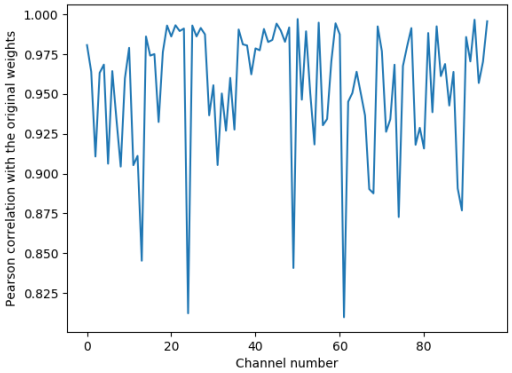
Similarly, we apply this second simulated attack to the first 13 layers MobileNetV2 with an additional parasite. On average, this also leads to an accuracy of 69.92% on the testing set and an extracted model with a very low accuracy: 10%.



(a) Convolution 1



(b) Convolution 3



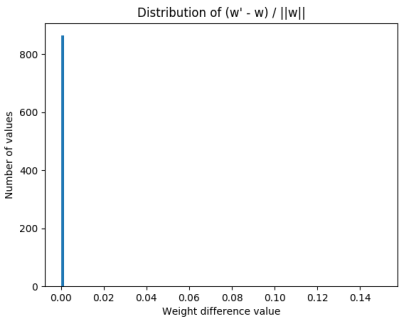
(c) Convolution 4

Figure 4.4: Pearson correlation coefficient between the extracted weights and the original ones, for the first, third and fourth convolutions in MobileNetV2. Each point in the graph corresponds to the correlation coefficient for one output channel.

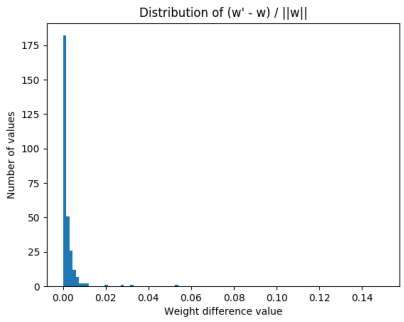
Thus, in both simulations, the attack fails, as the extracted model is unusable due to its low accuracy. This is the case despite the fact that we supposed the attacker had perfect traces and could average them over several inputs. We believe this proves the induced domino effect in the weight extraction process. Indeed, it does seem that the small error introduced in the first layer because of the small additional parasitic noise in the input is transferred to the following layers and amplified by them.

One could argue that an attacker might proceed the same way as the defender, and also train CNNs on Gaussian noise to improve the extracted weights. However, when training the parasites several times, we noticed that the resulting weights were very different from one training to the next. Thus, the attacker would still need more work to extract the correct weights.

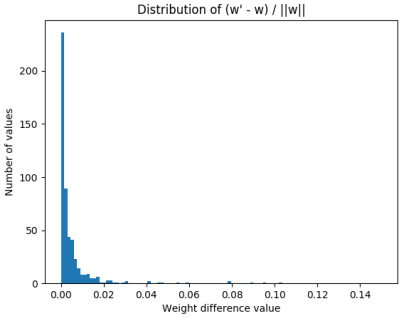
Additional computations Let us also compute the number of operations added through the parasitic models. As detailed in Section 4.2.1, one layer adds M multi-



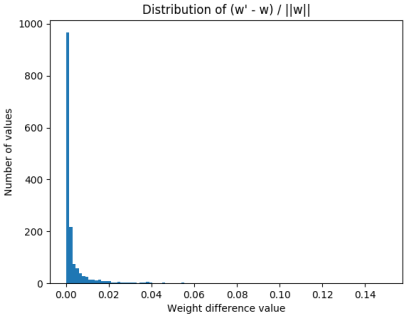
(a) First convolution (Conv1)



(b) Second convolution (Conv2)



(c) Third convolution (Conv3)



(d) Fourth convolution (Conv4)

Figure 4.5: Distribution of the weight differences $\frac{\hat{w}_i - w_i}{||W||_2}$ within each of the four MobileNetV2 convolutional layers extracted by the attacker, in the first simulation with 128 traces (see Section 4.3.1) and $\sigma = 0.01$.

plications:

$$M = c_{out} \times k \times k \times c_{in} \times n \times n$$

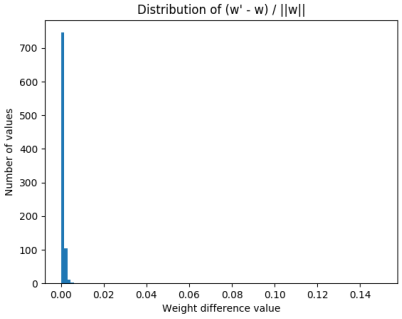
where c_{out}, c_{in} are the layer's number of output and input channels respectively, k is the filter size and (n, n) is the layer's input size. We also explained that here, we feed one channel at a time to the parasitic models.

In Section 4.3.1, we explain that, for the parasites, we consider models with various layer numbers.

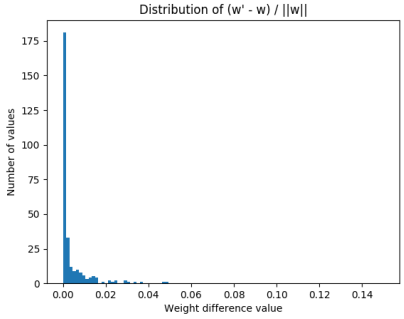
The LeNet architecture can be considered small compared to the ones used nowadays in the image processing field. Adding one parasitic layer should therefore already be noticeable in the total number of multiplications. For the one convolutional layer model described in Section 4.3.2, we have: $c_{out} = 1$, $c_{in} = 1$, $k = 5$ and $n = 32$. We also know that the inputs from the CIFAR10 dataset have 3 input channels. Ignoring the much less time consuming Batch Normalization layer, this amounts to:

$$M = nb\_input\_channels \times c_{in} \times c_{out} \times k \times k \times n \times n \qquad = 76,800 \qquad (4.1)$$

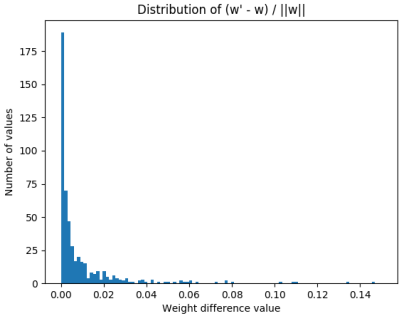
Thus, the one-layer model adds 76,800 multiplications to the original LeNet model. But for inputs of size $(32, 32, 3)$, LeNet requires 658,000 MAC (Multiply-Accumulate) operations. Thus, one such parasitic layer represents an 11.7% addition in the multiplications. Let us note that the tests carried out on LeNet were mainly for a proof-of-concept, as LeNet is no longer a standard model for image processing.



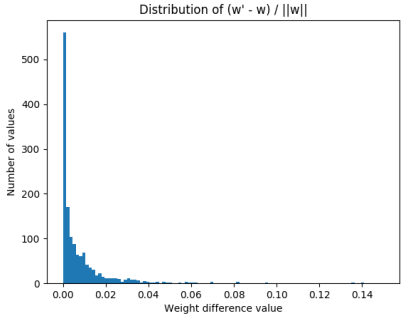
(a) First convolution (Conv1)



(b) Second convolution (Conv2)



(c) Third convolution (Conv3)



(d) Fourth convolution (Conv4)

Figure 4.6: Distribution of the weight differences $\frac{\hat{w}_i - w_i}{||W||_2}$ within each of the four MobileNetV2 convolutional layers extracted by the attacker, in the first simulation with 128 traces (see Section 4.3.1) and $\sigma = 0.1$.

MobileNetV2 is much larger than LeNet, and we can therefore consider larger parasites. For the two-layer model described in Section 4.3.2, we have, for the first layer:

$$\begin{aligned} c_{out}^{(1)} &= 2 \\ c_{in}^{(1)} &= 1 \\ k^{(1)} &= 5 \\ n^{(1)} &= 32 \\ c_{out}^{(2)} &= 1 \\ c_{in}^{(2)} &= 2 \\ k^{(2)} &= 5 \\ n^{(2)} &= 32 \end{aligned}$$

Ignoring the much less time consuming Batch Normalization layer, this amounts to:

$$\begin{aligned} M_{tot} &= nb\_input\_channels \times (c_{out}^{(1)} \cdot k^{(1)} \cdot k^{(1)} \cdot c_{in}^{(1)} \cdot n^{(1)} \cdot n^{(1)} && (4.2) \\ &\quad + c_{out}^{(2)} \cdot k^{(2)} \cdot k^{2} \cdot c_{in}^{(2)} \cdot n^{(2)} \cdot n^{(2)}) && (4.3) \\ &= 307,200 && (4.4) \end{aligned}$$

MobileNetV2 trained on the CIFAR10 dataset requires 6.32 millions MAC operations. Thus, the additional 307,200 multiplications only represent 4.9% of the original MAC operations.

4.4 Discussions

In this Section, we discuss the limits and possible improvements to our countermeasure. First, in Section 4.4.1, we consider the number of traces an attacker needs to thwart our protection. Increasing the entropy in the set of parasitic models would leverage more work for the attacker. Section 4.4.2 discusses how to achieve such a higher entropy. Finally, Section 4.4.3 compares the overhead incurred by our method to common cryptographic countermeasures.

4.4.1 Number of Traces to Recover the Weights

Since our protection consists in adding randomness, an attacker who gathers multiple traces should be able to approximate the correct weights. The question that remains is the number of traces necessary for an almost exact extraction.

For the first simulation, 1,024 traces already enable an attacker to increase the extracted model's accuracy from 11.74% with 128 traces to 71.19% for a mobileNetV2model protected by a two-layer parasite with $\sigma = 0.01$. But increasing σ still enables a protection: for instance, with $\sigma = 0.05$, the extracted accuracy is 17.12% with only a slight drop in the original accuracy: 69.51% instead of 71.41%. It is therefore paramount to find a balance between the standard deviation, the number of parasitic layers, the drop in the model's accuracy and the effectiveness of the protection.

4.4.2 Increasing the Entropy of the Added Noise

To maximize the efficiency of the protection, it is important to have a large entropy in terms of the possible parasitic architectures. One way to increase the said entropy is to consider selecting either one or several CNNs at random instead of only one. These can then be applied to several parts of the input, as long as all the neurons are affected by at least one model.

Thanks to the randomness in the parasites' architectures, we believe that our protection would be particularly advantageous when extending it to the black box context – and [10]'s original threat model – where the attacker does not know the model's architecture, and which is enforced by Chapter 3. This makes the parasitic set's entropy higher.

4.4.3 Comparing to Common Countermeasures

The authors of CSI NN [10] consider two possible SCA countermeasures to thwart the parameter extraction: masking and shuffling independent operations. In both cases, they mention that these would lead to a large overhead.

According to the authors of [10], masking each neuron at each iteration should prevent the attack. However, masking a multiplication is expensive: it is at least twice the cost of the initial multiplication, as can be seen in Table 1 of [11]. Since masking needs to be applied to all the multiplications, the total number of multiplications in one run would double. In our case, our overhead only consists in a small percentage of the original number of multiplications. More importantly, as explained in Section 2.3.7, the authors of [31] note that since the masking is not computed in an arithmetic field, a bias is incorporated due to the sign. Other methods are therefore necessary to fully mask the parameters. The authors of [31] introduce hiding techniques for the sign bit, while the authors of [30] take the masking further. The authors of [29] consider completely changing the way BNNs are computed to accommodate for arithmetic field masking. In all cases, the overhead goes beyond the basic masking techniques.

Shuffling is also an expensive operation. With the Fisher-Yates algorithm, if there are n elements to swap, then the algorithm runs in $O(n)$ but requires generating n truly random numbers [34]. Here, n is the number of operations which can be permuted. In an FC layer with n_x inputs and n_w weights, $n = n_x \times n_w$. In the convolutional case, supposing a stride of 1, an input shape (n_x, n_y, c_{in}) and a weight shape (k, k, c_{out}), there are $n = c_{out} \times (n_x - k + 1) \times (n_y - k + 1)$ convolutional operations to shuffle. Since random number generation is time consuming [10] and needs to be applied to all layers, we believe that our countermeasure should also generate less overhead.

4.5 Conclusion

This Chapter focuses on mitigating physical SCAs aiming at extracting a model's weights and biases. As in Chapter 3, we make use of additional dummy layers called parasites. But while in Chapter 3, we wished to change a model's structure by adding dummy *ReLU* activation functions, our goal in this Chapter is to obfuscate a model's input. The focus in this context is the dynamic addition of our dummy layers: because the input is obfuscated with changing noise at each run, the attacker struggles to recover precise weights for the first layer. This first error is then amplified by the following layers, rendering the extracted model inaccurate.

To comfort our claims, we realise two simulated attacks on a LeNet and a MobileNetV2 architectures, where the attacker is supposed to have exact traces. The attack fails in both cases (the extracted model has an accuracy around 10%). While the attack succeeds with enough traces in some cases, the defender can play with the standard deviation of the additional noise to make the model more robust. We also suggest other approaches to improve our defense in Section 4.4.

Even if the parasites incur an overhead, we explain in Section 4.4.3 that they appear to be less time-consuming than common cryptographic measures.

While Chapter 3 and the current Chapter leverage parasitic models to counter reverse-engineering attacks, we observe in Chapter 3 the impact that the current change of structure has on adversarial samples. In the wake of this observation, the next Chapter proposes a way to thwart adversarial attack through the introduction of a different type of parasite.

Chapter 5: Application to Adversarial Examples

While this thesis focuses on protecting NNs against reverse-engineering attacks, one of the motivations to do so is to limit the possibility of other attacks, such as adversarial ones. In Chapter 3, we explain that parasitic models also impact adversarial samples. Therefore, we follow up with that observation in this Chapter, and show that we can leverage parasitic models to mitigate adversarial examples.

So far, we have only exploited two aspects of the parasites:

- Additional layers make a model's internal structure more complex.
- Dynamically selecting parasites among a set of pretrained models and adding them at the entrance of a model leverages randomness in the input values. This, in turn, mitigates weight extraction physical SCAs.

We can mix the two techniques to tackle adversarial attacks. Indeed, the latter rely on the internal structure of a model to generate adversarial samples. However, simply changing the architecture does not make the adversarial search more complex. It only modifies the generated samples. We believe that a constantly changing architecture leads to the need to find transferable examples, and therefore makes it harder on an attacker. We show the efficiency of our proposal in this Chapter.

First, we introduce the considered threat scenario and give an overview of our proposal in Section 5.1. We then detail our proposal and justify the use of dynamic autoencoders in Section 5.2. After showing our results in Section 5.3, we discuss them in Section 5.4 and conclude this Chapter in Section 5.5.

5.1 Threat Scenario and Defense Overview

Adversarial attacks can be either black-box when the attacker has no prior knowledge about the architecture, white-box when the architecture and parameters are known, or gray-box when the attacker has access to part of the information. The best known attacks rely on the use of gradients, which require access to the weights.

Threat Scenario In this Chapter, we consider a remote gray-box scenario:

- The attacker knows the base model's architecture and parameters

- The first point implies that the attacker can compute gradients on the parameters, and therefore launch gradient-based adversarial attacks.
- The attacker has query access: she can query the model with crafted inputs and get the model's output.

As will be further detailed in Section 5.3, we either consider that the attacker only has access to the base model, or she also knows we have introduced a countermeasure.

Based on these assumptions, the attacker's goal is, given an input x, to generate an adversarial sample $x' := x + \epsilon$ such that the small noise ϵ is unnoticeable to the human eye but fools the base model.

More specifically, we focus on the standard evaluation from one library of attacks, AutoAttack [25].

Protection Overview Once again, the parasites introduced in Chapter 3 come in handy. As explained in that Chapter and in [111, 91], adversarial examples are based on classification boundaries, which are a result of the internal structure. As observed in Chapter 3, a change in the internal structure goes hand in hand with a modification of the adversarial examples. As detailed in Section 2.3.7, autoencoders have been used to deflect adversarial examples [9]. Indeed, they can keep the important features and eliminate unwanted noise. Based on these assumptions, we propose the following scheme:

- Train a set of parasitic autoencoders S to approximate a noisy identity.
- At each run, select one or several parasite(s) in S.
- Select one or several location(s) in the base model, where we place the selected models. Let us note that in this Chapter, we only consider the case where these autoencoders are placed at the entrance of the base model. But they can be placed after any layer as long as the accuracy does not drop much.

The change in the structure that the autoencoders bring, along with the dynamism of the method, ensure that a potential attacker would struggle in her search for adversarial samples. We detail our methodology in Section 5.2 and test our defense in Section 5.3.

Related Works Various protections against adversarial examples have been proposed so far. We detail them in Section 2.3.7. Denoising autoencoders [9] are the closest art to our technique. While we also consider incorporating autoencoders as a countermeasure, our proposal differs from denoising autoencoders in two ways:

- Instead of training our autoencoders on noisy inputs, we add noise to the labels (see Section 5.2.1).
- We consider a dynamic approach, so as to add randomness and protect the base model even when the attacker knows an autoencoder has been introduced (see Section 5.2.3).

5.2 Dynamic Parasites

In this section, we explain and detail our proposed defense.

5.2.1 Autoencoders for a Noisy Identity

Chapters 3 and 4 introduce the notion of parasites approximating a noisy identity. A parasitic model P is trained to return, for a given input x, $P(x) = x + \mathcal{N}(\sigma)$ where $\mathcal{N}$ is a Gaussian noise with standard deviation σ.

Training While the parasites used in this defense undergo the same training, they are of a different kind. Autoencoders [51] are NNs whose aim is to approximate the identity under constraints. They are comprised of an encoding phase e and a decoding phase d. After its training, an autoencoder A should therefore be such that $f(e(x)) \approx x$ for all x in the considered dataset. They have often been used for compression (with dimensionality reduction) or feature learning [42]. Indeed, they tend to have hidden layers with lower dimensions than the input and output layers, giving them a 'diabolo' shape. In that case, they are called *undercomplete* autoencoders. For instance, when the decoding of an undercomplete autoencoder is linear and the loss function used is the mean squared error (i.e. the ℓ_2 norm between the current output and the label), then the coding phase is similar to applying a Principal Component Analysis (PCA). This example explains why autoencoders can be used for feature extraction or compression.

In fact, data distributions are not random: they lie within a lower dimension manifold. This is why undercomplete autoencoders are generally able to learn compressed versions of the data: only part of the input is necessary for predictions.

Since their appearance decades ago, much progress has been made on autoencoders and their use. One particular use is interesting in our case: denoising autoencoders learn to eliminate noise. A model A is then trained on a training set X as follows:

- For $x \in X$, corrupt x to get $\tilde{x} = x + \epsilon$ for some noise ϵ.
- This forms the corrupted set $\tilde{X}$.
- Train the model with training set $\tilde{X}$ and labels X.

Our goal when training our parasitic autoencoders is, in a sense, the opposite of the denoising task. Instead of having $\tilde{X}$ as the training set and X as the labels, we proceed the other way around: X is the training set and $\tilde{X}$ are the labels. The reason for this difference is that we wish to introduce noise, so as to ensure a change in the base model's structure. Moreover, the different layers within the base model do not react similarly to noise. Since we can optionally select different locations in the model to add the autoencoder, it is easier – for the sake of a better adaptation – to train the model with a noisy label rather than a noisy input. Thus, the input is perturbed thanks to two aspects:

- The compression produced by the encoding phase in undercomplete autoencoders only keeps the important features. The decoding phase then generates details different from the original ones to get the input back to its initial dimension.
- We train the autoencoder to add Gaussian noise to the identity.

Architecture While autoencoders can be trained with only one hidden layer – i.e. only one layer besides the input and output ones –, depth presents four main advantages:

- Just like any DNNs, depth yields the same advantages for autoencoders as those mentioned in Section 2.2.2. Namely, complex tasks require large numbers of neurons, and even though one layer should be enough according to the universal approximator theorem, depth makes the implementation more realistic.
- Shallow networks do not leave room for additional constraints as required for many autoencoders.
- Approximating some functions takes less computational time with deeper autoencoders [42].
- Most often, deeper autoencoders require exponentially less training data [42].
- In practice, depth contributes to better compression performances [50].

Since we wish to maximize the change in the architecture without the base model's accuracy dropping much, we opt for a deep undercomplete autoencoder to compress the input data. The autoencoders we consider in our experiments are described in Section 5.3.

As explained in the two previous Chapters, approximating the identity is an easy task for CNNs [148]. Moreover, adding a parasite to a model alters its internal structure, especially thanks to the nonlinear activation functions. We exploited this change in Chapter 3 to thwart mathematical attacks relying on the piece-wise linearities of functions such as $ReLU$. However, in this Chapter, we do not limit ourselves to the introduction of hyperplanes to make attacks more complex. Indeed, new hyperplanes made the attacker in [16] struggle more because it added work to its generally time consuming sign recovery step. In the adversarial case, deeper – or larger – models are still easily targeted. The goal in this Chapter is to identify the important features as well as introduce noise thanks to the aforementioned autoencoders, and to ensure constantly changing samples through the dynamism of the parasites. Autoencoders trained differently, while still able to correctly compress the input data, lead to different changes to the structure. Indeed, they have various ways of extending the compressed data back to its initial shape. Then, if M_A^l denotes the base model to which we add an autoencoder A at location l, adversarial examples generated for M_A^l do not necessarily transfer to $M_{A'}^l$. We show this last statement in Section 5.2.2.

5.2.2 Transferability

As previously, let M be our base model, and M_A^l denote the base model to which we add an autoencoder A at a certain location l. The constraints put on autoencoders through their architecture, and the additional noise in the training labels lead to a model M_A different from that of M. Because the elements of S are trained differently, the various M_A models with $A \in S$ also have various structures, and lead to hardly transferable adversarial samples. The first step to ensure the efficiency of our countermeasure is to confirm this lack of transferability between M_A^l and $M_{A'}$.

With that aim, we train a set S of 14 autoencoders to approximate a noisy identity. The autoencoders are trained on a Gaussian noise with various standard deviations (between 0.008 and 0.15). In this experiment, we only consider adding autoencoders after the input layer, as in Figure 5.1. The base model is a PyTorch implementation of WideResNet [147]. Let M_A be the base model preceded by autoencoder A. We generate 200 adversarial examples for each M_A, where $A \in S$. Let x_A be the adversarial examples associated to M_A. For each $A \in S$, we test the accuracy of all $M_{A'}, A' \in S$ against x_A. For $1 \leq i \leq 13$, we then compute the ratio of elements in x_A that are no longer adversarial for at least i models in S. For instance, for $i = 2$, we wish to determine the number R of elements in x_A that are not adversarial for at least 2 models among the 13 considered. In other words, those R elements are adversarial for M_A, but there are at least two models M_{A_1} and M_{A_2} which output the original, correct labels for those samples.

Let us note that we give the details about the autoencoder and the base model's architectures and training process in Section 5.3.

Table 5.1 shows the results of the aforementioned experiment. We see that around 30% of adversarial samples for a model M_A are no longer adversarial for at least 8 other protected models. This high ratio is proof that adversarial samples generated with one autoencoder are rarely transferable to other protected models. It also means that if an attacker generates an adversarial example on model M_A for a certain $A \in S$, it is unlikely that it will also fool model $M_{A'}$.

We base our countermeasure on the lack of transferability observed in Table 5.1, as explained in Section 5.2.3.

5.2.3 Dynamism

Because they tend to eliminate unnecessary details, autoencoders have been previously proposed as a countermeasure to adversarial attacks [9]. However, an attacker who knows the existence of the autoencoder can still adapt its attack to the new architecture. This denoising technique has been attacked by Carlini and Wagner's transferable examples [18].

The fact that adversarial samples generated with AutoAttack on one model M_A^l are unlikely to be transferable to another model $M_{A'}^l$ justifies the dynamic use of the set S in the inference phase of the base model M. Every time an input is fed to the base model M, we select an autoencoder $A \in S$ at random. While in Section 5.3, we only present the case where an autoencoder is associated to a given location, the location l can also be selected at random.

Table 5.1: For each model $M_A, A \in S$, we generate adversarial examples x_A. We then compute, for $1 \leq i \leq 14$, the ratio of elements from x_A that are no longer adversarial for at least i models. S is our set of 14 pretrained autoencoders. M_A is the base model to which we add autoencoder $A \in S$ at the entrance. The base model has an accuracy of 94.5%. The adversarial samples were generated on 200 elements from the CIFAR10 testing set [71], using the standard evaluation of AutoAttack [25]

	Accuracy of M_A	i=1	i=2	i=3	i=4	i=5	i=6
Autoencoder 1	90.95%	68%	67.5%	52.5%	46%	44%	41.5%
Autoencoder 2	90.46%	68.5%	57%	52%	49.5%	42.5%	39.5%
Autoencoder 3	89.82%	69.5%	60%	53%	49%	47%	43%
Autoencoder 4	90.16%	75.5%	68%	61.5%	57%	53.5%	48.5%
Autoencoder 5	90.06%	72%	65%	57%	51%	49%	44.5%
Autoencoder 6	90.36%	69%	61.5%	54.5%	50.5%	45%	42%
Autoencoder 7	90.06%	72%	60.5%	55.5%	51%	46%	44.5%
Autoencoder 8	90.51%	74.5%	65.5%	56.5%	53%	49.5%	45.5%
Autoencoder 9	89.67%	73%	65%	59.5%	55.5%	51%	48.5%
Autoencoder 10	90.79%	72%	64%	53.5%	48.5%	44.5%	41%

	i = 7	i=8	i=9	i=10	i=11	i=12	i=13
Autoencoder 1	39.5%	34%	31.5%	27.5%	29%	24%	18%
Autoencoder 2	34%	30%	28.5%	26.5%	23%	18.5	13%
Autoencoder 3	40.5%	36.5%	34%	29.5%	25.5%	22%	17.5%
Autoencoder 4	44.5%	41%	36.5%	35%	33%	29.5%	24%
Autoencoder 5	40.5%	38%	33.5%	30%	26%	23.5%	16.5%
Autoencoder 6	40%	36.5%	33.5%	31%	26.5%	24%	20%
Autoencoder 7	40.5%	37%	34%	31%	27%	23.5%	20%
Autoencoder 8	42.5%	39.5%	38.5%	37%	34%	30.5%	23.5%
Autoencoder 9	46.5%	43.5%	40.5%	37.5%	35.5%	32.5%	25.5%
Autoencoder 10	39.5%	38%	35%	31%	27%	25.5%	19.5%

Because the model is constantly changing at each run, it becomes difficult for a potential attacker to carry out an adversarial attack. Indeed, even if she knows that an autoencoder was added to its architecture, and considers one such protected model to generate an adversarial sample, then it is highly probable that the generated example would not fool the dynamic protection.

The proposed countermeasure therefore consists in the following steps:

- Let X be a training set for the autoencoders. For instance, X can be the same set as the inputs of the base model, or the output of one of the base model's layers. The second example is useful in the case where the autoencoder is to be placed in the middle of the base model's architecture. We generate the corrupted sets $\tilde{X}_\sigma$ for various values of σ such that for each batch $x_B \in X$, the corrupted batch is $\tilde{x_B} = x_B + \mathcal{N}(0, \sigma)$ where $\mathcal{N}(0, \sigma)$ is a centered Gaussian noise with standard deviation σ.

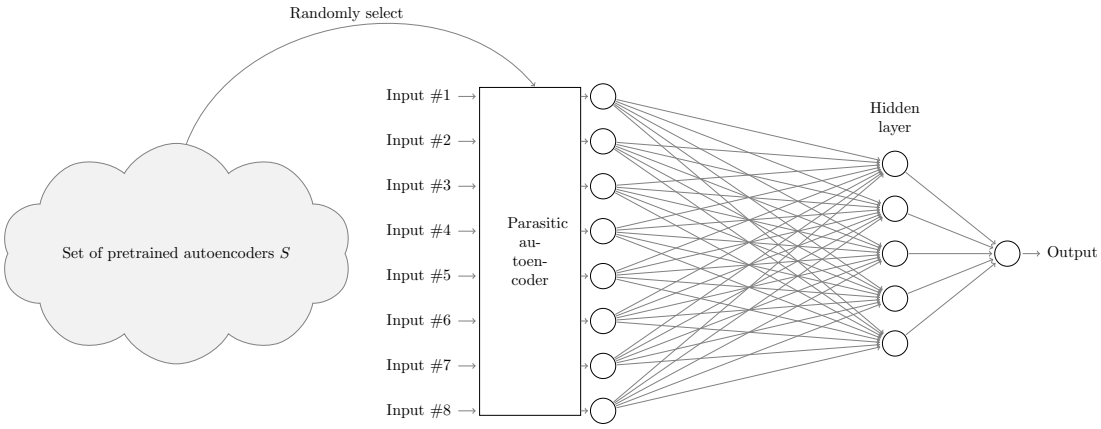


Figure 5.1: At least one autoencoder is selected from the pretrained set S, and is then incorporated in the model. In our experiments, we place them at the entrance of the model.

- Train a set S of undercomplete autoencoders. For each autoencoder A, select a standard deviation σ and train A on $(X, \tilde{X}_\sigma)$, where X is the training set and $\tilde{X}$ is its corrupted version.
- At each run of the protected model, select one or several parasitic autoencoders in S. Optionally, we can also select one or several locations within the base model as well. Let us note that to be able to select the location l, the chosen autoencoders should be adapted to the task. For instance, they can be trained on normalized inputs, or directly on the outputs of the previous layer $l-1$. Moreover, the autoencoders in our experiments do not have FC layers so as to be able to adapt to any input shape (see Section 5.3).
- The selected autoencoders are placed at the considered locations. It is important to note that this newly created model M' should not come with a substantial drop in the accuracy on the dataset X. The generation of the autoencoders' set is a sensitive task. The scheme is represented in Figure 5.1.

The dynamic selection of parasitic autoencoders along with their constrained training constitutes a protection against adversarial attacks, as a potential attacker would struggle to find an adversarial sample tailored to the current model.

5.3 Experiments

We have established the lack of transferability of adversarial examples, and detailed our proposed scheme in Section 5.2. In this Section, we test our countermeasure against the standard evaluation of AutoAttack.

Table 5.2: Architecture of the autoencoders. The inputs are assumed to have 3 input channels. `Conv2D`(ch, f, s) is a convolutional layer with ch output channels, filter size $f \times f$ and stride s. `ConvTranspose2D`(ch, s) is a transposed convolutional layer with output channels ch and stride s. Transposed convolutional layers compute an upsampling operation. We select its parameters so that it generates an output of shape $(s \cdot n \times s \cdot n)$ for an input of shape $n \times n$.

Encoder Layers
Conv2D(16, 3, 2)
BatchNormalization
ReLU
Conv2D(32, 3, 1)
BatchNormalization
ReLU
Conv2D(32, 3, 2)
BatchNormalization
ReLU
Conv2D(64, 3, 1)
BatchNormalization
ReLU
Conv2D(64, 3, 2)
BatchNormalization
ReLU

Decoder Layers
ConvTranspose2D(64, 2)
BatchNormalization
ReLU
ConvTranspose(32, 2)
BatchNormalization
ReLU
ConvTranspose(16, 2)
BatchNormalization
ReLU
ConvTranspose(3, 2)

5.3.1 Experimental Settings

Our aim with the autoencoders is to ensure that while dynamically changing the internal structure of the base model, they do not significantly drop its accuracy. We therefore opt for a deep architecture, described in Table 5.2. Some autoencoders have one additional convolutional layer $Conv2D(128, 3, 2)$ followed by a BN layer and a $ReLU$ activation.

We train the autoencoders on the CIFAR10 [71] dataset. The training is done by batches. Given a standard deviation σ, for each epoch, we generate a random Gaussian noise $\mathcal{N}(0, \sigma)$. Then, for each bacth X_B, the label is $X_B + \mathcal{N}(0, \sigma)$.

In our experiments, we place the autoencoders at the entrance of the base model. This means that for input x, base model M and autoencoder A, the protected model computes $M(A(x))$.

Once the set S of autoencoders is ready, there are various ways of randomly selecting one autoencoder in S during the inference phase. For each input x fed to the protected model, we compute a hash value for x and use the autoencoder corresponding to it. The hash function we settle on is SHA1 [33]. In other words, with N elements in S, we compute, for each input x, $i = hash(x) \mod N$. We then place the $i - th$ autoencoder at the entrance of the base model.

The base model we consider is the standard one for the CIFAR10 dataset in RobustBench [24], when considering attacks for the ℓ_2 norm. It is a type of ResNet called Wide ResNet [147], with accuracy 94.5% on the CIFAR testing set.

In the following experiments, we launch the standard execution of AutoAttack [25] on 200 elements from the CIFAR10 dataset. It consists in successively applying an untargeted then a targeted PGD attack, before moving on to a targeted FAB attack and finally a Square attack (see Section 2.3.6 for details of the attacks). In this evaluation, the maximum perturbation is $\epsilon = 0.5$.

While various versions of the PGD attack are implemented in AutoAttack, the one used in the standard evaluation is based on the cross entropy (CE) loss. This loss is used for classification tasks. Given input x, correct label y and current guessed label $\hat{y}$, CE for a task with C output classes is defined as:

$$CE(x, y) = -\sum_{i=1}^{C} y_i \cdot log(\hat{y}_i)$$

5.3.2 Results

We consider two experiments to measure the efficiency of our countermeasure. In both cases, we call M the base model and M_S the model protected with our dynamic defense.

- In the first, the attacker has access to M's architecture and parameters, but does not know about the parasites' existence. She therefore generates adversarial examples x_{adv} on 200 elements from the CIFAR10 dataset thanks to AutoAttack. We then test the accuracy of M_S against x_{adv}.
- In the second, the attacker is aware that an autoencoder has been added to the model. She therefore generates samples x_{adv}^A to fool M_A for a certain $A \in S$. We then test the accuracy of M_S against x_{adv}^A.

First experiment Once the adversarial examples against M have been generated, we try a dynamic defense with an autoencoders set of length N varying between 1 and 13. In this first experiment, all autoencoders are placed at the entrance of the model. The results are presented in Table 5.3. We see that one autoencoder is enough to mitigate an adversarial attack when the attacker is not aware of the defense. This confirms the efficiency of previously introduced autoencoder defenses [9]. Table 5.3 also shows that when the attacker is oblivious to the parasitic autoencoder, it is not necessary for the defense to be dynamic. Once again, this aligns with the previous defenses that only rely on one fixed autoencoder.

However, an attacker can thwart such an attack by training an autoencoder A on her own and generating adversarial samples against the compound model M_A. The following experiment shows the efficiency of our countermeasure in such a scenario.

Second experiment We present the results of the previously explained second experiment on 10 autoencoders. For each $A \in S$ where S has length $N = 10$, we generate adversarial examples x_A. We then test the accuracy of the dynamically

Table 5.3: We measure the robustness of our protected model in the case where the attacker knows the original architecture and parameters. She generates adversarial samples using the standard settings for AutoAttack on the CIFAR10 dataset for norm ℓ_2. We test the accuracy of protected models with $1 \leq i \leq 13$ autoencoders in the parasitic set S. The initial accuracy of our protected model is over 90% in all cases.

N autoencoders in S	Initial Accuracy	Robustness
1	90.46%	80%
2	90.82%	78.5%
3	89.16%	79.5%
4	90.06%	77%
5	90.36%	78%
6	90.06%	80%
7	90.51%	79.5%
8	90.67%	78.5%
9	89.61%	77.5%
10	90.04%	81%
11	90.79%	77.5%
12	90.52%	78.5%
13	89.73%	78%

Table 5.4: For each autoencoder $A \in S$, we generate adversarial examples x_A on M_A, thanks to AutoAttack. We only present the results for the first 10 autoencoders, as the other autoencoders present similar ones. We then test the accuracy of the protected model M_S, with $N = |S|$ varying between 1 and 13.

	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9	A_{10}
$N = 1$	0%	37 %	45%	50.5%	39.5%	42%	45.5%	52.5%	52%	46.5%
$N = 2$	14%	21%	42.5%	46.5%	41.5%	43.5%	38.5%	52.5%	52%	48.5%
$N = 3$	26%	25%	31%	48.5%	43.5%	47%	39.5%	47.5%	49%	50%
$N = 4$	29.5%	26.5%	30%	35.5%	38%	44.5%	40.5%	47%	49.5%	48%
$N = 5$	33%	31%	38.5%	41%	29.5%	42%	38%	47.5%	50.5%	52%
$N = 6$	31%	30%	36%	43%	36.5%	38%	39%	43%	52.5%	47%
$N = 7$	33.5%	26.5%	30.5%	35.5%	35%	39%	34%	46.5%	55%	50%
$N = 8$	34%	35.5%	32.5%	40.5%	37%	41%	36%	40%	52.5%	47%
$N = 9$	36.5%	28%	39%	39.5%	36.5%	41.5%	40%	38.5%	46%	47.5%
$N = 10$	35%	34.5%	41%	43%	41%	42%	38%	41%	49%	46%
$N = 11$	40%	35.5%	38%	43.5%	30.5%	38.5%	38.5%	43.5%	46%	35.5%
$N = 12$	37.5%	33%	38%	46%	37.5%	42%	39.5%	41.5%	44.5%	40%
$N = 13$	39.5%	33.5%	37%	43%	38%	40.5%	37%	44%	44%	40%

protected model against x_A, in the case where the autoencoders are placed at the entrance of the model.

Table 5.4 shows that dynamically adding autoencoders ensures at least a 40% accuracy when there are enough models in S. We also note that this threshold is increased to at least 50% when we allow the accuracy to drop further (around 85%). Let us note that the case where $N = 1$ often leads to a higher robustness because the selected autoencoder is different from the one used to generate the adversarial examples. This cannot always be ensured. The dynamism enables us to avoid the case where the attacker uses the correct autoencoder, as is the case in the first cell of Table 5.4.

Let us also note that the autoencoder A used to generate the adversarial examples is in the set of possible autoencoders S. This explains why in all columns, we observe a drop in the accuracy – that is counterbalanced with enough other models: the drop appears when A is among the models the defender can dynamically select.

Thus, this second experiment shows that our countermeasure mitigates attacks where a malicious user trains an autoencoder to generate adversarial samples. While knowing the full architecture enables an attacker to fool the model with or without an autoencoder, a changing architecture makes her task harder.

5.4 Discussion

Because we use RobustBench's standard evaluation, we can compare our countermeasure to their leaderboard. Two defenses in their leaderboard use the same WideResNet model as we do. When the attacker is unaware of the attack, our robustness is similar to that of the best model in the leaderboard [99]. Indeed, the authors of [99] propose a defense with a 78.8% accuracy, which is similar to the accuracies we get in Table 5.3.

In our second experiment, we have a robustness of over 40% with almost no drop in the accuracy with autoencoders at the entrance. We can increase this robustness to 50% if we are more lenient about the accuracy. The robustness might also be increased by varying the locations of the autoencoders. These robustness values are lower than the two models listed in the leaderboard [99, 104]. However, both those techniques rely on adversarial training. While, as explained in Section 2.3.7, adversarial training is the best method known so far, it requires retraining the model on generated adversarial samples. Both the training and the adversarial attacks take time, making adversarial training time consuming. In our defense, on the other hand, the autoencoders do not require any adversarial sample generation. Moreover, we do not need to train them along with the model: we only need to optimize the small parasites' parameters.

In Section 2.3.7, we also mention that autoencoders have already been used to mitigate adversarial attacks [9]. The authors of [9] use sparse denoising autoencoders to protect models against various adversarial attacks. While this countermeasure shows a robustness equivalent to ours in the gray-box context where the attacker does not know about the defense, the authors do not show the results for the white-box case on the CIFAR10 dataset. Moreover, our proposal is complementary to that in [9]. Instead of training simple undercomplete autoencoders, our parasitic set

can be comprised of sparse denoising autoencoders. We believe that applying our scheme to the sparse denoising autoencoders in [9] could improve their robustness in the case where the attacker knows an autoencoder has been introduced in the original model.

5.5 Conclusion

In this Chapter, we pretrain a set of parasitic autoencoders, and exploit a dynamic introduction of those parasites in order to thwart gradient-based adversarial attacks.

We carry out two experiments to show the efficiency of our countermeasure against the standard evaluation of AutoAttack on the CIFAR10 dataset. In the first, the attacker has access to the target model but does not know about the defense. In the second, the attacker knows that an autoencoder has been added to the model. In both cases, the robustness of the model is increased thanks to our countermeasure.

This Chapter suggests that the dynamic use of autoencoders is a promising direction when it comes to tackling adversarial examples.

After proposing the insertion of parasitic models to thwart parameter extraction reverse-engineering attacks in Chapters 3 and 4, we show in this Chapter that the proposal can be adapted to mitigate adversarial attacks. But all the attacks and defenses considered so far were set in a gray-box context, as they require prior knowledge about the model's architecture. In the following part, we directly aim at protecting a secret model architecture in a black-box context.

Part II

Preventing the Black-Box Reverse-Engineering of the Architecture

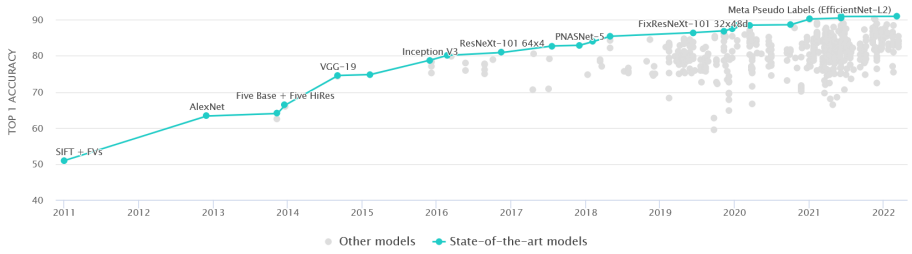


Figure 5.2: Evolution of the accuracy of image classification on the ImageNet dataset [70]. So far, the best model, Model Soups, achieves 90.94% accuracy. The metric used to measure the accuracy is the percentage of correctly labeled images. This graph is taken from `https://paperswithcode.com/sota/image-classification-on-imagenet`

Overview

Although the parameters of the model are what enable gradient-based attacks such as [91], an attacker cannot figure out the weights without recovering the architecture as well. Thus, one way of protecting the weights and biases of a model is to obfuscate its architecture.

We state in part Part I that for common tasks, it is realistic to assume that the attacker has access to the architecture. However, for sensitive tasks such as biometrics and facial recognition [13], or very specific ones, the architecture can be kept a secret. Finding the correct architecture for the task at hand can be an arduous task. Too deep architectures may overfit more easily, while shallow ones might not reach a high enough accuracy. Other hyperparameters such as skips (see Section 2.2.1) can improve a model's learning. The importance of the architecture selection can be seen in Figure 5.2, where we see the evolution of the accuracy on the ImageNet [70] dataset over the years.

Moreover, even though usual tasks generally require common architectures, there are various families of architectures that could be selected. In fact, even within one family of architectures – such as VGG or ResNet –, the numerous parameters make it almost impossible for an attacker to determine the exact architecture. The authors of [141], for instance, state that the initial set of possible architectures for the VGG16 architecture comprises over 5.4×10^{12} architectures.

Furthermore, knowing the model already provides some information that an attacker might exploit to carry out other types of attacks [64, 113].

Thus, despite it appearing as a trivial security aspect, it is also paramount to ensure the architecture's protection.

To the best of our knowledge, so far, most reverse-engineering attacks that aim at recovering the architecture of a target model rely on the sequential computation of NNs. Based on this, we propose, in this Part, a novel way of carrying out inference computations, mitigating the architecture recovery attacks published to this day.

Chapter 6: Protecting CNN Architectures

So far, most architecture extraction attacks rely on the sequential execution of NNs. CSI NN [10] counts on that fact to iteratively determine the number of neurons per layer. The authors of [53] and [52] monitor the call to individual layers through the cache to reconstruct the model. Cache Telepathy [141] targets matrix multiplications corresponding to FC and convolutional layers in order to determine a model's hyperparameters. In all cases, the attacker trusts the model to compute its predictions layer by layer. Based on this observation, we introduce Telepathic Headache, a countermeasure to cache-based architecture extraction attacks against CNNs.

In this Chapter, we first introduce the threat scenario and protection overview in Section 6.1. Then, we explain how CNN computations can be reordered in Section 6.2. Section 6.3 adds randomness to the reordering to make attacks more complex. Section 6.4 summarizes the full scheme of our defense. The results, including a theoretical analysis of the scheme's security, are presented in Section 6.5.2. Finally, Section 6.6 concludes this Chapter.

6.1 Threat Scenario and Defense Overview

Threat Scenario Our goal is to prevent an attacker from recovering a model's secret architecture. For this, we place ourselves in the same threat scenario as the Cache Telepathy attack [141].

The attacker in [141] is relatively strong since she has access to the cache and the computations are supposed to run on the CPU. On the other hand, this attack extracts more information than [53, 52] and does not require the help of trained ML models as in [93]. The last point makes the attack efficient, as it does not require the gathering of a dataset or training another ML model.

This is why we focus on this cache attack. But because our defense consists in reordering CNN computations, as mentioned above, several other architecture extraction attacks should also be mitigated [53, 52, 10].

The aim of [141] is to reduce the set of possible architectures to one small enough that an attacker can train all models in it and select the best one. This is further detailed in Section 2.3.4. It is a cache-based SCA in an MLaaS context. The attacker is assumed to have the following abilities:

- She knows a set S of possible architectures for the target model. The set is

intractable: an attacker cannot realistically train all models and determine the best one.

- The attacker shares the victim's ML frameworks.
- The attacker and the victim share the cache (co-location). This enables the attacker to monitor the GeMM functions (detailed in Section 2.2.4)

Given those assumptions, the attacker aims at recovering all of the hyperparameters (or at least reducing the number of possibilities for them). To do so, she uses Flush and Reload or Prime and Probe (see Section 2.3.3) to monitor internal functions in the GeMM algorithm, as explained in Section 2.3.4. The full description of Cache Telepathy can be found in Section 2.3.4.

Protection Overview The attacker in many reverse-engineering attacks relies on the sequential execution of NNs. Therefore, we consider reordering computations and adding randomness in the way neurons are computed to thwart those attacks. Although it is impossible to determine the value of an FC layer neuron before earlier layers are done computing, it is not the case for convolutional and pooling layers. As shown in Figure 6.1, a given neuron in convolutional layer i only requires the values from a window of neurons from layer $i-1$. In the figure, the blue neurons in layer $i-1$ are the only one necessary to compute the green neuron in layer i.

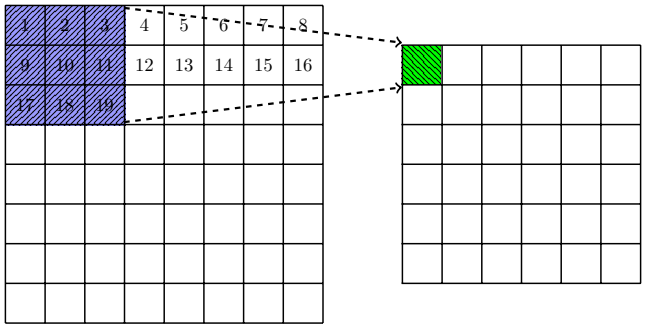


Figure 6.1: The blue neurons from convolutional layer i (left) are the ones needed to compute the green neuron in layer $i+1$ (right), when layer i's filter size is 3.

Related Works Several papers propose to modify the way convolutional layers are generally computed, with the aim of accelerating CNN computations on hardware devices.

The authors of [109] consider each neuron individually, and compute its value as soon as it is ready. Indeed, in convolutional layers, only a small window of values from the previous layer is used to compute a given neuron. In [109], a buffer in layer i stores the values as they arrive in a sequential order. Once enough values for a given neuron arrive on the buffer, it executes the computation and sends the value to the following layer. The scheme is described in Fig. 6.1.

The authors of [2] consider computing CNN layers in a similar fashion. The aim of [2] and [109] is to limit the bandwidth necessary to make NN computations, by

only loading on chip the necessary values. It also enables a parallelization of some computations.

As the following sections will explain it, we consider a similar approach as [109] and [2], but outside of the hardware context. Furthermore, our goal is not to accelerate the computations but to protect them from architecture extraction attacks.

[128] also considers the computations in NNs in a different order, but their goal is to run the victim NNs in Trusted Execution Environments (TEEs). Since TEEs have a limited memory space, all weights and inputs cannot be loaded at once. Therefore, they partition the target NNs in three different ways (per layer, within a layer and branched partitioning) when loading them into the TEEs.

Like us, some papers [78, 30, 31] aim at protecting NNs from architecture extraction attacks. They are further detailed in Section 2.3.7. While the authors of [31] ensure the power traces of NN inference are leakage free, [78] and [31] mitigate memory access pattern attacks. To the extent of our knowledge, no paper has tackled the cache-based SCAs for architecture extraction yet.

In this Chapter, we mix two ideas: the interleaving of layers presented in [109] and the block multiplications as in GeMM. We apply them to the software level rather than the hardware one. Our aim also differs from previously mentioned papers: while existing approaches [2, 109] apply the interleaving of layers for efficiency, we apply it for security purposes. As detailed in Section 6.3, we also added a randomization element. This mix of the two ideas along with the randomization lead us to some experimental results showing our idea does thwart the Cache Telepathy attack (see Section 6.5).

6.2 Reordering Computations: the Convolutional Case

In this Chapter, our goal is to mitigate cache attacks targeting the GeMM computations (the GeMM algorithm is detailed in Section 2.2.4) during the inference phase of a victim CNN. We consider out of scope other side-channel vectors such as power consumption or memory access patterns. Furthermore, our proposed method concerns convolutional and max pooling layers. Even though Cache Telepathy also targets FC layers, we will see in Section 6.5.1 that our approach still mitigates the attack.

6.2.1 Convolutional Layer

The protection we propose is based on two observations:

First, the sequential execution of layers enables a potential attacker to determine the depth of an NN. Indeed, the depth directly results from the number of observed matrix multiplications.

Second, the hyper-parameters of a given layer can be deduced by a potential attacker because each layer is executed as a whole before moving on to the next one.

Therefore, a depth-first computation should improve the security of an NN. In CNNs, several neurons in a layer i can be executed before layer $i-1$ has been fully

computed. Indeed, a neuron only requires a window of values from the previous layer, as described in Figure 6.1.

Based on these observations, we propose to compute layers in a depth-wise fashion. But instead of computing a neuron in layer i as soon as all necessary neurons in layer $i-1$ are ready, we wait for a block of neurons in layer i to be ready before starting the execution. With this method, we aim at making layer computations overlap, without being restricted to computing one neuron at a time.

Let us detail our proposed method. Our goal is to make the computations of several layers overlap. We start executing layer $i+1$ before the execution of layer i is over. The GeMM algorithm [45] is thoroughly optimized, and makes sure the entire cache is used for large matrix multiplications. Making one neuron computation at a time – to execute neurons as soon as enough data is available – would lead to too much overhead. Thus, a balance needs to be reached between the added overhead and the number of subdivisions of matrix multiplications.

Block computations Let us first consider the case of convolutional layers. Let layer i be a convolutional one, with $n_k^{(i)}$ filters of size $k^{(i)} \times k^{(i)}$. As a reminder, each convolutional layer is computed as one matrix multiplication between a reshaped input matrix R and a reshaped filter matrix F. Let R denote the reshaped matrix – as in Figure 2.7 – of the input I (of size $n \times n$). For an example of a standard way to reshape the input, see Figure 6.2. Let F denote the $n_k^{(i)} \times (k^{(i)} \cdot k^{(i)})$ matrix where each row is a flattened filter.

Here, our goal is to compute the matrix multiplication $F \times R$ by blocks. Every time a block A of neurons in matrix R is ready, we multiply A with the corresponding filter blocks in F. Let us detail how this is achieved.

If there is no padding, the reshaped matrix R has dimensions $({k^{(i)}}^2 \cdot n_k^{(i)}) \times (n - k^{(i)}+1)^2$. Let $B \in \mathbb{N}$. R is divided into N non-overlapping blocks $\{R_{B_l}\}_{1 \leq l \leq N}$ of size $B \times B$. There are $W^{(i)} := \left\lceil \frac{(n-k^{(i)}+1)^2}{B} \right\rceil$ blocks R_B width-wise and $H^{(i)} := \left\lceil \frac{k^{(i)} \times k^{(i)}}{B} \right\rceil$ blocks height-wise, corresponding to a total of:

$$N^{(i)} := \left\lceil \frac{(n - k^{(i)} + 1)^2}{B} \right\rceil \times \left\lceil \frac{k^{(i)} \times k^{(i)}}{B} \right\rceil$$

blocks R_B.

Each such block R_{B_r} needs to be multiplied by filter blocks F_{B_r} of size $n_k^{(i)} \times B$. We can further divide F_{B_r} into $M_r^{(i)}$ blocks $\{F_{b_{r,l}}\}_{1 \leq l \leq M_r^{(i)}}$ of size $B \times B$. If $n_k^{(i)}$ is not a multiple of B, we pad the last block $F_{b_{M_r^{(i)}}}$. Thus, each block R_B needs to be multiplied by:

$$M_r^{(i)} := \left\lceil \frac{n_k^{(i)}}{B} \right\rceil$$

filter blocks.

Once the layer receives all the values in the $r-th$ block R_{B_r} from the previous layer, all the multiplications $\{R_{B_r} \times F_{b_{r,j}}\}_{1 \leq j \leq M_r^{(i)}}$ are computed. The results are added to those of the other R_B blocks involving the same columns in R. Since there are $H^{(i)}$ blocks R_B height-wise, $H^{(i)}$ matrix multiplications are required to

Algorithm 2: `add_elts`: Receive elements from the previous layer, and compute ready blocks

```
   input : Block size B. Arrays A_C, B_C. Value v to add. Accumulator acc
 1 L = get_indices_reshaped_input(v) /* Gets indices where v needs to be added. */
 2 for i ∈ L do
 3     b_c = get_block_index(i); /* Gets index of the layer block i belongs to. */
 4     B_C[b_c] += 1;
 5     if B_C[b_c] = B × B then
           /* If the block is now full */
 6         filter_blocks = get_associated_filter_blocks(b_c) ;
 7         for F ∈ filter_blocks do
 8             R = compute_sgemm(F, b_c);
               /* Send values to next layer. */
 9             accumulator_handler(B, A_C, B_C, acc, R, get_next_layer());
10         endfor
11     end
12 endfor
```

compute one neuron for layer $i+1$. Since, moreover, computations are made with sub-matrices of sizes $B \times B$, B neurons are computed at a time.

Block characteristics Let us note that GeMM computations are more efficient when matrix sizes are multiples of 32 [45]. Thus, the default block size should be a multiple of 32 as well. Moreover, the number of computations is correlated with the block sizes: if we increase the block size, there are fewer matrix multiplications. But the efficiency of the computations also depends on the cache size, the various layers' input sizes and the padding added. It is therefore important to tailor the block size to the architecture's hyperparameters. Furthermore, taking into account the matrix sizes in each layer is important: if the block sizes are too large, no overlap can occur between a convolutional layer and the following one. Thus, block sizes need to be adapted to the architecture at hand.

Process for one convolutional layer Figure 6.3 summarizes the process for one convolutional layer. In Step 1, layer $i-1$ sends neuron 7 to layer i, which adds it in the reshaped matrix R. Since neuron 7 fills the red block R_{B_r}, a matrix multiplication can occur. Step 2 corresponds to the multiplications with the blue and green filter blocks $\{F_{b_{r,j}}\}_{j\in\{1,2\}}$. The results are added to the output matrix (O' in Figure 2.7). A value in the output matrix is only correct when the associated column in R is fully computed.

Algorithms 2 and 3 provide the pseudo-code for our method. Algorithm 2 receives a list of elements from layer $i-1$ and checks whether a block R_{B_r} is full (line 5). If it is the case, it computes $\{F_{b_{r,j}} \times R_{B_r}\}_{1\leq j\leq M_r}$ (using `compute_sgemm`, line 10) whether layer $i-1$'s execution is over or not. It sends the temporary values to layer i by calling Algorithm 3: `accumulator_handler` (lines 7-10). Algorithm 3 takes those output elements, adds them to the correct locations in $R^{(i)}$ (lines 1-4) and checks whether a neuron is completely computed (line 7). If it is the case, it sends the result to layer $i+1$.

Algorithm 3: `accumulator_handler`: Add computed elements and send computed neurons to the next layer

```
   input : Block size B. Arrays A_C, B_C. Array acc. Matrix of computed values R. Pointer to the next
           layer: next_layer
   /* acc stores the current values of the layer's elements.  A_C is used to check whether a
      value in acc is ready.  B_C is used to check whether a block is ready.  R is the result
      of a matrix multiplication.  */
 1 indices = get_indices_acc(R)
 2 n = len(R)
 3 for i = 0 to n do
 4 |   acc[indices[i]] += R[i];
 5 |   A_C[indices[i]] += 1;
 6 |   f_c = get_full_ac(indices[i], B); /* Gets the number of matrix multiplications necessary to
   |      have a correct value in acc[indices[i]] */
 7 |   if A_C[indices[i]] == f_c then
 8 |   |   next_layer→add_elts(indices[i], A_C, B_C);
 9 |   end
10 endfor
```

6.2.2 Dealing with Pooling Layers

Pooling layers need to be dealt with differently. The reason why is twofold: first, in such layers, the various channels are managed independently. Filter sizes are small, resulting in a small height for reshaped matrices. It is therefore more practical to consider blocks width-wise only. Second, no GeMM multiplication is involved in the computation. They are therefore not the target of the cache attack considered.

However, executing a pooling computation introduces overhead in between block computations. This leaks some information to a potential attacker and she might determine the number of multiplications required to obtain one column in the reshaped pooling input. This provides the attacker with a small range of possible hyper-parameter values. If the victim waits for several columns in the reshaped pooling layer to be ready before starting the execution, the range of possible hyper-parameter values increases, making it harder for an attacker to recover the correct architecture. It is therefore important to also consider a blocked computation for layers without GeMM computations, such as pooling ones.

We propose to adapt the methodology described in Section 6.2.1 to pooling layers. We still compute several neurons at once. But here, we wait for multiple entire columns to be completed instead of blocks. Moreover, we deal with the various channels independently. We consider a pooling layer i with a window size of $k^{(i)} \times k^{(i)}$. Let $B \in \mathbb{N}$. Given the max pooling layer's input I of size $n \times n$, let R (of size $k^{(i)^2} \times (\frac{n}{k^{(i)}})^2$) be its reshaping. We divide R into $N^{(i)} := \left\lceil \frac{(n/k^{(i)})^2}{B} \right\rceil$ blocks $\{R_{B_r}\}_{1 \leq r \leq N^{(i)}}$ width-wise. Whenever all of block R_{B_r}'s values are ready – i.e. whenever they were relayed by the previous layer –, the computation can be executed for that block. This results in B neurons that need to be passed on to the following layer.

6.3 Randomization of Block Sizes

Computing the matrix multiplications by blocks as explained in Section 6.2 helps mitigate the attack at hand, as the attacker only recovers a set of possible hyper-

Algorithm 4: `full_computation`: Full computation with random

```
   input : Model's input I. Arrays A_C, B_C. Block size B
1  For each convolutional layer, generate random arrays W (block widths) and H (block heights)
2  for input blocks i_b do
3  |  filter_blocks = get_associated_filters(i_b);
4  |  for F ∈ filter_blocks do
5  |  |  R = compute_sgemm(F, i_b);
6  |  |  accumulator_handler(B, A_C, B_C, acc, R, get_next_layer());
7  |  endfor
8  endfor
```

parameter values. However, it is still possible to increase that range of values, by injecting randomness in the block sizes.

6.3.1 Improving Security through Randomization

So far, within a layer i, all block sizes were identical. Having different block sizes within a layer provides more entropy in the number of multiplications per layer (see Section 6.5.1 for a more detailed analysis of the protection's security).

Let us first consider the convolutional case. When the architecture of the target NN is created, we generate two arrays `tw` and `th` of random block sizes. The first array corresponds to the number of columns of each block, while the second corresponds to the number of rows of each block. `tw[l]` returns the index of the column at which the $l-th$ block width-wise starts, and `th[j]` returns the index at which the $j-th$ block height-wise starts. A block with coordinates (l, j) therefore has `th[l]`−`th[l-1]` rows and `tw[j]`−`tw[j-1]` columns. Figure 6.4 provides an example of such a subdivision.

In that scenario, the blocks in the same column have the same width, and those in the same row have the same height. To prevent an attacker from using this information, we can zero-pad the blocks right before each multiplication to turn the rectangle blocks into squares. Thus, each block is a square of size $\max(width, height)$, where *width* and *height* are the block's original width and height. This way, the blocks in a same row or in a same column can have different block sizes.

As explained in Section 6.2.2, the reshaped pooling matrices cannot be divided height-wise due to their small height. Therefore, only one array of random block sizes is created, to divide the matrix width-wise.

In the random case, Algorithm 2 needs to be updated to take into account the height and width of the considered block. Only lines 5 and 6 in Algorithm 2 change. Before the loop on line 5, we get the correct block width and height thanks to the generated arrays `tw` and `th`. These are then provided to `accumulator_handler`.

Algorithm 4 is the full computation. For all input elements, the function computes the GeMM multiplications (line 5) and sends them, one by one, to the accumulator handler (line 6). This is enough to start the whole process.

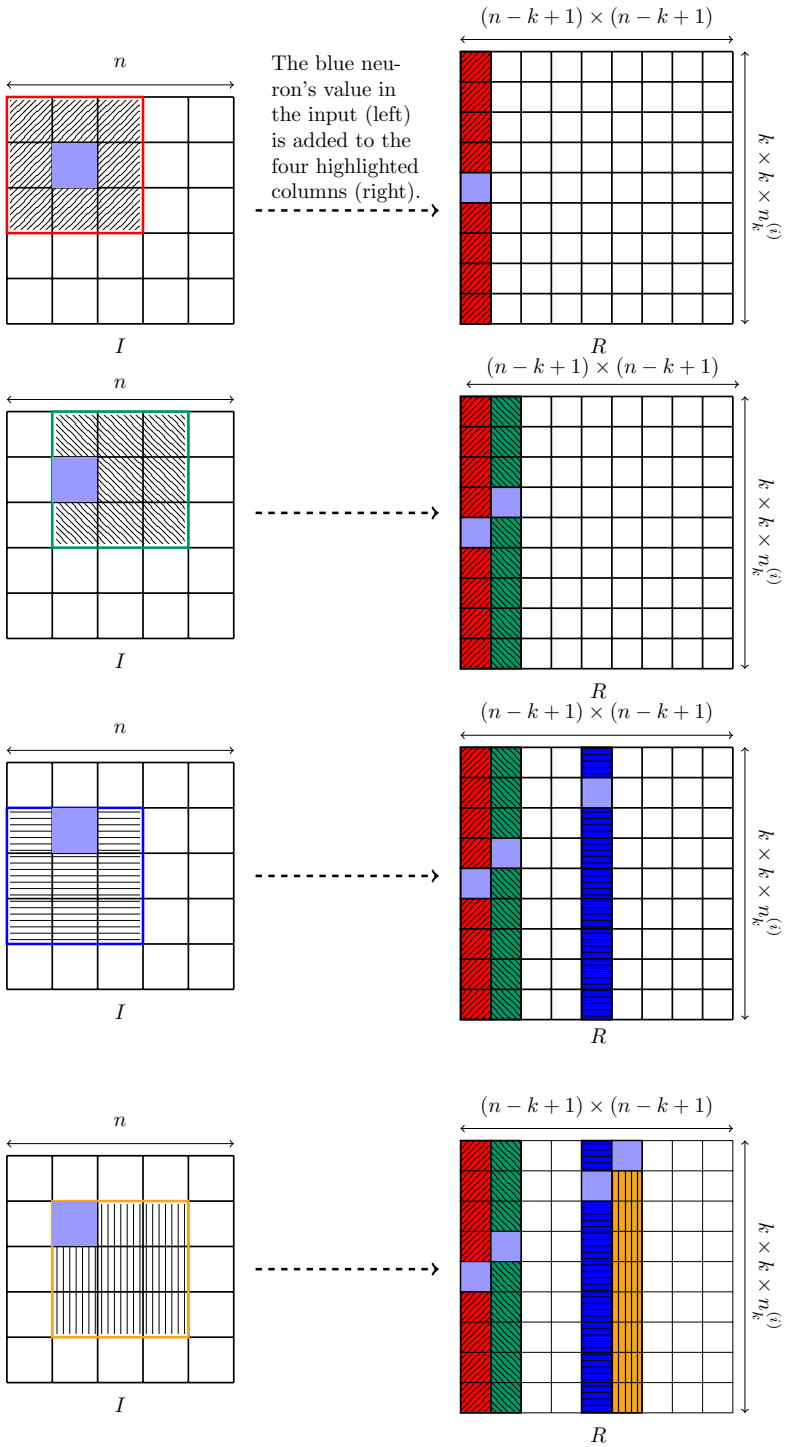


Figure 6.2: Reshaping the input to turn a convolution into a matrix multiplication. The input size is $n \times n$ and the filter size is $k \times k$.

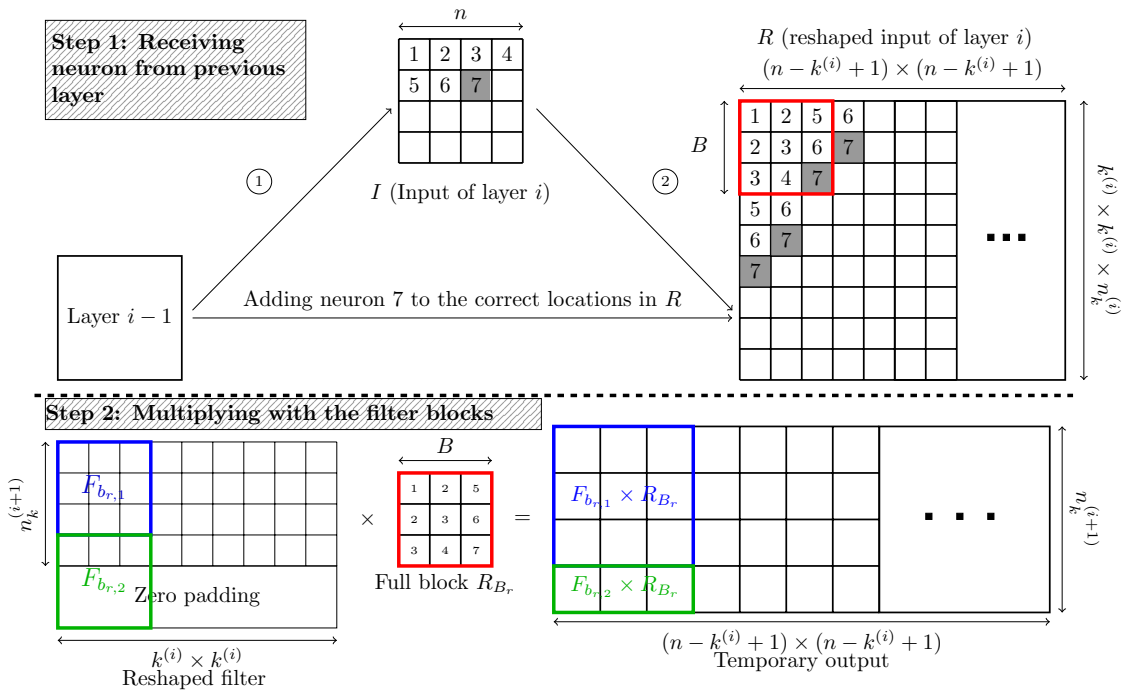


Figure 6.3: Process for a layer of size 4×4 receiving values from the previous layer. The block size is $B = 3$ and the filter's width and height are equal to 3.

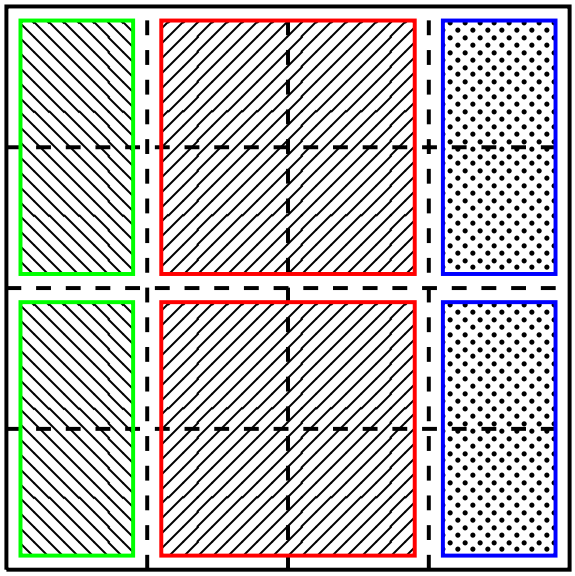
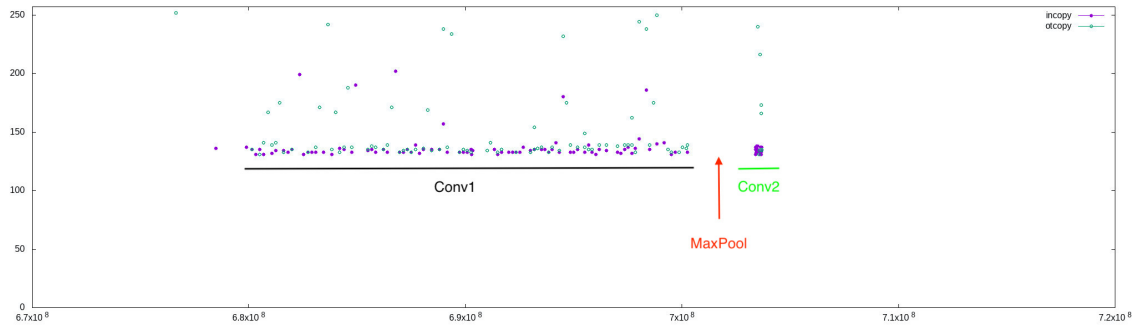


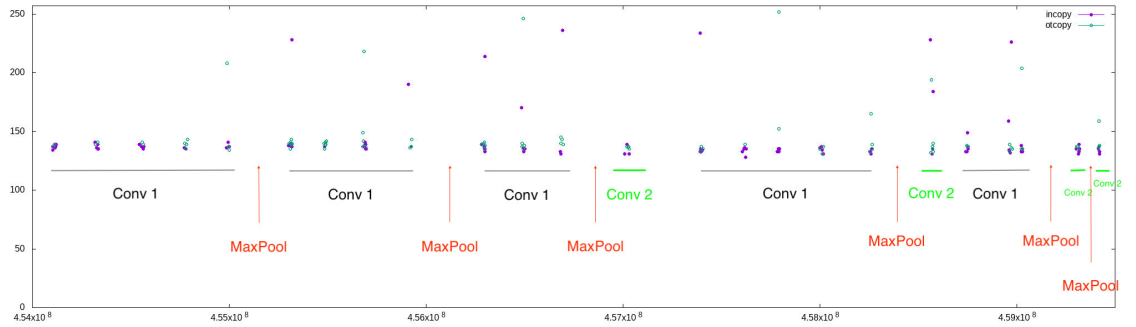
Figure 6.4: Example of a subdivision of a reshaped input

6.4 Full Scheme

Let us now clearly state the steps of our proposed countermeasure, that we call *Telepathic Headache*.



(a) Unprotected case. The layers are executed sequentially. We observe a large matrix multiplication corresponding to $Conv1$, a latency for the max pooling layer and a smaller, matrix multiplication corresponding to $Conv2$.



(b) Protected case. Block sizes are between 32 and 38. Each group of points corresponds to one multiplication. 5 multiplications from $Conv1$ are executed, then there is a latency for one max pooling computation, then another 4 multiplications from $Conv1$, etc. We see that $Conv1$, $MaxPool$ and $Conv2$ interleave.

Figure 6.5: Flush and Reload on an unprotected (a) and a protected (b) architecture with two convolutional layers (filter size: 3×3) separated by a max pooling layer (window size: 2×2). The y-axis corresponds to the number of clock cycles it takes to access the correct cache line. The x-axis indicates the time elapsed since the beginning of the experiment.

1. For each convolutional layer: Generate two arrays of block sizes, for the width and the height. Possible block sizes need to fulfill two conditions:
   (a) Not too large: it would defeat the purpose of the countermeasure, as we need an overlapping of layers. For instance, $B \leq n$ where n is the input size. But higher B values are often possible.
   (b) Not too small: it would lead to too much overhead (for instance, $B > 1$).
2. For each pooling layer: Generate an array of width-wise block sizes. Block sizes need to fulfill the two previous conditions as well.

3. Start the first layer computations.

4. If a block in layer i is full, compute the matrix multiplications.

5. If the previous computations lead to at least one neuron being completely computed, send the value to layer $i+1$ and pursue computations in layer $i+1$ if possible.

6. Repeat steps 4 and 5 until all values have been computed.

Steps 1 and 2 are part of the architecture creation, and are only executed once for a given model. Generating the new, protected architecture only once per model and user prevents a potential attacker from carrying out statistical attacks.

Let us further detail the way we deal with convolutional GeMM computations in step 4. Let R_{B_r} be a full block in layer i, and w and h be its corresponding width and height respectively. As explained in Section 6.2.1, each R_{B_i} is associated with filter blocks $\{F_{b_{r,j}}\}_{1 \leq j \leq M_r^{(i)}}$. $M_r^{(i)}$ multiplications $F_{b_{r,l}} \times R_{B_r}$ $\forall 0 \leq l \leq M_r^{(i)}$ are computed. The output of those multiplications are sent to the next layer. Since R_{B_r}'s columns are only subcolumns from the reshaped input R's columns, the outputs computed are only partial results. The results for all height-wise input blocks need to be summed in order for some neurons to be ready in the following layer.

Figure 6.5 shows that contrary to a normal execution, our protection leads to an overlap of layer computations.

6.5 Results

6.5.1 Security Analysis

We consider an attacker whose goal is to recover the architecture of a target CNN. The list of hyper-parameters to recover is therefore:

- Number and types of layers, and connections between layers.
- Filter sizes for convolutional layers and window sizes for pooling ones.
- Input and output sizes.
- Padding.

The authors of Cache Telepathy restrict the possible target architectures to a search space. The latter is built thanks to the following assumptions:

- In convolutional layers with filter size $k \times k$, we have that: $k \in \{1, ..., 11\}$.
- The padding p is such that $p \leq k$.
- In each layer, the number of output channels is such that: n_{out} is a multiple of 64 and $n_{out} \leq 64 \times 32$.

We make the same assumptions in this Chapter.

As explained in Section 2.3.4, the attackers in Cache Telepathy monitor GeMM computations in order to recover those hyper-parameters. The number of convolutional layers in the target CNN is equal to the number of matrix multiplications. Filter sizes are deduced from the sizes of the matrices involved in the multiplications. The possibilities for padding values and pooling sizes are restricted thanks to constraints on the dimensions of each layer.

Section 2.3.4 also mentions that the Flush and Reload attack along with these assumptions enable the attacker to limit the search space to a very small set of possible architectures. For instance, for the VGG16 architecture [115], the Cache Telepathy attack reduces the search space from over 5.4×10^{12} possible architectures to 16.

Let us consider layers 4 ($Conv4$), 5 ($Conv5$) and 6 ($MaxPool$) of VGG16 (see Figure 6.6). They are two convolutional layers followed by a max pooling layer. $Conv4$ and $Conv5$ have input size $n \times n = 112 \times 112$. Let $k^{(i)} \times k^{(i)}$ denote $Conv$i's filter size. In our case, $k^{(4)} = k^{(5)} = 3$. The input size is the same in both layers because a padding of two ($p = 1$) is applied in both directions to $Conv4$'s output. $Conv4$ has $in_4 = 64$ input channels and $out_4 = 128$ output channels. $Conv5$ has $in_5 = out_5 = 128$ input and output channels.

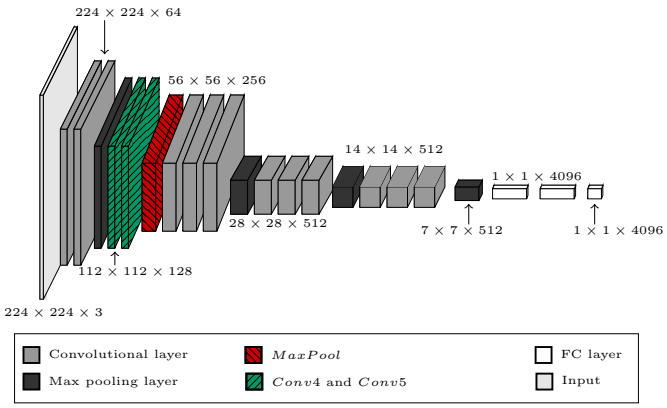


Figure 6.6: VGG16 architecture. The sizes mentioned in the figure are the input sizes of the layers in the form: (input width × input height × number of channels).

The reshaped input image R_4 for $Conv4$ has width:

$$W^{(4)} = (n + 2 \cdot p - k^{(4)} + 1)^2 = 112 \times 112$$

and height:

$$H^{(4)} = k^{(4)} \times k^{(4)} \times in_4 = 576$$

. The reshaped filter matrix F_4 for $Conv4$ has width:

$$\begin{aligned} F_W^{(4)} &= H^{(4)} = k^{(4)} \times k^{(4)} \times in_4 \\ &= 576 \end{aligned}$$

and height:

$$F_H^{(4)} = out_4 = 128$$

. A potential attacker is assumed to know n and in_4. This is a reasonable assumption, since the input size is known to an attacker who can query the target model. Furthermore, for simplicity, let us assume that the attacker knows that layers are sequentially connected.

Recovering the unprotected architecture with Cache Telepathy: By monitoring GeMM multiplications, the attacker in Cache Telepathy can identify the two convolutional layers, as each corresponds to one matrix multiplication. Moreover, the lack of a max pooling layer in between the two layers is identified through timing analysis. Indeed, a max pooling layer incurs a time overhead between GeMM operations. The attacker also determines the matrix sizes. This directly provides her with $out_4 = in_5$ and $H^{(4)}$. Since $H^{(4)} = k^{(4)} \times k^{(4)} \times in_4$ and in_4 is known, the attacker can deduce $k^{(4)}$. The attacker also directly observes that $n' = (n + 2 \cdot p - k^{(4)} + 1)^2$. Since the attacker now knows n and $k^{(4)}$, she can easily recover $p = \frac{\sqrt{n'} - n + k^{(4)} - 1}{2}$. $MaxPool$'s window size is determined to be 2 by comparing the input and output sizes of the pooling layer.

Thus, monitoring GeMM computations enables an attacker to find:

- The number of convolutional layers by counting the GeMM operations.
- The input and output shapes (including channels) of $Conv4$ and $Conv5$ thanks to the matrix sizes.
- The filter sizes for the two layers thanks to the matrix sizes.

In short, the attacker manages to recover all the hyper-parameters of $Conv4$, $Conv5$, as well as the maxpooling layer that follows them.

Protected Case, when convolutional layers can be distinguished: Let us now study the impact of our protection on those two convolutional layers. The attacker has at her disposal the same information as before. Once again, the goal is to recover the hyper-parameters of $Conv4$, $Conv5$ and $MaxPool$. By launching Cache Telepathy, the attacker is able to recover all matrix sizes in the GeMM multiplications. In our case, the layers are not executed sequentially but in blocks of various sizes, and depth-wise. This means that as soon as a block of values – of random size, as explained in Section 6.3 – in layer i is ready, it is executed, regardless of whether layer $i-1$ has finished its execution. Thus, the layers interleave, as shown in Figure 6.5.

Because the order of multiplications is changed and the layers are not executed sequentially, the attacker knows neither the number of layers nor which layer a matrix multiplication belongs to. However, an attacker can detect a pooling layer computation, since it incurs a latency between multiplications. Therefore, she might determine the number of multiplications required for the first blocked max pooling computation to occur. We should also bear in mind that several architectures have multiple consecutive convolutional layers [115, 75].

For a clearer explanation, we first focus on the unlikely case where the attacker can observe a change of layer (and therefore identify the first $Conv5$ operation). Let

us note that we believe this case to be mainly hypothetical. For it to hold, there should be a notable latency in between convolutional layers. The said latency should also be different from the overhead incurred by max pooling layers. We present this scenario for pedagogical purposes.

As $Conv4$ and $Conv5$ go through the GeMM process (see Figure 2.7), inputs I_4 and I_5 are reshaped into R_4 and R_5 respectively. Let us note that in order to have an output size equal to the input size, the input needs to be padded. We consider I_4 and I_5 to be the padded inputs. Their shape is $(64, 114, 114)$ and $(128, 114, 114)$ respectively. Let $O^{(j)}$ denote the reshaped output of layer j – i.e. the output of the layer's matrix multiplication. Let $H^{(4)} = \{H_1^{(4)}, ..., H_t^{(4)}\}$ be the number of blocks height-wise for each width-wise column of blocks. Let us denote $w^{(4)} = \{w_1^{(4)}, ..., w_{t'}^{(4)}\}$ – respectively $h^{(4)} = \{h_1^{(4)}, ..., h_t^{(4)}\}$ – the sizes of the blocks width-wise – respectively height-wise. These are determined as described in Section 6.3. Furthermore, let $\{M_{i,j}^{(4)}\}_{1 \leq i \leq t, 1 \leq t'}$ denote the number of filter blocks associated with each of $Conv4$'s input blocks. The reshaped filters are F_4 and F_5 respectively. The blocks we consider are submatrices in $R_{4,5}$ and $F_{4,5}$. R_4 and R_5 both have width $W_{4,5} = 112 \times 112$. R_4 has height $3 \times 3 \times 64$ while R_5 has height $3 \times 3 \times 128$.

In our case, each block has size between 32×32 and 64×64. Let (x_b, y_b) be the coordinates of the last element in the first block in R_5. If we take the minimal block size, then $(x_{\min}, y_{\min}) = (32, 32)$. In the case of the maximal block size, $(x_{\max}, y_{\max}) = (64, 64)$.

Let $tot_{x,y}^{i,j}$ denote the number of matrix multiplications in layer i required for element (x, y) in R_j to be ready. Since our goal is to determine the number of multiplications in $Conv4$ necessary to have the first block in $Conv5$ ready, we actually need to compute $tot_{x_b,y_b}^{4,5}$. We will consider block size extremes to give a range of values for $tot_{x_b,y_b}^{4,5}$:

$$tot_{x_{\max},y_{\max}}^{4,5} \leq tot_{x_b,y_b}^{4,5} \leq tot_{x_{\min},y_{\min}}^{4,5}$$

In order to compute that value, we need to find the coordinates of element $e = (x, y) = (x_b, y_b)$ in I_5 rather than R_5. Indeed, this will provide us with the blocks that need to be computed in $Conv4$ to obtain e.

First, let us find which input channel ch element e belongs to. Because of the way R_5 is obtained, we have that:

$$ch = \left\lfloor \frac{x}{k^{(4)} \times k^{(4)}} \right\rfloor$$

Thus:

$$ch_{e_{\min}} = \left\lfloor \frac{32}{k^{(4)} \times k^{(4)}} \right\rfloor = 3$$
$$ch_{e_{\max}} = \left\lfloor \frac{64}{k^{(4)} \times k^{(4)}} \right\rfloor = 7$$

where $e_{\min} = (x_{\min}, y_{\min})$ and $e_{\max} = (x_{\min}, y_{\min})$ indicate the maximal and minimal block sizes respectively.

Let us now find the coordinates (row, col) of element e in channel ch. Let $n' = n + 2 \cdot p = 114$ be the padded input width and height. With an input of shape

(n', n'), a convolution results in an output of size $(n' - k^{(4)} + 1, n' - k^{(4)} + 1) = (n, n)$ here. Because of the way I_5 is reshaped, each column in R_5 is a $k^{(5)} \times k^{(5)}$ window in I_5. Thus,

$$row = \left\lfloor \frac{y}{n} \right\rfloor + \left\lfloor \frac{x \mod (k^{(5)} \times k^{(5)})}{k^{(5)}} \right\rfloor$$
$$col = y \mod n + (x \mod (k^{(5)} \times k^{(5)})) \mod k^{(5)}$$

As mentioned before, these coordinates include the padding. We need to remove said padding to find out how many computations from the previous layer need to have been made. If (row, col) does not correspond to a padding value, we have:

$$row_{unpadded} = row - p$$
$$col_{unpadded} = col - p$$

If $row < p$ with $ch = 0$, then no value from the previous layer needs to be computed for the first block to be executed. This results in $tot^{4,5}_{x,y} = 0$. The same goes for $row = p$, $ch = 0$ and $col < p$.

If (x, y) is a padding value outside of the two previous cases, then we consider the previous non-padding value in I_5. This corresponds to:

$$row_{unpadded} = row - 1$$
$$col_{unpadded} = n'$$

Indeed, we then need to take the last (non-padding) element in the previous row.

With $(x_{\min}, y_{\min})$ and $(x_{\max}, y_{\max})$, we have that:

$$\begin{aligned}
row_{\min} &= \left\lfloor \frac{32}{112} \right\rfloor + \left\lfloor \frac{32 \mod (3 \times 3)}{3} \right\rfloor && = 2 \\
col_{\min} &= 32 \mod 112 + (32 \mod (3 \times 3)) \mod 3 && = 34 \\
row_{\max} &= \left\lfloor \frac{64}{112} \right\rfloor + \left\lfloor \frac{64 \mod (3 \times 3)}{3} \right\rfloor && = 0 \\
col_{\max} &= 64 \mod 112 + (64 \mod (3 \times 3)) \mod 3 && = 65
\end{aligned}$$

Because $e_{\max} = (x_{\max}, y_{\max})$ is in the first row of I_5, it corresponds to a padding element. However, not all elements in that block stem from padding elements. Instead, we can take the last non-padding element in the previous channel. We therefore need to consider channel $ch'_{e_{\max}} = ch_{e_{\max}} - 1 = 6$. We also need to select the last element of the (3×3) input window, meaning:

$$row'_{\max} = 2$$
$$col'_{\max} = 66$$

We then remove the padding from $e_{\min} = (x_{\min}, y_{\min})$ so as to find the coordinates

in $Conv4$'s output. This gives us:

$$
\begin{aligned}
row'_{\min} &= 1 \\
col'_{\min} &= 33 \\
ch'_{e_{\min}} &= 3 \\
row'_{\max} &= 2 \\
col'_{\max} &= 66 \\
ch'_{e_{\max}} &= 6
\end{aligned}
$$

These are the coordinates of the last element we need in the output of $Conv4$. The coordinates c of that element in the said output are:

$$
\begin{aligned}
c_{\min} &= (ch'_{e_{\min}}, row'_{\min} \times n + col'_{\min}) = (3, 145) \\
c_{\max} &= (ch'_{e_{\max}}, row'_{\max} \times n + col'_{\max}) = (6, 290)
\end{aligned}
$$

We now have enough information to compute $tot_e^{4,5}$.
Let $b = \arg\min_{b'} \left(\sum_{q=0}^{b'} w_1 \geq c \right)$. b is the number of the block width-wise containing c. For each block number $b' < b$, we need to multiply all the blocks height-wise with their associated block filters, meaning $H_{b'}^{(4)} \times M_{b'}^{(4)}$ multiplications for each block b'. For b, we need to compute all the height-wise block multiplications up to the block containing ch. This is equal to: $\left\lceil \frac{ch}{h_1^{(4)}} \right\rceil \times H_b^{(4)}$. The full formula is then:

$$
tot_e^{1,2} = \left(\sum_{q=0}^{b-1} H_q^{(4)} \times M_q^{(4)} \right) + \left\lceil \frac{ch}{h_1^{(4)}} \right\rceil \times H_b^{(4)} \tag{6.1}
$$

We can apply that to $e_{\max}$ and $e_{\min}$, considering constant block shapes of either

$(32, 32)$ for $e_{\min}$ or $(64, 64)$ for $e_{\max}$:

$$b_{e_{\min}} = \left\lceil \frac{145}{32} \right\rceil = 5 \tag{6.2}$$

$$H^{(4)}_{q,\min} = \left\lceil \frac{3 \times 3 \times 64}{32} \right\rceil = 18 \ \forall q \leq b \tag{6.3}$$

$$M^{(4)}_{q,\min} = \left\lceil \frac{128}{32} \right\rceil = 4 \ \forall q \leq b \tag{6.4}$$

$$\left\lceil \frac{ch_{e'_{\min}}}{32} \right\rceil \times H^{(4)}_{b,\min} = 4 \tag{6.5}$$

$$tot^{4,5}_{e_{\min}} = b_{e_{\min}} \times H^{(4)}_q \times M^{(4)}_q + \left\lceil \frac{ch_{e'_{\min}}}{32} \right\rceil \times H^{(4)}_b \tag{6.6}$$

$$= 292 \tag{6.7}$$

$$b_{e_{\max}} = \left\lceil \frac{290}{64} \right\rceil = 5 \tag{6.8}$$

$$H^{(4)}_{\max} = H^{(4)}_{q,\max} = \left\lceil \frac{3 \times 3 \times 64}{64} \right\rceil = 9 \ \forall q \leq b \tag{6.9}$$

$$M^{(4)}_{\max} = M^{(4)}_{q,\max} = \left\lceil \frac{128}{64} \right\rceil = 2 \tag{6.10}$$

$$\left\lceil \frac{ch_{e'_{\max}}}{64} \right\rceil \times H^{(4)}_{b,\max} = 2 \tag{6.11}$$

$$tot^{4,5}_{e_{\max}} = (b_{e_{\max}} - 1) \times H^{(4)}_{\max} \times M^{(4)}_{\max} + \left\lceil \frac{ch_{e'_{\max}}}{64} \right\rceil \times H^{(4)}_{b,\max} \tag{6.12}$$

$$= 74 \tag{6.13}$$

Thus, the total number of matrix multiplications required for one block in $Conv5$ to be ready ranges between 74 and 292 depending on the block sizes: $tot^{4,5} \in [74, 292]$ (by Equations (6.7) and (6.13)).

This range of values can be obtained because we know the padding and filter sizes. A potential attacker, however, does not have access to this information. She only knows the number of input channels and the input shape.

The remaining question is whether the range of values for different filter and padding sizes overlap. If it is the case, then an attacker cannot differentiate between the various hyper-parameter values. Let us suppose, once again, that $k = k^{(4)} = k^{(5)}$ and compute all the possible values for $tot^{4,5}_{x,y}$ ($(x'_{\min}, y'_{\min}) = (32, 32)$ and $(x'_{\max}, y'_{\max}) = (64, 64)$), given out_4, out_5 and in_5. Table 6.1 shows the resulting values.

Table 6.1: Maximal and minimal number of multiplications depending on the filter size, when the attacker can distinguish between convolutional layers.

$k^{(4)}$	1	2	3	4	5	6	7	8	9	10	11
$tot^{4,5}_{x'_{\min}, y'_{\max}}$	2	8	306	928	1,400	4,896	5,488	7,168	6,480	8,000	6,776
$tot^{4,5}_{x'_{\max}, y'_{\min}}$	1	4	9	16	25	36	49	64	324	400	242

If $k = 1$, $tot^{4,5} \in [1, 2]$. If $k = 2$, $tot^{4,5} \in [4, 8]$ and so on. Because of those ranges, if an attacker manages to recover $tot^{4,5}$, she can determine whether $k = 1$ or $k = 2$. However, the ranges for $k > 2$ overlap, meaning that for a given $tot^{4,5}$, she has at least two possibilities for k. In our architecture, we have that $tot^{4,5} \in [74, 292]$. An attacker observing this number only knows that $9 > k > 2$. This multiplies by 6 the number of possible architectures for that set of layers. There are 13 convolutional layers in VGG16. If we extrapolate and assume we have 6 possibilities for each layer, the reduced search space is multiplied by 6^{13}. This is an overevaluation of the added uncertainty. But we see, based on our computations on layers 4 to 6, that because of the overlapping, we have at least two possibilities per layer for the filter value, when there was previously no uncertainty. This leads to a multiplication of the search space by at least 2^{13}. Let us note that it represents a high increase in the search space, as recovering the correct architecture then requires training all of the remaining possibilities.

In [141], the authors reduced the search space for VGG16 to 16 possibilities. With our protection, we can increase it to 2^{17}.

Furthermore, this computation does not provide the attacker with the padding, as she does not know the full output size, resulting in an even larger search space.

Protected Case, When an Attacker Can Only See Max Pooling Layers: In reality, as stated in the previous paragraph, it is difficult for an attacker to differentiate between convolutional layers. This means that $tot^{4,5}$ is not observable, and the attacker can only recover the number of multiplications required to reach the following max pooling layer $MaxPool$. We therefore need to compute $tot^{4,6}$.

If we denote (x, y) the last element in the first $MaxPool$ block B, and (x', y') the last element in $Conv5$ that needs to be computed for B to be ready, then:

$$tot^{4,6}_{x,y} = tot^{4,5}_{x',y'} + tot^{5,6}_{x,y}$$

.

This time, we consider the possible ranges of $tot^{4,6}$, given that in_5 and out_5 are not available to the attacker.

In our case, $MaxPool$ has window size $k^{(6)} = 2$, and does not require any padding.

Furthermore, all channels need to be computed at once in the pooling case. Thus, the coordinates of (x, y) in the output of $Conv5$ are:

$$\begin{aligned} row &= \left\lfloor \frac{k^{(6)} \cdot y}{n} \right\rfloor + \left\lfloor \frac{x}{k^{(6)}} \right\rfloor \\ col &= \left(y \mod \frac{n}{k^{(6)}} \right) \times k^{(6)} + x \mod k^{(6)} \\ c &= row \times n + col \end{aligned}$$

We also have that $b = \arg\min_{b'} \left(\sum_{q=0}^{b'} w_1 \geq c \right)$. Applying it to our case for $(x_{\min}, y_{\min}) =$

$(32, 32)$ and $(x_{\max}, y_{\max}) = (64, 64)$, we have:

$$
\begin{aligned}
row_{\min} &= \left\lfloor \frac{2 \cdot 32}{112} \right\rfloor + \left\lfloor \frac{32}{2} \right\rfloor && = 16 \\
col_{\min} &= \left(32 \mod \frac{112}{2} \right) \times 2 + 32 \mod 2 && = 64 \\
c_{\min} &= row \times 112 + col && = 1856 \\
b_{\min} &= \left\lceil \frac{c_{\min}}{32} \right\rceil && = 58 \\
row_{\max} &= \left\lfloor \frac{2 \cdot 64}{112} \right\rfloor + \left\lfloor \frac{64}{2} \right\rfloor && = 33 \\
col_{\max} &= \left(64 \mod \frac{112}{2} \right) \times 2 + 64 \mod 2 && = 16 \\
c_{\max} &= row \times 112 + col && = 3712 \\
b_{\max} &= \left\lceil \frac{c_{\max}}{64} \right\rceil && = 58
\end{aligned}
$$

In $O^{(5)}$, we need the element with coordinates $(ch, B \cdot b)$ where ch is the last filter of $Conv5$, since $MaxPool$ requires all channels to be completed. That element requires element $(x', y') = (in_5 \times k^{(5)} \times k^{(5)}, B \cdot b)$ in $I^{(5)}$. Indeed, the column number in $O^{(5)}$ and $I^{(5)}$ must be the same. Furthermore, the whole column must be computed, which is why $x' = in_5 \times k^{(5)} \times k^{(5)}$. Thus, the previous formulas directly provide us with (x', y'):

$$
\begin{aligned}
(x'_{\min}, y'_{\min}) &= (in_5 \times k^{(5)} \times k^{(5)} - 1, 32 \cdot b_{\min}) && = (128 \times 9 - 1, 1856) \\
(x'_{\max}, y'_{\max}) &= (in_5 \times k^{(5)} \times k^{(5)} - 1, 64 \cdot b_{\max}) && = (128 \times 9 - 1, 3712)
\end{aligned}
$$

Thus, given $Conv5$ has 128 input and output channels:

$$
\begin{aligned}
H^{(5)}_{\min} &= \left\lceil \frac{128 \times 9 - 1}{32} \right\rceil && = 36 \\
H^{(5)}_{\max} &= \left\lceil \frac{128 \times 9 - 1}{64} \right\rceil && = 18 \\
M^{(5)}_{\min} &= \left\lceil \frac{128}{32} \right\rceil && = 4 \\
M^{(5)}_{\max} &= \left\lceil \frac{128}{64} \right\rceil && = 2 \\
tot^{5,6}_{32,32} &= b_{\min} \times H^{(5)}_{\min} \times M^{(5)}_{\min} && = 8352 \\
tot^{5,6}_{64,64} &= b_{\max} \times H^{(5)}_{\max} \times M^{(5)}_{\max} && = 2088
\end{aligned}
$$

Here, we have $H^{(5)} \times M^{(5)}$ for width-wise block b as well because the whole b-th block width-wise needs to be computed, as well as all channels.

Applying the same process as previously to compute $tot^{4,5}_{e_{\min}}$ and $tot^{4,5}_{e_{\max}}$, we have:

$$
\begin{aligned}
tot^{4,5}_{x'_{\min}, y'_{\min}} &= 4338 \\
tot^{4,5}_{x'_{\max}, y'_{\max}} &= 1053
\end{aligned}
$$

Thus, we have:

$$tot^{4,6}_{x'_{\min},y'_{\min}} = tot^{5,6}_{x_{\min},y_{\min}} + tot^{4,5}_{in_5\times k^{(5)}\times k^{(5)},e_{\min}\cdot b_{\min}} = 12690$$
$$tot^{4,6}_{x'_{\max},y'_{\max}} = tot^{5,6}_{x_{\max},y_{\max}} + tot^{4,5}_{in_5\times k^{(5)}\times k^{(5)},e_{\max}\cdot b_{\max}} = 3141$$

Therefore, the total number of multiplications for one $MaxPool$ block to be ready ranges between $3,141$ and $12,690$ in the set of layers we study from the VGG16 architecture: $tot^{4,6} \in [3141, 12690]$.

Once again, the attacker does not know $k^{(4)}, k^{(5)}, k^{(6)}$ or the padding values. For each layer, $k^{(i)} \in \{1, ..., 11\}$. We also suppose, like before, that $k^{(4)} = k^{(5)}$. The padding p between $Conv4$ and $Conv5$ (the only one that intervenes in the computations) is such that $p < k^{(4)}$. We get tables of possible values depending on $k^{(4)} = k^{(5)}$, p and $k^{(6)}$. As before, we take the maximum and minimal values over all possible p. Table 6.2 considers the range of $tot^{4,6}$ depending on $k^{(4)}$.

Table 6.2: Maximal and minimal number of multiplications depending on the filter size, when the attacker cannot distinguish between convolutional layers but knows out_4, in_5 and out_5.

$k^{(4)}$	1	2	3	4	5	6
$tot^{4,6}_{x_{\min},y_{\min}}$	2,712	10,976	24,912	44,800	70,600	102,528
$tot^{4,6}_{x_{\max},y_{\max}}$	126	504	1,134	1,984	3,100	4,464
$k^{(4)}$	7	8	9	10	11	
$tot^{4,6}_{x_{\min},y_{\min}}$	141,120	185,856	237,168	295,200	360,096	
$tot^{4,6}_{x_{\max},y_{\max}}$	5,978	7,808	9,882	12,000	14,520	

Once again, the ranges for all possible $tot^{4,6}$ values overlap, making it harder for the attacker to determine the architecture. In our case, we had $tot^{4,6} \in [3141, 12041]$. An attacker could therefore only deduce that $1 < k^{(4)} < 11$. The range can be further deduced depending on the value actually observed, but there are at least two filter size values in every case. Furthermore, for most $k^{(6)}$ values, $2 < k^{(4)} < 10$.

So far, we had assumed, for simplicity, that the number of input and output channels are known for all layers. But it is generally not the case. Taking this last fact into account, the possible $tot^{4,6}$ values are given in Table 6.3.

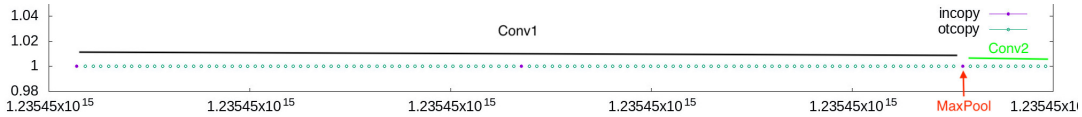
All $k^{(4)}$ values are possible. The attacker cannot determine $k^{(4)}$, $k^{(5)}$ or $k^{(6)}$. In all cases, we have at least 2 possibilities for each of the filter sizes. Since there are 18 max pooling and convolutional layers in VGG16, we can extrapolate once again and say that our protection could lead to a multiplication of the search space by 2^{18}. This leads to a reduced search space of 2^{22}. Let also remind the reader that $k^{(4)}$ and $k^{(5)}$ can actually differ, making a potential attacker's life even harder.

6.5.2 Performance Evaluation

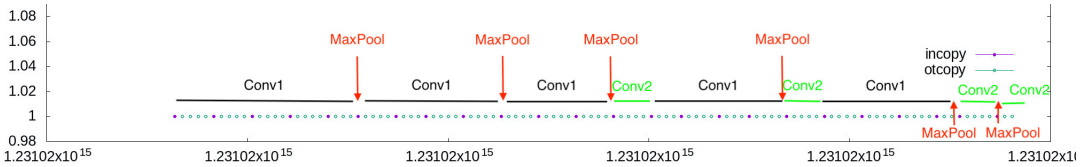
Experimental Platform. We launch our experiments on a DELL work station OptiPlex 7040 with a 4-core Intel Core i7 processor and three levels of cache of sizes,

Table 6.3: Maximal and minimal number of multiplications depending on the filter size, when the attacker cannot distinguish between convolutional layers.

$k^{(4)}$	1	2	3	4
$tot^{4,6}_{x_{\min},y_{\min}}$	$> 9.6 \cdot 10^6$	$> 10.5 \cdot 10^6$	$> 10 \cdot 10^6$	$> 11 \cdot 10^6$
$tot^{4,6}_{x_{\max},y_{\max}}$	42	42	44	44
$k^{(4)}$ 5	6	7	8	
$tot^{4,6}_{x_{\min},y_{\min}}$	$> 11 \cdot 10^6$	$> 12 \cdot 10^6$	$> 13.4 \cdot 10^6$	$> 15.8 \cdot 10^6$
$tot^{4,6}_{x_{\max},y_{\max}}$	46	48	54	64
$k^{(4)}$	9	10	11	
$tot^{4,6}_{x_{\min},y_{\min}}$	$> 19.2 \cdot 10^6$	$> 28.3 \cdot 10^6$	$> 54.7 \cdot 10^6$	
$tot^{4,6}_{x_{\max},y_{\max}}$	80	116	226	



(a) Simulation of a Flush and Reload attack on `itcopy` and `oncopy` patterns of an unprotected model.



(b) Simulation of a Flush and Reload attack on `itcopy` and `oncopy` patterns of a protected model.

Figure 6.7: Simulation of Flush and Reload on an unprotected (a) and a protected (b) model. The model considered has 2 convolutional layers separated by a max pooling one. Block sizes are between 32 and 38 in (b).

respectively, 32 KB, 256 KB and 8192 KB. Our experiments are carried out using Debian 4.19.152-1.

Performance. Let us consider a perfect attacker, who can recover the Flush and Reload traces without any noise. To simulate this, we use the GNU Debugger (GDB) to log all calls to `itcopy` and `oncopy`, along with the cycles at which they were called. Figure 6.7a shows such a simulation of an unprotected model.

Once again, we can see in Figure 6.7 that the layers' traces are interleaved, and it is difficult for an attacker to distinguish between the various layers.

Let us now consider our protection's overhead. Note that we do note take the creation of the architecture into account here, as it only needs to be executed once, rather than at each inference computation. We consider three architectures in our experiments. In all cases, there is only 1 input and output featuremap, the convolutional filter size is 3×3, the pooling window size is 2×2 and there is no padding.

The architectures are as follows :

1. Conv – MaxPool – Conv – MaxPool (`arch1`)
2. Conv – MaxPool – Conv (`arch2`)
3. Conv – Conv (`arch3`)

We average the execution time over 1,000 runs for several input sizes, as shown in Table 6.4. The time overhead depends heavily on the architecture's depth, the input size, the layer types and the block sizes.

The case $B = 1$ in Table 6.5 is included to show the importance of correctly selecting the minimal block size. The high overhead incurred in the case where one neuron is computed at a time confirms our assertion in Section 6.2.1, stating that considering one neuron at a time would take too long.

Let us now consider the case $B = 32$. In some cases, such as `arch2` with input shape $1 \times 20 \times 20$, the randomized protection has almost the same execution time as a normal execution.

We believe that the higher overhead in `arch3` is due to the lack of maxpooling layers: this leads to higher input sizes for the convolutional layers, and convolutional layers take naturally longer to compute than maxpooling.

Let us remind the reader that we did not use common frameworks such as PyTorch [98] or TensorFlow [83], as explained in Section 6.5.4. Moreover, because the GeMM operation is operated on a very low level and ensures an optimally filled cache, it is very efficient. It is not the case for our high-level block subdivisions and multiplications. Thus, since common ML frameworks are heavily optimized, and so is the GeMM operation, we believe that the worst case in Table 6.4 being 8 times more time consuming than an unprotected operation is reasonable, and the time overhead could be improved with further optimization. This is especially the case since we compare our implementation's execution time to a standard, highly optimized, Keras – with TensorFlow as a backend – one.

Furthermore, despite the incurred overhead, we believe that our countermeasure is a first important step towards a secure real time execution.

Table 6.4: Execution times for three different architectures, depending on the type of protection added. In all cases, the block size is $B = 32$.

Input shape	Protection Type	`arch1`		`arch2`		`arch3`	
$1 \times 28 \times 28$	None	/	/	$842\mu s$	1	$606\mu s$	1
	Blocks, no random	/	/	$1634\mu s$	$\times 1.94$	$4966\mu s$	$\times 8.19$
	Blocks, random	/	/	$1714\mu s$	$\times 2.04$	$4995\mu s$	$\times 8.24$
$1 \times 22 \times 22$	None	$698\mu s$	1	$610\mu s$	1	$764\mu s$	1
	Blocks, no random	$904\mu s$	$\times 1.30$	$927\mu s$	$\times 1.52$	$2340\mu s$	$\times 3.06$
	Blocks, random	$1004\mu s$	$\times 1.44$	$989\mu s$	$\times 1.62$	$2391\mu s$	$\times 3.13$
$1 \times 20 \times 20$	None	/	/	$635\mu s$	1	$542\mu s$	1
	Blocks, no random	/	/	$745\mu s$	$\times 1.17$	$1718\mu s$	$\times 3.17$
	Blocks, random	/	/	$816\mu s$	$\times 1.29$	$1765\mu s$	$\times 3.26$

Table 6.5: Execution times for three different architectures, depending on the block size considered. We compare the case $B = 1$ to the case $B = 32$ to show how crucial it is to select the correct block size. In all cases, we consider the random protection.

Block size	Input shape	`arch1`	`arch2`	`arch3`
1	$1 \times 28 \times 28$	/	$2539\mu s$	$50198\mu s$
	$1 \times 22 \times 22$	$1047\mu s$	$1006\mu s$	$13923\mu s$
	$1 \times 20 \times 20$	/	$842\mu s$	$9406\mu s$
32	$1 \times 28 \times 28$	/	$1714\mu s$	$4995\mu s$
	$1 \times 22 \times 22$	$1004\mu s$	$989\mu s$	$2391\mu s$
	$1 \times 20 \times 20$	/	$816\mu s$	$1765\mu s$

6.5.3 Discussion

Our approach is similar to [2] and [109] in the sense that we obtain the value of some neurons in layer $i + 1$ before the execution of layer i is done. But besides the fact that we do not consider a hardware context, our suggestion differs in that we introduce randomization in the computation of neurons: we do not compute the value of a neuron as soon as enough elements are ready. Rather, we compute them in a random way determined at the creation of the architecture. Moreover, our goal is different: we aim at increasing the security by mitigating cache attacks based on the GeMM computations during an NN inference.

One limitation in our method is the pooling layer. Indeed, because its execution differs from that of convolutional layers, and no GeMM is applied, a potential attacker can detect when an execution switches between a convolutional layer and a pooling one. As shown in Section 6.5.1, however, our methodology still mitigates Cache Telepathy in architectures with pooling layers. We believe that architectures such as Fully Convolutional Networks [75], which have several consecutive convolutional layers, could make the architecture almost completely leakage free.

The overhead induced depends on the block sizes and the model's depth. Balancing security of the protection – linked to the random block sizes – and the incurred overhead is however possible, as the said overhead is still manageable. We also believe that a better optimization of our implementation should reduce the observed overhead.

Even if our protection targets Cache Telepathy specifically, we believe it could be used against other side-channel attacks such as CSI [10], DeepRecon [53] or How to 0wn NAS [52]. Both CSI [10] and How to 0wn NAS [52] mention that reordering would indeed be a countermeasure to their method. CSI [10] bases its attack on the sequential nature of NNs. It proceeds layer by layer to find the number of neurons and the weights in each layer. In each layer, and for each multiplication $w \cdot x$ (where w is the weight and x is an input value), the attacker makes two hypotheses: either this multiplication takes place in layer i, or it takes place in layer $i + 1$. If the layers are not computed sequentially, then the hypotheses no longer make sense, as the multiplication could take place at a later layer. In the case of [52], layers as a whole are targeted. Indeed, specific functions corresponding to specific layers are monitored. Thus, splitting the layers would prevent them from targeting the

pre-existing functions.

6.5.4 Scope and Limitations

Let us now discuss the scope and limitations of our countermeasure.

First of all, because the proposed approach needs to compute layers in a non-sequential order, it does not apply to FC layers, even though Cache Telepathy targets them as well. Indeed, these layers require all neurons from the previous layer to be computed before they can start their execution. We therefore limit our countermeasure to CNNs. However, as CNNs are now widely used in various fields such as the medical one [101], image processing [13] or game playing [114], our countermeasure can still apply to many commonly used architectures [115, 48, 119].

Second, as in the Cache Telepathy paper [141], we place ourselves in the CPU context. Indeed, as mentioned in [53], monitoring GeMM functions requires them to be in the same instruction line in the cache for the victim and the attacker. They therefore need to run on the CPU. As mentioned in Section 2.2.4, several popular MLaaS frameworks [3] provide CPU computations for inference.

Third, instead of using high-level, highly optimized frameworks such as TensorFlow [83] or PyTorch [98], we wrote our own implementation in C++. We required more freedom when writing code, as commonly used functions in those frameworks apply to entire layers, when we needed to deal with neurons individually. Moreover, using C++ still enabled us to apply GeMM, which is the target of the attacker. This C++ implementation – necessarily less optimized than common ML frameworks – partially explains the overhead observed in Table 6.4 in Section 6.5.2.

Finally, as detailed in Section 6.5.2, our goal is to mitigate the Cache Telepathy attack by preventing an attacker from getting a tractable search space for the victim model's architecture. Thus, even though not all leakages are eliminated, an attacker still cannot recover the correct architecture without training over 2^{17} architectures, which is not feasible in a reasonable period of time.

6.6 Conclusion

While the first part of this dissertation focused on the direct protection of models' weights and biases, we discuss a way to defend the architecture in this second part. Not only does this help in the parameter protection and secure IP, but it also prevents other attacks such as membership inference attacks [113].

To achieve our goal, we propose Telepathic Headache, a protection against cache-based attacks targeting the GeMM algorithm used by most ML frameworks to implement FC and convolutional layers in a CPU setting. Our scheme consists in mixing a model's layer by computing neurons in blocks, depth-wise. As soon as certain random-sized blocks of values in a layer are ready, they are multiplied by the associated filter values and the results are sent to the next layer.

The security analysis has shown that our proposal leads to a multiplication by at least 2^{18} of the reduced search space obtained by Cache Telepathy on VGG16. Even though we have only considered the Cache Telepathy attack when measuring the efficiency of our defense, the methodology we have presented could also be used

against other timing-based SCAs, such as DeepRecon [53] and EM-based SCAs such as CSI Neural Network [10]. In fact, the authors of [10] mention in their discussions that randomizing the computation of neurons would thwart their attack.

Chapter 7: Conclusion

The increasing number of reverse-engineering attacks on NNs pose serious security issues, be it concerning the intellectual property of industries or the users' privacy. Because of the novelty of the attacks, not many countermeasures have appeared. Moreover, most existing protections are hardware-based. In this thesis, we provided three software-based countermeasures to reverse-engineering attacks. Our work also led us to apply one of our proposals to mitigate adversarial attacks.

7.1 Contributions

We started by introducing NNs and their structure, as well as the attacks and protections that have already been published. Let us note that we published a survey titled "Side-Channel Attacks for Architecture Extraction of Neural Networks", in *CAAI Transactions on Intelligence Technology* [158].

The first part was set in a gray-box context, where the attacker had partial knowledge about the target model. In Chapters 3 and 4, we proposed defenses against weight extraction attacks where the attacker had access to the architecture.

In Chapter 3, we first introduced the notion of parasitic models, a staple in Part I. We showed how those parasitic CNNs alter a model's internal structure, therefore making it harder to carry out mathematical weight extraction attacks. Indeed, the introduction of layers with *ReLU* activation functions creates hyperplanes associated to the new neurons being equal to 0. The non-linearity of activation functions ensure that the newly introduced hyperplanes bend the previously existing ones, thus enabling a modification of the internal structure. Due to their strong relation with the internal structure, we verified our claim by measuring the modification of adversarial examples. The results from this Chapter led to a publication in ICISSP 2021 [157], titled "A Protection against the Extraction of Neural Network Models".

A key element, dynamism, lacked in the contribution from Chapter 3. Indeed, the protection previously suggested did not prevent physical side-channel attacks, which can reconstruct an entire model based on power or electromagnetic traces. Selecting a different set of parasitic models at each run enabled us to hide the input of a target model in Chapter 4. By placing at least one parasite, selected at random, at the entrance of a model, we made sure a malicious user saw constantly changing input values, preventing her from extracting precise weight values. In this case, the overhead at runtime corresponds to the addition of only a few layers to the original model, as well as selecting a model at random among a pretrained set. The

latter is equivalent to randomly generating one integer. Overall, the overhead only corresponds to a small percentage of a large NN's computations. The content of this Chapter was published in SPACE 2021 [155], under the name "Parasite: Mitigating Physical Side-Channel Attacks Against Neural Networks".

When studying the impact of parasites on the internal structure of a model, we noticed that the change in structure is closely linked to adversarial example generation. This led us to the third contribution of this thesis, in Chapter 5. We used autoencoders as parasites to both learn the important features in images and add artificial noise. We showed the increased robustness that our proposal brought in two experiment settings. In the first, the attacker only had access to the base model. In the second, the attacker also knew that an autoencoder had been added to the original model. The content of this Chapter was submitted to a conference [159].

As the parameter protection can be achieved through architecture security and the architecture itself constitutes IP, we focused on a black-box model in the second part of this thesis. Because SCAs that aim at recovering the architecture of a model rely on the sequential execution of NN models, we proposed, in Chapter 6, to reorder computations to mitigate a cache-based attack [141]. To this effect, we considered blocks of neurons of random sizes. We computed neurons from the following layer in a depth-wise fashion – i.e., we computed a block as soon as it was ready. We published this work as "Telepathic Headache: Mitigating Cache Side-Channel Attacks on Convolutional Neural Networks" in ACNS 2021 [156].

In Chapter 4 and Chapter 6, we have made use of simulations for the potential attacker. However, in both cases, we assumed an attacker with perfect side-channel leakage traces, making our defense task harder and thus validating our results.

Although we have demonstrated the feasibility and efficiency of our countermeasures, they are not without disadvantages. While small parasitic models only introduce a few optimized layers, and their training can generally be made independent from the model to protect, using them is a sensitive task. Indeed, we have highlighted the importance of carefully selecting an appropriate architecture and of proper training. In all three Chapters using them – Chapters 3 to 5, we noted that a balance must be found between the drop in the accuracy and the protection provided.

7.2 Future Work

In Chapters 3 to 6, we proposed four countermeasures against reverse-engineering and adversarial attacks, and we showed their efficiency. As in any proposal, these defenses can be improved and built upon. In the wake of the work presented here, future research could focus on the following subjects:

- In Chapter 4, on top of hiding the input, one could hide the activation functions. This could be achieved by approximating them in a similar fashion as for the input. One could study the impact of training small CNNs that approximate the activation functions. For instance, one could train a CNN to approximate a noisy *ReLU* instead of a noisy identity function. Because *ReLUs* are more complex than the identity, it would also be interesting to

study whether *ReLU* is complex enough that a small CNN cannot very precisely approximate it, and therefore introduce noise.

- It is also important to note, in Chapter 4, that a balance between the number of added operations and a high entropy in the set of parasitic CNNs needs to be reached. For this reason, further work could conduct a study on which architecture could best approximate a noisy identity while minimising the number of parasitic operations. For instance, one could consider a smaller filter size in the countermeasure: 3×3 instead of 5×5. One could also increase the stride of the introduced convolutional layers. Indeed, the larger the filter and the higher the number of layers, the higher the number of operations. But limiting the possible filters and/or the number of layers decreases the size of the parasitic CNN set. Hence, further testing could be carried out to determine the right balance.

- In Chapter 5, we train parasitic autoencoders on the output of the layers after which we wish to place them. However, one could train them on one normalized channel at a time. Such autoencoders could then be placed anywhere in the model. Moreover, they could only be applied to part of a layer. But when doing so, one should take care that the trained autoencoders do not lead to a noticeable drop in the base model's accuracy.

- In general, one could study the impact of the various parasitic parameters for different ML models. One interesting lead could be to set the parasites to be NNs with numerous architectures – varying number of layers, of filter sizes, of activation functions,... – and determine the impact of each parameter on the accuracy and robustness against reverse-engineering and adversarial attacks. Another option would be to make a full analysis of the best locations where one should introduce the models, depending on the task at hand. In our dissertation, we placed them near the entrance of the model, as we observed it led to a small drop in the accuracy. However, applying parasites in the middle of the base model, to some carefully selected neurons could yield better results. A third possibility would be to determine the ideal number of parasites to incorporate in a target model M, taking into account M's characteristics, the incurred time overhead and the induced drop in the accuracy. Other similar approaches could be carried out so as to conduct a full analysis of the impact of parasitic models on both the base model's accuracy and the resistance to reverse-engineering and adversarial attacks.

- In Chapter 3, we impose constraints on the training of parasites so as to ensure the newly added hyperplanes are visible to the attacker. Improving these constraints might intensify the effect of the parasitic hyperplanes. We made the choice to train the parasites independently from the base model, so that they can generalize to any target. But if we remove this condition, we could, for example, determine the noise so as to maximize the impact on later layers. Indeed, we explained that a hyperplane from layer l bends hyperplanes from layers $l+1$ and deeper. We can therefore determine a loss function $\mathcal{L}(\theta, \theta', x, y)$

such that minimizing $\mathcal{L}$ maximizes the intersections between neurons from the base model θ and the parasitic one θ'.

- In Chapter 6, we propose to reorder a CNN's neuron computations. While we deem the incurred overhead acceptable in our experiments, some architectures presented a higher overhead than others. Further work could therefore consist in a deeper analysis of the overhead depending on the architecture at hand. It might help make the proposal more time efficient. Moreover, testing the effectiveness of the reordering against other reverse-engineering attacks such as CSI NN [10], DeepRecon [53] or How to Own NAS [52] might confirm our beliefs that our countermeasure should also thwart them.

- In this thesis, we have not considered binary or quantized NNs [57, 47, 41, 152, 59]. In BNNs, the weights and outputs of activation functions are binary while in quantized networks, these values are quantized. The two types of networks enable other types of security measures. The authors of [31, 30] propose a hardware masking countermeasure for BNNs. We could therefore apply such common cryptographic countermeasures to the software setting. As some models [115, 48] have millions of parameters, masking the entire model in the software case would be impractical. However, we show in [154] that some parameters are more sensitive to change. Adding noise to those parameters leads to a larger drop in the model's accuracy. We used that observation to provide *premium* users with a higher accuracy model, when common users had access to a functioning but lower accuracy one. Further work could build on this and perform a full analysis on the neurons' sensitivity from a range of NN models. Once the analysis is done, we could select the most sensitive neurons and encrypt those. With a sufficiently large set of encrypted parameters, an attacker might not be able to access a model with a suitable accuracy. In this quantized context, another line of research could consist in using look-up tables (LUTs) for activation functions. This could prevent timing, magnetic and power SCAs extracting them [10].

- When it comes to quantized NNs, an interesting line of further research would also be to investigate the impact of mounting the mathematical attack described in Section 2.3.2 against quantized NNs. Indeed, in the non-quantized case, a great care should be taken when dealing with floating-point imprecision with real numbers machine representation. Today, quantized NNs share almost the same accuracy as the floating-point ones, but do not require such a high precision. This lead could bring us closer to differential cryptanalysis and make us explore alternative cryptanalytic defenses.

- Finally, it would also be interesting to consider mixing existing countermeasures with our proposals. For instance, we could ensure constant-time behaviour for activation functions (as proposed in [124]) or through LUTs, reorder computations and introduce dynamic parasites. In the wake of Chapter 5, one could also mix dynamism with the preexisting autoencoders from [8].

Article References

[1] Md. Zahangir Alam, M. Saifur Rahman, and M. Sohel Rahman. "A Random Forest based predictor for medical data classification using feature ranking". In: *Informatics in Medicine Unlocked* 15 (2019), p. 100180. ISSN: 2352-9148.

[2] Manoj Alwani, Han Chen, Michael Ferdman, and Peter A. Milder. "Fused-layer CNN accelerators". In: *MICRO*. IEEE Computer Society, 2016, 22:1–22:12.

[3] Amazon. *SageMaker ML InstanceTypes*. 2018. URL: `https://aws.amazon.com/sagemaker/pricing/instance-types/`.

[4] Filip Andersson and Simon Arvidsson. *Generative Adversarial Networks for photo to Hayao Miyazaki style cartoons*. 2020. arXiv: `2005.07702 [cs.GR]`.

[5] Maksym Andriushchenko, Francesco Croce, Nicolas Flammarion, and Matthias Hein. "Square Attack: A Query-Efficient Black-Box Adversarial Attack via Random Search". In: *ECCV (23)*. Vol. 12368. Lecture Notes in Computer Science. Springer, 2020, pp. 484–501.

[6] Alessio Ansuini, Alessandro Laio, Jakob H. Macke, and Davide Zoccolan. "Intrinsic dimension of data representations in deep neural networks". In: *NeurIPS*. 2019, pp. 6109–6119.

[7] Anish Athalye, Logan Engstrom, Andrew Ilyas, and Kevin Kwok. "Synthesizing Robust Adversarial Examples". In: *ICML*. Vol. 80. Proceedings of Machine Learning Research. PMLR, 2018, pp. 284–293.

[8] Yassine Bakhti, Sid Ahmed Fezza, Wassim Hamidouche, and Olivier Déforges. "DDSA: A Defense Against Adversarial Attacks Using Deep Denoising Sparse Autoencoder". In: *IEEE Access* 7 (2019), pp. 160397–160407.

[9] Yassine Bakhti, Sid Ahmed Fezza, Wassim Hamidouche, and Olivier Déforges. "DDSA: A Defense Against Adversarial Attacks Using Deep Denoising Sparse Autoencoder". In: *IEEE Access* 7 (2019), pp. 160397–160407.

[10] Lejla Batina, Shivam Bhasin, Dirmanto Jap, and Stjepan Picek. "CSI NN: Reverse Engineering of Neural Network Architectures Through Electromagnetic Side Channel". In: *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*. USENIX Association), 2019, pp. 515–532.

[11] Alex Biryukov, Daniel Dinu, Yann Le Corre, and Aleksei Udovenko. "Optimal First-Order Boolean Masking for Embedded IoT Devices". In: *Smart Card Research and Advanced Applications - 16th International Conference, CARDIS 2017, Lugano, Switzerland, November 13-15, 2017, Revised Selected Papers*. Ed. by Thomas Eisenbarth and Yannick Teglia. Vol. 10728. Lecture Notes in Computer Science. Springer, 2017, pp. 22–41.

[12] Eric Brier, Christophe Clavier, and Francis Olivier. "Correlation power analysis with a leakage model". In: *International workshop on cryptographic hardware and embedded systems*. Springer. 2004, pp. 16–29.

[13] Andrew Bud. "Facing the future: the impact of Apple FaceID". In: *Biometric Technology Today* 2018 (2018), pp. 5–7.

[14] Giovanni Camurati, Sebastian Poeplau, Marius Muench, Tom Hayes, and Aurélien Francillon. "Screaming Channels: When Electromagnetic Side Channels Meet Radio Transceivers". In: *CCS*. ACM, 2018, pp. 163–177.

[15] "Can one hear the shape of a neural network?: Snooping the GPU via Magnetic Side Channel". In: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022. URL: `https://www.usenix.org/conference/usenixsecurity22/presentation/maia`.

[16] Nicholas Carlini, Matthew Jagielski, and Ilya Mironov. "Cryptanalytic Extraction of Neural Network Models". In: *CRYPTO (3)*. Vol. 12172. Lecture Notes in Computer Science. Springer, 2020, pp. 189–218.

[17] Nicholas Carlini, Guy Katz, Clark W. Barrett, and David L. Dill. "Ground-Truth Adversarial Examples". In: *CoRR* abs/1709.10207 (2017).

[18] Nicholas Carlini and David A. Wagner. "Towards Evaluating the Robustness of Neural Networks". In: *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2017, pp. 39–57.

[19] Jianbo Chen, Michael I. Jordan, and Martin J. Wainwright. "HopSkipJumpAttack: A Query-Efficient Decision-Based Attack". In: *IEEE Symposium on Security and Privacy*. IEEE, 2020, pp. 1277–1294.

[20] Tianlong Chen, Yu Cheng, Zhe Gan, Jingjing Liu, and Zhangyang Wang. "Ultra-Data-Efficient GAN Training: Drawing A Lottery Ticket First, Then Training It Toughly". In: *CoRR* abs/2103.00397 (2021).

[21] Lukasz Chmielewski and Leo Weissbart. "On Reverse Engineering Neural Network Implementation on GPU". In: *IACR Cryptol. ePrint Arch.* 2021 (2021), p. 720.

[22] Francois Chollet et al. *Keras*. 2015. URL: `https://github.com/fchollet/keras`.

[23] Nello Cristianini and John Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods*. Cambridge University Press, 2010.

[24] Francesco Croce, Maksym Andriushchenko, Vikash Sehwag, Nicolas Flammarion, Mung Chiang, Prateek Mittal, and Matthias Hein. "RobustBench: a standardized adversarial robustness benchmark". In: *CoRR* abs/2010.09670 (2020).

[25] Francesco Croce and Matthias Hein. "Mind the box: l_1-APGD for sparse adversarial attacks on image classifiers". In: *ICML*. 2021.

[26] Francesco Croce and Matthias Hein. "Minimally distorted Adversarial Examples with a Fast Adaptive Boundary Attack". In: *ICML*. Vol. 119. Proceedings of Machine Learning Research. PMLR, 2020, pp. 2196–2205.

[27] Amit Daniely and Elad Granot. "An Exact Poly-Time Membership-Queries Algorithm for Extraction a three-Layer ReLU Network". In: *CoRR* abs/2105.09673 (2021).

[28] Li Deng. "The mnist database of handwritten digit images for machine learning research". In: *IEEE Signal Processing Magazine* 29.6 (2012), pp. 141–142.

[29] Anuj Dubey, Afzal Ahmad, Muhammad Adeel Pasha, Rosario Cammarota, and Aydin Aysu. "ModuloNET: Neural Networks Meet Modular Arithmetic for Efficient Hardware Masking". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2022.1 (2022), pp. 506–556.

[30] Anuj Dubey, Rosario Cammarota, and Aydin Aysu. "BoMaNet: Boolean Masking of an Entire Neural Network". In: *CoRR* abs/2006.09532 (2020).

[31] Anuj Dubey, Rosario Cammarota, and Aydin Aysu. "MaskedNet: A Pathway for Secure Inference against Power Side-Channel Attacks". In: *CoRR* abs/1910.13063 (2019).

[32] Vasisht Duddu, Debasis Samanta, D. Vijay Rao, and Valentina E. Balas. "Stealing Neural Networks via Timing Side Channels". In: *CoRR* abs/1812.11720 (2018).

[33] D. Eastlake and P. Jones. "RFC3174: US Secure Hash Algorithm 1 (SHA1)". In: (2001).

[34] Manuel Eberl. "Fisher-Yates shuffle". In: *Arch. Formal Proofs* 2016 (2016).

[35] Ronen Eldan and Ohad Shamir. "The Power of Depth for Feedforward Neural Networks". In: *COLT*. Vol. 49. JMLR Workshop and Conference Proceedings. JMLR.org, 2016, pp. 907–940.

[36] Lasse Espeholt, Shreya Agrawal, Casper Kaae Sønderby, Manoj Kumar, Jonathan Heek, Carla Bromberg, Cenk Gazen, Jason Hickey, Aaron Bell, and Nal Kalchbrenner. "Skillful Twelve Hour Precipitation Forecasts using Large Context Neural Networks". In: *CoRR* abs/2111.07470 (2021).

[37] Tombul Fatih and Cakar Bekir. "Police Use of Technology to Fight Against Crime". In: *European Scientific Journal, ESJ* 11.10 (2015).

[38] Xinyang Feng, Dongjin Song, Yuncong Chen, Zhengzhang Chen, Jingchao Ni, and Haifeng Chen. "Convolutional Transformer based Dual Discriminator Generative Adversarial Networks for Video Anomaly Detection". In: *ACM Multimedia*. ACM, 2021, pp. 5546–5554.

[39] Jan-Mark Geusebroek, Gertjan J. Burghouts, and Arnold W. M. Smeulders. "The Amsterdam Library of Object Images". In: *Int. J. Comput. Vision* 61.1 (2005), pp. 103–112. ISSN: 0920-5691.

[40] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. "Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization". In: *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*. USENIX Association), 2012, pp. 475–490.

[41] Yunchao Gong, Liu Liu, Ming Yang, and Lubomir D. Bourdev. "Compressing Deep Convolutional Networks using Vector Quantization". In: *CoRR* abs/1412.6115 (2014).

[42] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[43] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. "Explaining and Harnessing Adversarial Examples". In: *ICLR (Poster)*. 2015.

[44] Michael T. Goodrich and Michael Mitzenmacher. "Anonymous Card Shuffling and Its Applications to Parallel Mixnets". In: *Automata, Languages, and Programming - 39th International Colloquium, ICALP 2012, Warwick, UK, July 9-13, 2012, Proceedings, Part II*. Vol. 7392. Lecture Notes in Computer Science. Springer), 2012, pp. 549–560.

[45] Kazushige Goto and Robert A. van de Geijn. "Anatomy of high-performance matrix multiplication". In: *ACM Trans. Math. Softw.* 34.3 (2008), 12:1–12:25.

[46] Yiwen Guo, Chao Zhang, Changshui Zhang, and Yurong Chen. "Sparse DNNs with Improved Adversarial Robustness". In: *NeurIPS*. 2018, pp. 240–249.

[47] Song Han, Huizi Mao, and William J. Dally. "Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding". In: *ICLR*. 2016.

[48] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep Residual Learning for Image Recognition". In: *CVPR*. IEEE Computer Society, 2016, pp. 770–778.

[49] Zhezhi He, Adnan Siraj Rakin, and Deliang Fan. "Parametric Noise Injection: Trainable Randomness to Improve Deep Neural Network Robustness Against Adversarial Attack". In: *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*. Computer Vision Foundation / IEEE, 2019, pp. 588–597.

[50] G E Hinton and R R Salakhutdinov. "Reducing the dimensionality of data with neural networks". In: *Science* 313.5786 (July 2006), pp. 504–507.

[51] Geoffrey E. Hinton and Richard S. Zemel. "Autoencoders, Minimum Description Length and Helmholtz Free Energy". In: *NIPS*. Morgan Kaufmann, 1993, pp. 3–10.

[52] Sanghyun Hong, Michael Davinroy, Yigitcan Kaya, Dana Dachman-Soled, and Tudor Dumitras. "How to 0wn the NAS in Your Spare Time". In: *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020.

[53] Sanghyun Hong, Michael Davinroy, Yigitcan Kaya, Stuart Nevans Locke, Ian Rackow, Kevin Kulda, Dana Dachman-Soled, and Tudor Dumitras. "Security Analysis of Deep Neural Networks Operating in the Presence of Cache Side-Channel Attacks". In: *CoRR* abs/1810.03487 (2018).

[54] Jeremy Howard. *Imagenette*. URL: `https://github.com/fastai/imagenette/`.

[55] Xing Hu, Ling Liang, Shuangchen Li, Lei Deng, Pengfei Zuo, Yu Ji, Xinfeng Xie, Yufei Ding, Chang Liu, Timothy Sherwood, and Yuan Xie. "DeepSniffer: A DNN Model Extraction Framework Based on Learning Architectural Hints". In: *ASPLOS*. ACM), 2020, pp. 385–399.

[56] Weizhe Hua, Zhiru Zhang, and G. Edward Suh. "Reverse engineering convolutional neural networks through side-channel information leaks". In: *DAC*. ACM), 2018, 4:1–4:6.

[57] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. "Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations". In: *J. Mach. Learn. Res.* 18 (2017), 187:1–187:30.

[58] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*. Vol. 37. JMLR Workshop and Conference Proceedings. JMLR.org), 2015, pp. 448–456.

[59] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew G. Howard, Hartwig Adam, and Dmitry Kalenichenko. "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference". In: *CVPR*. IEEE Computer Society, 2018, pp. 2704–2713.

[60] Matthew Jagielski, Nicholas Carlini, David Berthelot, Alex Kurakin, and Nicolas Papernot. "High Accuracy and High Fidelity Extraction of Neural Networks". In: *USENIX Security Symposium*. USENIX Association, 2020, pp. 1345–1362.

[61] Matthew Jagielski, Nicholas Carlini, David Berthelot, Alex Kurakin, and Nicolas Papernot. "High Accuracy and High Fidelity Extraction of Neural Networks". In: *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*. USENIX Association, 2020, pp. 1345–1362.

[62] K. Janocha and W. M. Czarnecki. "On Loss Functions for Deep Neural Networks in Classification". In: *arXiv:1702.05659v1* (2017).

[63] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. "Caffe: Convolutional Architecture for Fast Feature Embedding". In: *arXiv preprint arXiv:1408.5093* (2014).

[64] Kazuya Kakizaki, Taiki Miyagawa, Inderjeet Singh, and Jun Sakuma. "Toward Practical Adversarial Attacks on Face Verification Systems". In: *BIOSIG*. Vol. P-315. LNI. Gesellschaft für Informatik e.V., 2021, pp. 113–124.

[65] Md Rezaul Karim, Oya Beyan, Achille Zappa, Ivan G Costa, Dietrich Rebholz-Schuhmann, Michael Cochez, and Stefan Decker. "Deep learning-based clustering approaches for bioinformatics". In: *Briefings in Bioinformatics* 22.1 (2020), pp. 393–415.

[66] Sanjay Kariyappa, Atul Prakash, and Moinuddin K. Qureshi. "MAZE: Data-Free Model Stealing Attack Using Zeroth-Order Gradient Estimation". In: *CVPR*. Computer Vision Foundation / IEEE, 2021, pp. 13814–13823.

[67] A. Karpathy. "Convolutional Neural Networks: Architectures, Convolution / Pooling Layers". CS231n Convolutional Neural Networks for Visual Recognition.

[68] Ilyes Khemakhem, Diederik P. Kingma, Ricardo Pio Monti, and Aapo Hyvärinen. "Variational Autoencoders and Nonlinear ICA: A Unifying Framework". In: *AISTATS*. Vol. 108. Proceedings of Machine Learning Research. PMLR, 2020, pp. 2207–2217.

[69] Wenjun Kou, Dustin A. Carlson, Alexandra J. Baumann, Erica Donnan, Yuan Luo, John E. Pandolfino, and Mozziyar Etemadi. "A deep-learning-based unsupervised model on esophageal manometry using variational autoencoder". In: *Artif. Intell. Medicine* 112 (2021), p. 102006.

[70] A. Krizhevsky, I. Sustskever, and G. E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in neural information processing systems* (2012), pp. 1097–1105.

[71] Alex Krizhevsky. *Learning multiple layers of features from tiny images*. Tech. rep. 2009.

[72] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[73] Alexander LeNail. "NN-SVG: Publication-Ready Neural Network Architecture Schematics". In: *Journal of Open Source Software* 4.33 (2019), p. 747.

[74] Moshe Leshno, Vladimir Ya. Lin, Allan Pinkus, and Shimon Schocken. "Multilayer feedforward networks with a nonpolynomial activation function can approximate any function". In: *Neural Networks* 6.6 (1993), pp. 861–867.

[75] Jeng-Hau Lin, Atieh Lotfi, Vahideh Akhlaghi, Zhuowen Tu, and Rajesh K. Gupta. "Accelerating Local Binary Pattern Networks with Software-Programmable FPGAs". In: *Design, Automation & Test in Europe Conference & Exhibition, DATE 2019, Florence, Italy, March 25-29, 2019*. Ed. by Jürgen Teich and Franco Fummi. IEEE, 2019, pp. 1112–1117.

[76] Xuanqing Liu, Minhao Cheng, Huan Zhang, and Cho-Jui Hsieh. "Towards Robust Neural Networks via Random Self-ensemble". In: *ECCV (7)*. Vol. 11211. Lecture Notes in Computer Science. Springer, 2018, pp. 381–397.

[77] Xuanqing Liu, Yao Li, Chongruo Wu, and Cho-Jui Hsieh. "Adv-BNN: Improved Adversarial Defense through Robust Bayesian Neural Network". In: *ICLR (Poster)*. OpenReview.net, 2019.

[78] Yuntao Liu, Dana Dachman-Soled, and Ankur Srivastava. "Mitigating Reverse Engineering Attacks on Deep Neural Networks". In: *2019 IEEE Computer Society Annual Symposium on VLSI, ISVLSI 2019, Miami, FL, USA, July 15-17, 2019*. IEEE), 2019, pp. 657–662.

[79] Yuntao Liu and Ankur Srivastava. "GANRED: GAN-based Reverse Engineering of DNNs via Cache Side-Channel". In: *IACR Cryptol. ePrint Arch.* 2020 (2020), p. 1014.

[80] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. "Towards Deep Learning Models Resistant to Adversarial Attacks". In: *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018.

[81] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. "Towards Deep Learning Models Resistant to Adversarial Attacks". In: *ICLR (Poster)*. OpenReview.net, 2018.

[82] Saurav Maji, Utsav Banerjee, and Anantha P. Chandrakasan. "Leaky Nets: Recovering Embedded Neural Network Models and Inputs Through Simple Power and Timing Side-Channels - Attacks and Defenses". In: *IEEE Internet Things J.* 8.15 (2021), pp. 12079–12092.

[83] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: `https://www.tensorflow.org/`.

[84] Takashi Matsubara, Ryo Akita, and Kuniaki Uehara. "Stock Price Prediction by Deep Neural Generative Model of News Articles". In: *IEICE Trans. Inf. Syst.* 101-D.4 (2018), pp. 901–908.

[85] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. "Innovative instructions and software model for isolated execution." In: *Hasp@ isca* 10.1 (2013).

[86] Dongyu Meng and Hao Chen. "MagNet: A Two-Pronged Defense against Adversarial Examples". In: *CCS*. ACM, 2017, pp. 135–147.

[87] Smitha Milli, Ludwig Schmidt, Anca D. Dragan, and Moritz Hardt. "Model Reconstruction from Model Explanations". In: *FAT*. ACM, 2019, pp. 1–9.

[88] Smitha Milli, Ludwig Schmidt, Anca D. Dragan, and Moritz Hardt. "Model Reconstruction from Model Explanations". In: *FAT*. ACM), 2019, pp. 1–9.

[89] Takayuki Miura, Satoshi Hasegawa, and Toshiki Shibahara. "MEGEX: Data-Free Model Extraction Attack against Gradient-Based Explainable AI". In: *CoRR* abs/2107.08909 (2021).

[90] Takeru Miyato, Shin-ichi Maeda, Masanori Koyama, and Shin Ishii. "Virtual Adversarial Training: A Regularization Method for Supervised and Semi-Supervised Learning". In: *IEEE Trans. Pattern Anal. Mach. Intell.* 41.8 (2019), pp. 1979–1993.

[91] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. "DeepFool: A Simple and Accurate Method to Fool Deep Neural Networks". In: *CVPR*. IEEE Computer Society, 2016, pp. 2574–2582.

[92] Smita Nayak and Rajesh Kumar Das. "Application of Artificial Intelligence (AI) in Prosthetic and Orthotic Rehabilitation". In: *Service Robotics*. Ed. by Volkan Sezer, Sinan Öncü, and Pınar Boyraz Baykas. Rijeka, 2020. Chap. 2.

[93] S. J. Oh, M. Augustin, B. Schiele, and M. Fritz. "Towards Reverse-engineering Black-Box Neural Networks". In: *International Conference on Learning Representations* (2018).

[94] Tribhuvanesh Orekondy, Bernt Schiele, and Mario Fritz. "Knockoff Nets: Stealing Functionality of Black-Box Models". In: *CVPR*. Computer Vision Foundation / IEEE, 2019, pp. 4954–4963.

[95] Dag Arne Osvik, Adi Shamir, and Eran Tromer. "Cache attacks and countermeasures: the case of AES". In: *Cryptographers' track at the RSA conference*. Springer. 2006, pp. 1–20.

[96] Utku Ozbulak, Manvel Gasparyan, Wesley De Neve, and Arnout Van Messem. "Perturbation analysis of gradient-based adversarial attacks". In: *Pattern Recognition Letters* 135 (2020), pp. 313–320. ISSN: 0167-8655.

[97] Omkar M Parkhi, Andrea Vedaldi, Andrew Zisserman, and C. V. Jawahar. "Cats and dogs". In: *2012 IEEE Conference on Computer Vision and Pattern Recognition*. 2012, pp. 3498–3505.

[98] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett. Curran Associates, Inc., 2019, pp. 8024–8035.

[99] Sylvestre-Alvise Rebuffi, Sven Gowal, Dan A. Calian, Florian Stimberg, Olivia Wiles, and Timothy A. Mann. "Fixing Data Augmentation to Improve Adversarial Robustness". In: *CoRR* abs/2103.01946 (2021).

[100] Kui Ren, Tianhang Zheng, Zhan Qin, and Xue Liu. "Adversarial Attacks and Defenses in Deep Learning". In: *Engineering* 6.3 (2020), pp. 346–360. ISSN: 2095-8099.

[101] Jonathan G. Richens, Ciarán M. Lee, and Saurabh Johri. "Improving the accuracy of medical diagnosis with causal machine learning". In: *Nature Communications* 11 (2020).

[102] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds". In: *Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS 2009, Chicago, Illinois, USA, November 9-13, 2009*. ACM), 2009, pp. 199–212.

[103] David Rolnick and Konrad P. Kording. "Reverse-engineering deep ReLU networks". In: *ICML*. Vol. 119. Proceedings of Machine Learning Research. PMLR, 2020, pp. 8178–8187.

[104] Jérôme Rony, Luiz G. Hafemann, Luiz S. Oliveira, Ismail Ben Ayed, Robert Sabourin, and Eric Granger. "Decoupling Direction and Norm for Efficient Gradient-Based L2 Adversarial Attacks and Defenses". In: *CVPR*. Computer Vision Foundation / IEEE, 2019, pp. 4322–4330.

[105] Zuherman Rustam and Glori Stephani Saragih. "Predicting Bank Financial Failures using Random Forest". In: *IWBIS*. IEEE, 2018, pp. 81–86.

[106] Abhinav Sagar and Rajkumar Soundrapandiyan. "Semantic Segmentation With Multi Scale Spatial Attention For Self Driving Cars". In: *ICCVW*. IEEE, 2021, pp. 2650–2656.

[107] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. "MobileNetV2: Inverted Residuals and Linear Bottlenecks". In: *CVPR*. Computer Vision Foundation / IEEE Computer Society, 2018, pp. 4510–4520.

[108] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. "MobileNetV2: Inverted Residuals and Linear Bottlenecks". In: *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*. IEEE Computer Society, 2018, pp. 4510–4520.

[109] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R. Stanley Williams, and Vivek Srikumar. "ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars". In: *ISCA*. IEEE Computer Society, 2016, pp. 14–26.

[110] Neil Shah, Nandish Bhagat, and Manan Shah. "Crime forecasting: a machine learning and computer vision approach to crime prediction and prevention". In: *Vis. Comput. Ind. Biomed. Art* 4.1 (2021), p. 9.

[111] Adi Shamir, Itay Safran, Eyal Ronen, and Orr Dunkelman. "A Simple Explanation for the Existence of Adversarial Examples with Small Hamming Distance". In: *CoRR* abs/1901.10861 (2019).

[112] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. "Membership Inference Attacks Against Machine Learning Models". In: *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2017, pp. 3–18.

[113] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. "Membership Inference Attacks Against Machine Learning Models". In: *2017 IEEE Symposium on Security and Privacy (SP)*. 2017, pp. 3–18.

[114] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. "Mastering the game of Go with deep neural networks and tree search". In: *Nat.* 529.7587 (2016), pp. 484–489.

[115] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *ICLR*. 2015.

[116] Dawn Song, Kevin Eykholt, Ivan Evtimov, Earlence Fernandes, Bo Li, Amir Rahmati, Florian Tramèr, Atul Prakash, and Tadayoshi Kohno. "Physical Adversarial Examples for Object Detectors". In: *WOOT @ USENIX Security Symposium*. USENIX Association, 2018.

[117] Xinkai Song, Tian Zhi, Zhe Fan, Zhenxing Zhang, Xi Zeng, Wei Li, Xing Hu, Zidong Du, Qi Guo, and Yunji Chen. "Cambricon-G: A Polyvalent Energy-Efficient Accelerator for Dynamic Graph Neural Networks". In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 41.1 (2022), pp. 116–128.

[118] Zhichuang Sun, Ruimin Sun, Long Lu, and Alan Mislove. "Mind Your Weight(s): A Large-scale Study on Insufficient Machine Learning Model Protection in Mobile Apps". In: *USENIX Security Symposium*. USENIX Association, 2021, pp. 1955–1972.

[119] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. "Going deeper with convolutions". In: *CVPR*. IEEE Computer Society, 2015, pp. 1–9.

[120] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. "Intriguing properties of neural networks". In: *ICLR (Poster)*. 2014.

[121] Go Takatoi, Takeshi Sugawara, Kazuo Sakiyama, and Yang Li. "Simple Electromagnetic Analysis Against Activation Functions of Deep Neural Networks". In: *ACNS Workshops*. Vol. 12418. Lecture Notes in Computer Science. Springer, 2020, pp. 181–197.

[122] Gabriel Lins Tenório, Cristian E. Muñoz Villalobos, Leonardo A. Forero Mendoza, Eduardo Costa da Silva, and Wouter Caarls. "Improving Transfer Learning Performance: An Application in the Classification of Remote Sensing Data". In: *ICAART (2)*. SciTePress, 2019, pp. 174–183.

[123] Theano Development Team. "Theano: A Python framework for fast computation of mathematical expressions". In: *arXiv e-prints* abs/1605.02688 (2016).

[124] Shruti Tople, Karan Grover, Shweta Shinde, Ranjita Bhagwan, and Ramachandran Ramjee. "Privado: Practical and Secure DNN Inference". In: *CoRR* abs/1810.00602 (2018).

[125] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. "Stealing Machine Learning Models via Prediction APIs". In: *USENIX Security* (2016), pp. 5–7.

[126] Jean-Baptiste Truong, Pratyush Maini, Robert J. Walls, and Nicolas Papernot. "Data-Free Model Extraction". In: *CVPR*. Computer Vision Foundation / IEEE, 2021, pp. 4771–4780.

[127] C. Devi Arockia Vanitha, D. Devaraj, and M. Venkatesulu. "Gene Expression Data Classification Using Support Vector Machine and Mutual Information-based Gene Selection". In: *Procedia Computer Science* 47 (2015). Graph Algorithms, High Performance Implementations and Its Applications (ICGHIA 2014), pp. 13–21. ISSN: 1877-0509.

[128] Peter M. VanNostrand, Ioannis Kyriazis, Michelle Cheng, Tian Guo, and Robert J. Walls. "Confidential Deep Learning: Executing Proprietary Models on Untrusted Devices". In: *CoRR* abs/1908.10730 (2019).

[129] Binghui Wang and Neil Zhenqiang Gong. "Stealing Hyperparameters in Machine Learning". In: *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2018, pp. 36–52.

[130] Chen Wang, Jian Chen, Yang Yang, Xiaoqiang Ma, and Jiangchuan Liu. "Poisoning attacks and countermeasures in intelligent networks: Status quo and prospects". In: *Digital Communications and Networks* (2021). ISSN: 2352-8648.

[131] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. "Intel Math Kernel Library". In: *High-Performance Computing on the Intel R© Xeon Phi* 2018 (2014).

[132] Ruize Wang, Huanyu Wang, and Elena Dubrova. "Far Field EM Side-Channel Attack on AES Using Deep Learning". In: *ASHES@CCS*. ACM, 2020, pp. 35–44.

[133] Xingbin Wang, Rui Hou, Yifan Zhu, Jun Zhang, and Dan Meng. "NPU-Fort: a secure architecture of DNN accelerator against model inversion attack". In: *Proceedings of the 16th ACM International Conference on Computing Frontiers, CF 2019, Alghero, Italy, April 30 - May 2, 2019*. Ed. by Francesca Palumbo, Michela Becchi, Martin Schulz, and Kento Sato. ACM, 2019, pp. 190–196.

[134] Yizhen Wang, Somesh Jha, and Kamalika Chaudhuri. "Analyzing the Robustness of Nearest Neighbors to Adversarial Examples". In: *ICML*. Vol. 80. Proceedings of Machine Learning Research. PMLR, 2018, pp. 5120–5129.

[135] Junyi Wei, Yicheng Zhang, Zhe Zhou, Zhou Li, and Mohammad Abdullah Al Faruque. "Leaky DNN: Stealing Deep-Learning Model Secret with GPU Context-Switching Side-Channel". In: *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2020, pp. 125–137.

[136] Lingxiao Wei, Bo Luo, Yu Li, Yannan Liu, and Qiang Xu. "I Know What You See: Power Side-Channel Attack on Convolutional Neural Network Accelerators". In: *ACSAC*. ACM), 2018, pp. 393–406.

[137] Yun Xiang, Zhuangzhi Chen, Zuohui Chen, Zebin Fang, Haiyang Hao, Jinyin Chen, Yi Liu, Zhefu Wu, Qi Xuan, and Xiaoniu Yang. "Open DNN Box by Power Side-Channel Attack". In: *CoRR* abs/1907.10406 (2019).

[138] Kai Yuanqing Xiao, Vincent Tjeng, Nur Muhammad (Mahi) Shafiullah, and Aleksander Madry. "Training for Faster Adversarial Robustness Verification via Inducing ReLU Stability". In: *ICLR (Poster)*. OpenReview.net, 2019.

[139] Cihang Xie, Yuxin Wu, Laurens van der Maaten, Alan L. Yuille, and Kaiming He. "Feature Denoising for Improving Adversarial Robustness". In: *CVPR*. Computer Vision Foundation / IEEE, 2019, pp. 501–509.

[140] Qian Xu, Md Tanvir Arafin, and Gang Qu. "Security of Neural Networks from Hardware Perspective: A Survey and Beyond". In: *ASP-DAC*. ACM, 2021, pp. 449–454.

[141] Mengjia Yan, Christopher W. Fletcher, and Josep Torrellas. "Cache Telepathy: Leveraging Shared Resource Attacks to Learn DNN Architectures". In: *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*. Ed. by Srdjan Capkun and Franziska Roesner. USENIX Association, 2020, pp. 2003–2020.

[142] Yuval Yarom and Katrina Falkner. "FLUSH+ RELOAD: a high resolution, low noise, L3 cache side-channel attack". In: *23rd {USENIX} Security Symposium ({USENIX} Security 14)*. 2014, pp. 719–732.

[143] Ville Yli-Mäyry, Akira Ito, Naofumi Homma, Shivam Bhasin, and Dirmanto Jap. "Extraction of Binarized Neural Network Architecture and Secret Parameters Using Side-Channel Information". In: *ISCAS*. IEEE, 2021, pp. 1–5.

[144] Kota Yoshida, Takaya Kubota, Mitsuru Shiozaki, and Takeshi Fujino. "Model-Extraction Attack Against FPGA-DNN Accelerator Utilizing Correlation Electromagnetic Analysis". In: *FCCM*. IEEE, 2019, p. 318.

[145] Kota Yoshida, Mitsuru Shiozaki, Shunsuke Okura, Takaya Kubota, and Takeshi Fujino. "Model Reverse-Engineering Attack against Systolic-Array-Based DNN Accelerator Using Correlation Power Analysis". In: vol. 104-A. 1. 2021, pp. 152–161.

[146] Honggang Yu, Haocheng Ma, Kaichen Yang, Yiqiang Zhao, and Yier Jin. "DeepEM: Deep Neural Networks Model Recovery through EM Side-Channel Information Leakage". In: *HOST*. IEEE, 2020, pp. 209–218.

[147] Sergey Zagoruyko and Nikos Komodakis. "Wide Residual Networks". In: *BMVC*. BMVA Press, 2016.

[148] Chiyuan Zhang, Samy Bengio, Moritz Hardt, and Yoram Singer. "Identity Crisis: Memorization and Generalization under Extreme Overparameterization". In: *CoRR* abs/1902.04698 (2019).

[149] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. "Cross-Tenant Side-Channel Attacks in PaaS Clouds". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*. ACM), 2014, pp. 990–1003.

[150] Ding-Xuan Zhou. "Theory of deep convolutional neural networks: Downsampling". In: *Neural Networks* 124 (2020), pp. 319–327.

[151] Shengyuan Zhou, Qi Guo, Zidong Du, Dao-Fu Liu, Tianshi Chen, Ling Li, Shaoli Liu, Jinhong Zhou, Olivier Temam, Xiaobing Feng, Xuehai Zhou, and Yunji Chen. "ParaML: A Polyvalent Multicore Accelerator for Machine Learning". In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 39.9 (2020), pp. 1764–1777.

[152] Shuchang Zhou, Zekun Ni, Xinyu Zhou, He Wen, Yuxin Wu, and Yuheng Zou. "DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients". In: *CoRR* abs/1606.06160 (2016).

[153] Yuankun Zhu, Yueqiang Cheng, Husheng Zhou, and Yantao Lu. "Hermes Attack: Steal DNN Models with Lossless Inference Accuracy". In: *CoRR* abs/2006.12784 (2020).

My Contributions

Conferences

[154] Julien Bringer, Hervé Chabanne, and Linda Guiga. "Premium Access to Convolutional Neural Networks". In: *CRiSIS*. Vol. 12528. Lecture Notes in Computer Science. Springer, 2020, pp. 219–234.

[155] Hervé Chabanne, Jean-Luc Danger, Linda Guiga, and Ulrich Kühne. "Parasite: Mitigating Physical Side-Channel Attacks Against Neural Networks". In: *SPACE*. Vol. 13162. Lecture Notes in Computer Science. Springer, 2021, pp. 148–167.

[156] Hervé Chabanne, Jean-Luc Danger, Linda Guiga, and Ulrich Kühne. "Telepathic Headache: Mitigating Cache Side-Channel Attacks on Convolutional Neural Networks". In: *ACNS (1)*. Vol. 12726. Lecture Notes in Computer Science. Springer, 2021, pp. 363–392.

[157] Hervé Chabanne, Vincent Despiegel, and Linda Guiga. "A Protection against the Extraction of Neural Network Models". In: *ICISSP*. SCITEPRESS, 2021, pp. 258–269.

Journals

[158] Hervé Chabanne, Jean-Luc Danger, Linda Guiga, and Ulrich Kühne. "Side-channel attacks for architecture extraction of neural networks". In: *CAAI Transactions on Intelligence Technology* 6.1 (2021), pp. 3–16.

Submissions

[159] Hervé Chabanne, Vincent Despiegel, Stéphane Gentric, and Linda Guiga. "Let's Get Dynamic: Dynamic Autoencoders Against Adversarial Attacks". In: *Submission*. 2022.

[160] Hervé Chabanne, Vincent Despiegel, and Linda Guiga. "One Picture is Worth a Thousand Words: A New Wallet Recovery Process". In: *Submission*. 2022.

Filed Patents

[161] Sébastien Bahloul, Hervé Chabanne, and Linda Guiga. *Procédés d'enrôlement de données pour vérifier l'authenticité d'une donnée de sécurité ou de vérification de l'authenticité d'une donnée de sécurité*. 2021.

[162] Hervé Chabanne, Jean-Luc Danger, Linda Guiga, and Ulrich Kühne. *Procédé de traitement de données par un réseau de neurones artificiels avec exécutions groupées d'opérations individuelles pour éviter les attaques par canal auxiliaire, et système correspondant*. 2021.

[163] Hervé Chabanne, Vincent Despiegel, Stéphane Gentric, and Linda Guiga. *Procédé d'utilisation sécurisée d'un premier réseau de neurones sur une donnée d'entrée*. 2022.

[164] Hervé Chabanne and Linda Guiga. *Procédés d'utilisation sécurisée d'un premier réseau de neurones sur une donnée d'entrée, et d'apprentissage de paramètres d'un deuxième réseau de neurones*. 2022.

ECOLE
DOCTORALE

Titre : Protections Logicielles des Réseaux de Neurones Artificiels

Mots clés : Réseaux de neurones, Rétro-ingénierie, Analyse de canaux auxiliaires, Exemples contradictoires

Rêsumê : Les réseaux de neurones (NNs) sont très présents dans notre vie quotidienne, à travers les smartphones, la reconnaissance faciale et biométrique ou même le domaine médical. Leur sécurité est donc de la plus haute importance. Si de tels modèles fuitent, cela mettrait non seulement en péril la confidentialité de données sensibles, mais porterait aussi atteinte à la propriété intellectuelle.

La sélection d'une architecture adaptée et l'entraînement de ses paramètres prennent du temps - parfois des mois – et nécessitent d'importantes ressources informatiques. C'est pourquoi un NN constitue une propriété intellectuelle. En outre, une fois l'architecture et/ou les paramètres connus d'un utilisateur malveillant, de multiples attaques peuvent être menées, telles des attaques contradictoires. Un attaquant trompe alors le modèle en ajoutant à l'entrée un bruit indétectable par l'œil humain. Cela peut mener à des usurpations d'identité. Les attaques par adhésion, qui visent à divulguer des informations sur les données d'entraînement, sont également facilitées par un accès au modèle. Plus généralement, lorsqu'un utilisateur malveillant a accès à un modèle, il connaît les sorties du modèle, ce qui lui permet de le tromper plus facilement. La protection des NNs est donc primordiale. Mais depuis 2016, ils sont la cible d'attaques de rétro-ingénierie de plus en plus puissantes. Les attaques de rétro-ingénierie mathématique résolvent des équations ou étudient la structure interne d'un modèle pour révéler ses paramètres. Les attaques par canaux cachés exploitent des fuites dans l'implémentation d'un modèle – par exemple à travers le cache ou la consommation de puissance – pour extraire le modèle. Dans cette thèse, nous visons à protéger les NNs en modifiant leur structure interne et en changeant leur implémentation logicielle.

Nous proposons quatre nouvelles défenses. Les trois premières considèrent un contexte de boîte grise où l'attaquant a un accès partiel au modèle, et exploitent des modèles parasites pour contrer trois types d'attaques. Nous abordons d'abord des attaques mathématiques qui récupèrent les paramètres d'un modèle à partir de sa structure interne. Nous proposons d'ajouter un – ou plusieurs – réseaux de neurones par convolution (CNNs) parasites à divers endroits du modèle de base et de mesurer leur impact sur la structure en observant la modification des exemples contradictoires générés.

La méthode précédente ne permet pas de contrer les attaques par canaux cachés extrayant les paramètres par l'analyse de la consommation de puissance ou électromagnétique. Pour cela, nous proposons d'ajouter du dynamisme au protocole précédent. Au lieu de considérer un – ou plusieurs – parasite(s) fixe(s), nous incorporons différents parasites à chaque exécution, à l'entrée du modèle de base. Cela nous permet de cacher l'entrée, nécessaire à l'extraction précise des poids. Nous montrons l'impact de cette défense à travers deux attaques simulées.

Nous observons que les modèles parasites changent les exemples contradictoires. Notre troisième contribution découle de cela. Nous incorporons dynamiquement un autre type de parasite, des autoencodeurs, et montrons leur efficacité face à des attaques contradictoires courantes. Dans une deuxième partie, nous considérons un contexte de boîte noire où l'attaquant ne connaît ni l'architecture ni les paramètres. Les attaques d'extraction d'architecture reposent sur l'exécution séquentielle des NNs. La quatrième et dernière contribution que nous présentons dans cette thèse consiste à réordonner les calculs des neurones. Nous proposons de calculer les valeurs des neurones par blocs en profondeur, et d'ajouter de l'aléa. Nous prouvons que ce réarrangement des calculs empêche un attaquant de récupérer l'architecture du modèle initial.

ECOLE DOCTORALE

Title : Software Protections for Artificial Neural Networks

Keywords : Neural Networks, Reverse-Engineering, Side-channel analysis, Adversarial examples

Abstract : In a context where Neural Networks (NNs) are very present in our daily lives, be it through smartphones, face and biometrics recognition or even in the medical field, their security is of the utmost importance. If such models leak information, not only could it imperil the privacy of sensitive data, but it could also infringe on intellectual property.

Selecting the correct architecture and training the corresponding parameters is time-consuming – it can take months – and requires large computational resources. This is why an NN constitutes intellectual property. Moreover, once a malicious user knows the architecture and/or the parameters, multiple attacks can be carried out, such as adversarial ones. Adversarial attackers craft a malicious datapoint by adding a small noise to the original input, such that the noise is undetectable to the human eye but fools the model. Such attacks could be the basis of impersonations. Membership attacks, which aim at leaking information about the training dataset, are also facilitated by the knowledge of a model. More generally, when a malicious user has access to a model, she also has access to the manifold of the model's outputs, making it easier for her to fool the model.

Protecting NNs is therefore paramount. However, since 2016, they have been the target of increasingly powerful reverse-engineering attacks. Mathematical reverse-engineering attacks solve equations or study a model's internal structure to reveal its parameters. On the other hand, side-channel attacks exploit leaks in a model's implementation – such as in the cache or power consumption – to uncover the parameters and architecture. In this thesis, we seek to protect NN models by changing their internal structure and their software implementation.

To this aim, we propose four novel countermeasures. In the first three, we consider a gray-box context where the attacker has partial access to the model, and we leverage parasitic models to counter three types of attacks.

We first tackle mathematical attacks that recover a model's parameters based on its internal structure. We propose to add one – or multiple – parasitic Convolutional Neural Networks (CNNs) at various locations in the base model and measure the incurred change in the structure by observing the modification in generated adversarial samples.

However, the previous method does not thwart side-channel attacks that extract the parameters through the analysis of power or electromagnetic consumption. To mitigate such attacks, we propose to add dynamism to the previous protocol. Instead of considering one – or several – fixed parasite(s), we incorporate different parasites at each run, at the entrance of the base model. This enables us to hide a model's input, necessary for precise weight extraction. We show the impact of this dynamic incorporation through two simulated attacks.

Along the way, we observe that parasitic models affect adversarial examples. Our third contribution is derived from this, as we suggest a novel method to mitigate adversarial attacks. To this effect, we dynamically incorporate another type of parasite: autoencoders. We demonstrate the efficiency of this countermeasure against common adversarial attacks.

In a second part, we focus on a black-box context where the attacker knows neither the architecture nor the parameters. Architecture extraction attacks rely on the sequential execution of NNs. The fourth and last contribution we present in this thesis consists in reordering neuron computations. We propose to compute neuron values by blocks in a depth-first fashion, and add randomness to this execution. We prove that this new way of carrying out CNN computations prevents a potential attacker from recovering a small enough set of possible architectures for the initial model.

Institut Polytechnique de Paris
91120 Palaiseau, France