UNIVERSITY OF CALIFORNIA SAN DIEGO

Towards Holistic Secure and Trustworthy Deep Learning

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Electrical Engineering (Intelligent Systems, Robotics & Control)

by

Huili Chen

2022

University of California San Diego

2022

DEDICATION

To my beloved parents and boyfriend.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my Ph.D. advisor, Professor Farinaz Koushanfar, for her sincere support and valuable guidance throughout my Ph.D. study. Her dedication, professional insights, and enthusiastic passion for ground-breaking research have taught me valuable lessons and greatly influenced my life as an academic researcher. I will always be grateful for her advice and support.

I would like to thank my committee members, Professor Jishen Zhao, Professor Tara Javidi, and Professor Truong Nguyen for being part of my committees and for their insightful suggestions on my dissertation. I also want to express my sincere gratitude to my mentor Dr. Ro Cammarota at Intel Lab and Dr. Jie Ding at Amazon Alexa AI for their warm support and guidance that helped me to work on the cutting-edge research directions. I was fortunate to work with and learn from Professor Jishen Zhao, Professor Alexandra Dmitrienko, and Professor Ahmad-Reza Sadeghi on the interdisciplinary research between security and machine learning.

I am grateful that I had the opportunity and pleasure to collaborate with brilliant people throughout my Ph.D. experience. In particular, I would like to thank Dr. Bita Darvish Rouhani, Cheng Fu, Xinqiao Zhao, Mohammad Samragh, Mojan Javaheripi, Hamid Ghasemzadeh, Siam Hussian, and Shehzeen Hussain, Malhar Jere, Seira Hidano, Thien Nguyen, Yein Kim, and Diego Garcia for all their help and the joyful time we had together.

Last but not the least, I would like to express my sincere gratitude to my beloved parents for their unconditional love and persistent support, for always believing in me and for encouraging me to dream big and work hard to pursue my goals.

The material in this dissertation is based on the following papers that are published in conferences or journal venues.

Chapter 1 and 2, in part, has been published at (i) the Proceedings of the 2019 International Conference on Multimedia Retrieval (ICMR) and appeared as: Huili Chen, Bita Darvish Rouhani, Cheng Fu, Jishen Zhao, Farinaz Koushanfar, “DeepMarks: A Secure Fingerprinting Framework for Digital Rights Management of Deep Learning Models”, and (ii) 2019 ACM/IEEE 46th

Annual International Symposium on Computer Architecture (ISCA) and appeared as Huili Chen, Cheng Fu, Bita Darvish Rouhani, Jishen Zhao, Farinaz Koushanfar, “DeepAttest: An End-to-End Attestation Framework for Deep Neural Networks”, and (iii) 2020 INTERSPEECH and appeared as Huili Chen, Bita Darvish, Farinaz Koushanfar, “SpecMark: A Spectral Watermarking Framework for IP Protection of Speech Recognition Systems”, and (iv) the Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV), 2021 and appeared as Huili Chen, Cheng Fu, Jishen Zhao, Farinaz Koushanfar, “ProFlip: Targeted Trojan Attack with Progressive Bit Flips”, and (v) the Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI), 2019 and appeared as Huili Chen, Cheng Fu, Jishen Zhao, Farinaz Koushanfar, “DeepInspect: A Black-box Trojan Detection and Mitigation Framework for Deep Neural Networks”, and (vi) 2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD) and appeared as Huili Chen, Cheng Fu, Jishen Zhao, Farinaz Koushanfar, “GenUnlock: An Automated Genetic Algorithm Framework for Unlocking Logic Encryption”, and (vii) 2022 ACM Transactions on Embedded Computing Systems (TECS) and appeared as Huili Chen, Xinqiao Zhang, Ke Huang, Farinaz Koushanfar, “AdaTest: Reinforcement Learning and Adaptive Sampling for On-chip Hardware Trojan Detection”. The dissertation author was the primary author of these materials.

Chapter 3, in part, has been published at the Proceedings of 2019 International Conference on Multimedia Retrieval (ICMR) and appeared as: Huili Chen, Bita Darvish Rouhani, Cheng Fu, Jishen Zhao, Farinaz Koushanfar, “DeepMarks: A Secure Fingerprinting Framework for Digital Rights Management of Deep Learning Models”. The dissertation author was the primary author of this conference paper.

Chapter 4, in part, has been published at the 2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA) and appreared as Huili Chen, Cheng Fu, Bita Darvish Rouhani, Jishen Zhao, Farinaz Koushanfar, “DeepAttest: An End-to-End Attestation Framework for Deep Neural Networks”. The dissertation author was the primary author of this material.

Chapter 5, in part, has been published at 2020 INTERSPEECH and appeared as Huili Chen, Bita Darvish, Farinaz Koushanfar, “SpecMark: A Spectral Watermarking Framework for IP Protection of Speech Recognition Systems”. The dissertation author was the primary author of this material.

Chapter 6, in part, has been published at the 2021 Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV) and appeared as Huili Chen, Cheng Fu, Jishen Zhao, Farinaz Koushanfar, “ProFlip: Targeted Trojan Attack with Progressive Bit Flips”. The dissertation author was the primary author of this material.

Chapter 7, in part, has been published at the Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI), 2019 and appeared as Huili Chen, Cheng Fu, Jishen Zhao, Farinaz Koushanfar, “DeepInspect: A Black-box Trojan Detection and Mitigation Framework for Deep Neural Networks”. The dissertation author was the primary author of this material.

Chapter 8, in part, has been published at the 2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD) and appeared as Huili Chen, Cheng Fu, Jishen Zhao, Farinaz Koushanfar, “GenUnlock: An Automated Genetic Algorithm Framework for Unlocking Logic Encryption”. The dissertation author was the primary author of this material.

Chapter 9, in part, has been published at 2022 ACM Transactions on Embedded Computing Systems (TECS) and appeared as Huili Chen, Xinqiao Zhang, Ke Huang, Farinaz Koushanfar, “AdaTest: Reinforcement Learning and Adaptive Sampling for On-chip Hardware Trojan Detection”. The dissertation author was the primary author of this material.

This dissertation was supported, in parts, by the Office of Naval Research (ONR) N00014-17-1-2500, MURI (FA9550-14-1-0351), National Science Foundation (NSF) Trust-Hub (CNS-1649423, CNS-2016737), NSF-TILOS (CCF-2112665), Army Research Office (ARO) W911NF1910317, and Semiconductor Research Corporation (SRC)-Auto (2019-AU-2899).

VITA

2016	Bachelor of Science in School of Optical and Electronic Information, Huazhong University of Science & Technology
2016–2018	Master of Science in Electrical Engineering (Intelligent Systems, Robotics & Control), University of California San Diego
2016–2022	Graduate Research Assistant, University of California San Diego
2022	Doctor of Philosophy in Electrical Engineering (Intelligent Systems, Robotics & Control), University of California San Diego

PUBLICATIONS

H. Chen, Jie Ding, Eric Tramel, Shuang Wu, Anit Kumar Sahu, Salman Avestimehr, and Tao Zhang, "Self-Aware Personalized Federated Learning", Conference on Neural Information Processing Systems (NeurIPS), 2022.

H. Chen, Jie Ding, Eric Tramel, Shuang Wu, Anit Kumar Sahu, Salman Avestimehr, and Tao Zhang, "ActPerFL: Active Personalized Federated Learning", Federated Learning for Natural Language Processing (FL4NLP), 2022.

H. Chen, Xinqiao Zhang, Ke Huang, and Farinaz Koushanfar, "AdaTest: Reinforcement Learning and Adaptive Sampling for On-chip Hardware Trojan Detection", ACM Transactions on Embedded Computing Systems (TECS), 2022.

X. Zhang, **Huili Chen**, Ke Huang, and Farinaz Koushanfar, "An Adaptive Black-box Backdoor Detection Method for Deep Neural Networks", arXiv preprint, 2022.

H. Chen, Cheng Fu, Jishen Zhao, and Farinaz Koushanfar, "GALU: A Genetic Algorithm Framework for Logic Unlocking", Digital Threats: Research and Practice, 2022.

T. Nguyen, Phillip Rieger, **Huili Chen**, Hossein Yalame, Helen Möllering, Hossein Fereidooni, Samuel Marchal, Markus Miettinen, Azalia Mirhoseini, Shaza Zeitouni, Farinaz Koushanfar, Ahmad-Reza Sadeghi , and Thomas Schneider, "FLAME: Taming Backdoors in Federated Learning", USENIX Security, 2022.

O. Lutz, **Huili Chen**, Hossein Fereidooni, Christoph Sendner, Alexandra Dmitrienko, Ahmad Reza Sadeghi, and Farinaz Koushanfar, "ESCORT: ethereum smart contracts vulnerability detection using deep neural network and transfer learning", arXiv preprint, 2021.

H. Chen, Cheng Fu, Jishen Zhao, and Farinaz Koushanfar, "Proflip: Targeted trojan attack with progressive bit flips", IEEE/CVF International Conference on Computer Vision (ICCV), 2021.

H. Chen, Siam Umar Hussain, Fabian Boemer, Emmanuel Stapf, Ahmad Reza Sadeghi, Farinaz Koushanfar, and Rosario Cammarota, "Developing privacy-preserving AI systems: The lessons learned", ACM/IEEE Design Automation Conference (DAC), 2020.

M. Javaheripi, **Huili Chen**, and Farinaz Koushanfar, "Unified architectural support for secure and robust deep learning", ACM/IEEE Design Automation Conference (DAC), 2020.

H. Chen, Rosario Cammarota, Felipe Valencia, Francesco Regazzoni, and Farinaz Koushanfar. "AHEC: End-to-end compiler framework for privacy-preserving machine learning acceleration", ACM/IEEE Design Automation Conference (DAC), 2020.

C. Fu, **Huili Chen**, Zhenheng Yang, Farinaz Koushanfar, Yuandong Tian, and Jishen Zhao, "Enhancing model parallelism in neural architecture search for multidevice system", IEEE Micro, 2020.

H. Chen, Seetal Potluri, and Farinaz Koushanfar, "FlowTrojan: Insertion and detection of hardware Trojans on flow-based microfluidic biochips", IEEE International New Circuits and Systems Conference (NEWCAS), 2020.

H. Chen, Seetal Potluri, and Farinaz Koushanfar, "Security of microfluidic biochip: Practical attacks and countermeasures", ACM Transactions on Design Automation of Electronic Systems (TODAES), 2020

H. Chen, Bita Darvish Rouhani, and Farinaz Koushanfar, "SpecMark: A Spectral Watermarking Framework for IP Protection of Speech Recognition Systems", INTERSPEECH, 2020.

H. Chen, Rosario Cammarota, Felipe Valencia, and Francesco Regazzoni, "Plaidml-he: Acceleration of deep learning kernels to compute on encrypted data", IEEE International Conference on Computer Design (ICCD), 2019.

H. Chen, Cheng Fu, Jishen Zhao, and Farinaz Koushanfar, "GenUnlock: An automated genetic algorithm framework for unlocking logic encryption", IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2019.

H. Chen, Cheng Fu, Jishen Zhao, and Farinaz Koushanfar, "DeepInspect: A Black-box Trojan Detection and Mitigation Framework for Deep Neural Networks", International Joint Conference on Artificial Intelligence (IJCAI), 2019.

C. Fu, **Huili Chen**, Haolan Liu, Xinyun Chen, Yuandong Tian, Farinaz Koushanfar, and Jishen Zhao, "Coda: An end-to-end neural program decompiler", Neural Information Processing Systems (NeurIPS), 2019.

H. Chen, Cheng Fu, Bita Darvish Rouhani, Jishen Zhao, and Farinaz Koushanfar, "DeepAttest: an end-to-end attestation framework for deep neural networks", ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA), 2019.

H. Chen, Bita Darvish Rouhani, Cheng Fu, Jishen Zhao, and Farinaz Koushanfar, "DeepMarks: A secure fingerprinting framework for digital rights management of deep learning models", International Conference on Multimedia Retrieval (ICMR), 2019.

B. Rouhani, **Huili Chen**, and Farinaz Koushanfar, "DeepSigns: An end-to-end watermarking framework for ownership protection of deep neural networks", International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2019.

H. Chen, Bita Darvish Rouhani, and Farinaz Koushanfar, "BlackMarks: Blackbox multibit watermarking for deep neural networks”, arXiv preprint, 2019.

H. Chen,, Bita Darvish Rouhani, Xinwei Fan, Osman Cihan Kilinc, and Farinaz Koushanfar, “Performance comparison of contemporary dnn watermarking techniques”, arXiv preprint, 2018.

H. Chen, Seetal Potluri, and Farinaz Koushanfar, “BioChipWork: Reverse engineering of microfluidic biochips”, IEEE International Conference on Computer Design (ICCD), 2017.

ABSTRACT OF THE DISSERTATION

Towards Holistic Secure and Trustworthy Deep Learning

by

Huili Chen

Doctor of Philosophy in Electrical Engineering (Intelligent Systems, Robotics & Control)

University of California San Diego, 2022

Professor Farinaz Koushanfar, Chair

Machine Learning (ML) models, in particular Deep Neural Networks (DNNs), have been evolving exceedingly fast in the past few decades although the idea of DNNs was proposed in the nineteenth century. The success of contemporary ML models can be attributed to two key factors: (i) Data of various modalities is becoming more abundant for designers, which makes data-driven approaches such as DNNs more applicable in real-world settings; (ii) The computing power of emerging hardware platforms (e.g., GPUs, TPUs) is becoming stronger due to the architecture advance. The increasing computation capability makes the training of large-scale DNNs practical for complex data applications. While ML has enabled a paradigm shift in various fields such as autonomous driving, natural language processing, and biomedical

diagnosis, training high-performance ML models can be both time and resource-consuming. As such, commercial ML models (which typically contain a tremendous amount of parameters to learn complex tasks) are trained by large tech companies and then distributed to the end users or deployed on the cloud for Machine Learning as a Service (MLaaS).

This supply chain of ML models raises concerns for both model designers and end users. From the model developer's perspective, he/she wants to ensure ownership proof of the trained model in order to prevent copyright infringement and preserve the commercial advantage. For the end user, he/she needs to verify the obtained ML model is not maliciously altered before deploying the model. This dissertation introduces holistic algorithm-level and hardware-level solutions to resolving the Intellectual Property (IP) protection and security assessment challenges of ML models, thus facilitating safe and reliable ML deployment.

The key contributions of this dissertation are as follows:

- Devising an end-to-end collusion-secure DNN fingerprinting framework named *Deep-Marks* that enables the model owner to prove model authorship and identify unique users in the context of Deep Learning (DL). I design a fingerprint embedding technique that combines anti-collusion codes and weight regularization to ensure the fingerprint is encoded in the marked DL model in a robust manner while preserving the main task accuracy.

- Designing a hardware-level IP protection and usage control technique for DL applications using on-device DNN attestation. The proposed framework *DeepAttest* leverages device-specific fingerprints to 'mark' authentic DNNs and verifies the legitimacy of the deployed DNN with the support of the Trusted Execution Environment (TEE). The algorithm and hardware architecture of *DeepAttest* are co-optimized to ensure the process of on-device DNN attestation is lightweight and secure.

- Developing a spectral-domain DNN watermarking framework named *SpecMark* that removes the requirement of model re-training for watermark embedding and is robust against transfer learning. I adapt the idea of spread spectrum watermarking in the conventional multi-media domain to protect the IP of model designers using spectral watermarking. The effectiveness and robustness of *SpecMark* are corroborated on various automatic speech recognition datasets.

- Demonstrating a targeted Trojan attack against DNNs named *ProFlip* that exploits bit flipping techniques (particularly Row Hammer attacks) for Trojan insertion. Compared to previous Neural Trojan attacks that require poisoned training to backdoor the model, ProFlip can embed the Trojan after model deployment. To this end, I develop a new layer-wise sensitivity analysis technique to pinpoint the vulnerable layer for attack and a novel critical bit search algorithm that identifies the most susceptible weights bits.

- Designing a black-box Trojan detection and mitigation framework called *DeepInspect* that can assess a pre-trained DL model and determines if it has been backdoored. DeepInspect defense scheme identifies the footmark of Trojan insertion by learning the probability distribution of potential triggers with a conditional generative model. DeepInspect further leverages the trained generator to patch the model for higher Trojan robustness.

- Proposing a genetic algorithm-based logic unlocking scheme named *GenUnlock* that outperforms prior satisfiability (SAT)-based counterpart with better runtime efficiency. GenUnlock performs fast and effective key searching by algorithm/hardware co-design and an ensemble-based method. Empirical results show that GenUnlock reduces the attack runtime by an average of $4.68\times$ compared to SAT-based attacks.

- Introducing a new logic testing-based Hardware Trojan detection framework named *AdaTest* that combines Reinforcement Learning (RL) and adaptive sampling. AdaTest achieves dynamic and progressive test pattern generation by defining a domain-specific reward function for circuits that characterizes both the static and dynamic properties of the circuit status. Experimental results show that AdaTest obtains a higher Trojan coverage with a shorter test pattern generation time compared to prior arts.

Chapter 1

Introduction

The evolution of computing architectures and the rapid growth of sensor deployments have led to an unprecedented amount of data. This significant improvement in computing capability and the increasing volume of data enable practical deployment of data-demanding Deep Learning (DL) systems, particularly Deep Neural Networks (DNNs), in various real-world applications. There are two critical concerns associated with the wide usage of DL models. From the perspective of the model designer, he/she needs to protect the copyright of the pre-trained DNNs for ensuring commercial advantage in the market. Protecting the Intellectual Property (IP) of high-performance DL models is important since the training process requires tremendous computation and proprietary labeled training data. From the viewpoint of the end user, he/she obtains the pre-trained model from the third-party provider without additional information on the training process. Such opaque access to the DNN renders backdoor insertion possible for malicious model providers and might jeopardize the security of deployed DNNs.

This dissertation addresses the aforementioned two concerns for safe and reliable deep learning. In particular, I propose holistic, end-to-end solutions to enabling DL model tracing/attestation as well as security inspection with algorithm/software/hardware co-design. Furthermore, I adapt Machine Learning (ML) techniques to solve long-standing hardware security problems with better effectiveness and efficiency. In this section, I will review the challenges and my proposed solutions to secure and trustworthy deep learning.

1.1 Intellectual Property Protection of Deep Learning Models

The complexity and size of deep learning models are increasing rapidly to accommodate the performance requirement of contemporary data applications such as natural language processing [YHPC18], protein structure prediction [SEJ$^+$20], and autonomous driving [GTCM20]. For instance, the GPT3 model from OpenAI has 175 billion parameters and the Switch Transformer model from Google consists of 1.6 trillion parameters [Mar21]. As an example of the prohibitive training overhead, the reinforcement learning-based device placement tool [MPL$^+$17] takes about 300 hours of GPU training. Such a high resource consumption makes it impractical for end users to train their own DL model and also makes pre-trained DNNs valuable properties of the model designer. Therefore, the model owner shall protect the IP of his/her profitable DNNs so that illegal usages/copyright infringement of the model can be identified and traced.

In the following, I first discuss a new DL fingerprinting framework that enables traceable DNNs and then present a hardware-bounded variant for on-device DNN attestation. In the last subsection, I show a lightweight DL watermarking scheme using spread spectrum modulation.

1.1.1 Traceable Deep Neural Networks

The intellectual property concern of deep neural networks has received interest from both the research community and industrial practitioners. Previous works have devised ownership proof techniques for DNNs by embedding an owner-specific signature into the trained model. Particularly, Uchida *et al.* [UNSS17] make the first attempt to extend multi-media watermarking to deep learning models using weight regularization. Their paper [UNSS17] considers a scenario where the model owner has full access to the deployed DNN (i.e., white-box setting). Later works develop 'backdoor-based' watermarking approaches for DL models [ABC$^+$18, GP18, LMPT20] in the setting where the DNN is employed in a remote service (i.e., black-box scenario). While the above DNN watermarking techniques provide ownership proof for pre-trained DNNs, they cannot attribute the illegal activity of DL model abuse to the true liability (which is the person that violates the IP of the model owner).

I introduce DeepMarks [CRF$^+$19], the first *collusion-resilient fingerprinting framework* that enables unique user identification and digital right management for deep learning models. In particular, DeepMarks allows the model owner to encode a user-specific fingerprint in the corresponding distributed DNN. The unique fingerprint in each model allows the model designer to *trace* the usage of individual copies of the same model. In addition, DeepMarks outperforms existing DL watermarking techniques in terms of robustness against fingerprint collusion attacks. I empirically demonstrate that multiple users that have different copies of the same DL model can collaborate and perform analytical analysis to remove the owner's signature from their marked models [CRF$^+$19]. On the contrary, DeepMarks deploys anti-collusion codes when designing the fingerprints of users and can detect the 'traitors' who participate in the fingerprint collusion attack given the manipulated DNN.

With the increasing popularity of large-scale deep learning systems, for example, Federated Learning (FL), it is common that multiple DNNs are distributed to a tremendous number of edge users, making model usage tracing more challenging for DL model owners. With my proposed DeepMarks framework, the model owner can keep track of the usage of individual DNNs as well as detect malicious users who participate in the fingerprint removal attack. DeepMarks sheds light on scalable IP protection and reliable tracing of deep learning models, thus building the foundation for traceable large-scale DL deployment.

1.1.2 Usage Control of Deep Learning Hardware

While researchers have tried to adapt digital watermarking and fingerprinting techniques to deep learning models [UNSS17, ABC$^+$18, CRF$^+$19, DRCK19], such protection of the intellectual property is only limited to the software-level (i.e., only concerning the DNN program). Nowadays, an increasing amount of intelligent devices with accompanying DL models are deployed in various real-world applications. It is worth noting that developing deep learning accelerators [WGY$^+$16, SQLC17] also requires extensive experts' efforts and resource allocation. Therefore, the IP of intelligent devices needs to be protected as well, suggesting that software-level DL watermarking techniques are not adequate for this purpose. Particularly, existing DNN watermarking methods [UNSS17, ABC$^+$18, DRCK19] do not consider the overhead or the security of signature extraction when implemented on the hardware.

I design DeepAttest [CFR$^+$19], the first on-device DNN attestation framework that assures the legitimacy of the deployed model for a given DL hardware with the support of the Trusted Execution Environment (TEE). As opposed to previous DNN watermarking techniques, DeepAttest aims to provide *hardware-bounded* IP protection and usage control for the intelligent devices by co-designing the algorithm and architecture for on-device signature extraction. To control the activation of DNN attestation, I design a hybrid trigger that consists of a secure timer in the TEE and a dynamic memory monitoring signal. This trigger scheme allows DeepAttest to detect both static and dynamic data tampering attacks. Empirical results show that DeepAttest achieves high detection rates of undesired model deployment and incurs negligible attestation overhead in terms of runtime and energy.

State-of-the-art accelerators for deep learning applications require an enormous amount of development efforts, which makes hardware-linked IP protection of these intelligent devices indispensable. Leveraging DeepAttest, the hardware provider can restrict the usage of DL devices and detect/prevent undesired usage or abuse. The algorithm/hardware co-optimization principle enables lightweight and secure on-device attestation, making DeepAttest applicable to resource-constrained devices in the real world scenario.

1.1.3 Lightweight and Robust Spectral Watermarking

The motivation and the importance of intellectual property protection of DNNs have been discussed in the earlier sections. Recall that the white-box DL watermarking techniques [DRCK19, UNSS17, WK21] embed the owner's signature in the weight/activation distribution via regularization, while black-box DL watermarking methods [ABC$^+$18, GP18, CRK19] embed the signature in the output behavior of the model when given specific key inputs. Both types of DNN watermarking techniques require model training/fine-tuning to encode the signature information in the protected model, thus incurring non-trivial overhead when the pertinent DL model is complicated.

I introduce SpecMark [CRK20], a novel *spectral-domain* DNN watermarking framework that does not require model training for signature embedding, thus is very lightweight and suitable for large DNNs. Particularly, I adapt conventional spread-spectrum watermarking techniques in the multi-media domain to the deep learning models. This new spectral DL watermarking scheme automatically identifies the suitable frequency bins of the weight parameters that can carry the signature information without affecting the normal task accuracy. More specifically, the owner's signature (typically a sequence of binary bits) is encoded in the significant spectrum regions of the model weights by *additive modulation*. Given the unmarked model weights as the reference, the model owner can extract the embedded signature from the weight spectrum of the marked model for authorship proof. Empirical results show that SpecMark is not only lightweight, but also robust against parameter tuning and transfer learning.

Nowadays, the size and the complexity of DNNs are increasing rapidly to satisfy the demands of emerging applications. This trend implies that existing DL watermarking techniques [ABC$^+$18, DRCK19, CRK19, WK21] that require model training might incur non-negligible overhead for watermark embedding. In addition, prior works have shown that strategical fine-tuning can remove the signature in the marked model [CWD$^+$19, GZQ$^+$20], thus defeating current DL watermarking techniques. SpecMark demonstrates superior *efficiency* and *robustness* against these model transformations since we embed the signature in the high-magnitude spectrum region of the weight parameters instead of modifying the model weights directly. With the assistance of SpecMark, the model owners can protect the IP of their valuable DNNs without expensive re-training and prove the ownership even if the marked model undergoes pruning, fine-tuning, or transfer learning.

1.2 Security Assessment of Deep Learning Models

Deep learning models are widely employed in diverse applications such as biomedical diagnosis, financial analysis, and autonomous driving due to their autonomy and unprecedented performance. From the perspective of the end users, they obtain and deploy pre-trained DNNs from the third-party model providers while the security of these models is unknown. There has been a line of work focusing on the vulnerability of DNNs to various attacks, including adversarial samples [MMS$^+$17, AM18], Neural Trojan attacks [DS19, WPB$^+$21], and data poisoning attacks [CLL$^+$17, SESL18]. In the following of this section, I first discuss a new test-time threat against DNNs that exploits the vulnerability of DRAMs to disturbance errors. Then, I introduce a novel black-box Neural Trojan detection framework that can assess the safety of a pre-trained DNN against potential Trojan attacks.

1.2.1 Bit Flip Attacks against Deep Neural Networks

DNNs typically possess a large number of parameters to learn complex data applications. For instance, as an exemplar of language models, BERT [TDP19] has 110 million parameters and can handle question-answering tasks. Such a large capacity of DNNs, in turn, increases the attack surface for *parameter manipulation attacks* [HRL$^+$20, EBGLOUM22, HFK$^+$19]. This type of attack is feasible since the weights of DNNs are typically stored in the memory modules such as DRAMs. However, the data stored in DRAMs are susceptible to disturbance errors (also known as '*Rowhammer attacks*') caused by frequent row activations [KDK$^+$14, SD15, Mut17]. Hong *et al.* [HFK$^+$19] show that an adversary can launch untargeted attacks and divert the pre-trained DL model by modifying a single bit of the floating-point DNN. Targeted Bit Trojan (TBT) [RHF20] demonstrates that the attacker can embed the Trojan into Quantized Neural Networks (QNNs) by gradient-guided vulnerable neuron identification and partial finetuning-based bit flipping. However, TBT always exploits the last layer of the model for attack (which is not necessarily optimal) and its success rate is sensitive to the number of modified neurons.

I propose ProFlip [CFZK21], the first progressive *Bit Flip Attack* (BFA) framework that achieves targeted Trojan insertion without poisoned model re-training. ProFlip consists of three key steps: salient neuron identification, trigger generation, and progressive critical bits search. Particularly, I adapt the adversarial saliency map [WX18] to find neurons associated with the attack target class in the last layer. Furthermore, I propose a new parameter sensitivity metric for QNNs that allows the adversary to select the most vulnerable layer for the attack. To search for a few weight bits that are critical for the targeted Trojan attack, ProFlip computes the sensitivity metric for each element in the attack parameter and estimates the corresponding optimal value that maximizes the Attack Success Rate (ASR). Empirical results show that ProFlip achieves the same level or higher ASRs ($\geq 94\%$) compared to TBT [RHF20], while reducing the number of bit flips by an average of $32\times$.

1.2.2 Neural Trojan Detection for Safe Model Deployment

The security concerns of end users when deploying pre-trained DL models have been discussed in the previous section. Considering the supply chain of DNNs where model training is performed by third-party companies with sufficient computing power, Neural Trojan attacks [DS19, LXS17, LMA$^+$18] are of particular concern since the customers do not have any knowledge about model training. A typical Neural Trojan attack consists of two components: Trojan trigger and Trojan payload. The trigger is a pre-defined pattern in the input space. The Trojan payload is the malicious behavior that the adversary desires to achieve. Prior works have shown that the attacker can design a stealthy Trojan [LMA$^+$18, CMN$^+$20, CLMZ21] against the victim model such that the Trojaned model performs normally on clean inputs, while it malfunctions (i.e. payload activated) when the trigger is present.

I introduce DeepInspect [CFZK19a], the first black-box Trojan detection and mitigation framework that can assess the security of the pre-trained model against Neural Trojan attacks. To this end, I leverage a conditional *Generative Adversarial Network* (GAN) to emulate the potential Trojan attack for each output class and characterize the decision boundary of the given model. Particularly, the conditional GAN learns to generate the trigger that can transform the data prediction from the source class to the target class. The footprint of the GAN's output (i.e., the trigger) is used as the test statistics of *hypothesis testing* for anomaly detection. If an outlier is detected, then the queried model is determined to be Trojaned. Experimental results show that DeepInspect achieves higher Trojan detection rates compared to the prior art Neural Cleanse [WYS$^+$19] across various Trojan configurations.

With the capability of DeepInspect, the end users can evaluate the safety of the pre-trained DNN obtained from the third-party model provider before deploying it in the field. This *model-level security assessment* is particularly important since it can avoid unnecessary safety/privacy breaches when employing the Trojaned model for critical tasks. DeepInspect is scalable to applications involving high-dimensional data. To handle inputs with a large dimensionality, DeepInspect proposes to incorporate an auto-encoder such that the conditional GAN learns to recover the potential trigger in the embedding space (which has a much lower dimension). Furthermore, DeepInspect enables the end users to enhance the robustness of the pre-trained DNN and mitigate the threat of Trojan attacks by provisioning additional 'perturbed' data with correct labels using the converged conditional GAN. With our *model patching* scheme, the Trojan activation rate can be effectively reduced to below 10%.

1.3 Deep Learning for Hardware Security

The data-driven nature of deep learning and its capability of automated representation learning make it suitable for solving diverse problems. In this section, I will discuss my interdisciplinary study between deep learning and hardware security. In particular, this section shows how I adapt DL techniques to solve two long-standing hardware security problems: logic locking and hardware Trojan detection.

1.3.1 Attacking Logic Encryption

Logic encryption (also called logic locking) [YRSK15, YSN$^+$17, HYR21] is an IP protection technique for digital circuits that obfuscates the functionality of the circuit by inserting additional key gates. The protected circuit only gives correct outputs when the ground-truth key inputs are applied to the key gates. Therefore, logic locking allows the circuit designer to protect his/her IP even if the netlist of the locked circuit is distributed to the foundry for fabrication. However, prior works have shown that logic locking is susceptible to various attacks such as functional analysis [CCB19, SS20], removal attacks [YMSR17b, YMSR17a], and satisfiability (SAT) attacks [SRM15, AKHS19]. While SAT-based attacks can eliminate the equivalent class of incorrect keys at each iteration (thus shrinking the key searching space), the worst-case complexity has an exponential relation with the number of primary inputs.

I design GenUnlock [CFZK19b], the first Genetic Algorithm (GA)-based *logic unlocking attack* framework that can find an effective decryption key for the given circuit. Instead of trying to find the exact decryption key, GenUnlock aims to search for an 'approximate' unlocking key that enables the locked circuit to produce the correct outputs with a high probability. Given oracle access to an active (unlocked) circuit, GenUnlock formulates key searching as a *combinatorial optimization* problem where the goal is to maximize the matching ratio of output signals between the locked circuit and the corresponding unlocked one. I leverage the evolutionary nature of GAs and explore the key searching space efficiently by designing a domain-specific fitness score for logic unlocking. With algorithm/hardware co-design, empirical results show that GenUnlock can achieve up to three orders of magnitude of runtime speedup and energy reduction compared to the SAT-based attacks [SRM15].

1.3.2 Detecting Hardware Trojans

For digital circuits, Hardware Trojan (HT) [CNB09, TK10] is a type of attack that maliciously modify the digital circuits to insert the desired payload. In addition, the adversary designs a trigger signal to control the activation of the hardware Trojan. The Trojan trigger is stimulated only in very rare conditions for ensuring attack stealthiness. The attacker can exploit HTs for different purposes including producing the wrong outputs (i.e., malfunctioning) or stealing private information. Previous works have explored side-channel analysis [NDC$^+$12, LHM14] and logic testing [CWP$^+$09, NFH18] primitives for hardware Trojan detection. However, side-channel analysis-based methods yield high false positive rates when detecting small-scale hardware Trojans, while logic testing-based approaches typically require a large number of testing patterns (thus long detection time) to reach a sufficiently high Trojan coverage.

I propose AdaTest [CZHK22], the first reinforcement learning-based Automated Testing Pattern Generation (ATPG) framework for hardware Trojan detection with the assistance of adaptive sampling. AdaTest takes a progressive approach to generate the test inputs given the netlist of the queried circuit. To achieve a high Trojan activation rate, I design a customized reward function for HT detection by characterizing the circuit status using transition probabilities, testability measures, and graph-level diversity. In each round, AdaTest generates multiple tentative test inputs and selects the ones with high rewards to update the final test set. Empirical results show that AdaTest attains more than 10% Trojan detection rate improvement compared to prior arts [CWP$^+$09, NFH18] while reducing the test set size by an order of magnitude.

Besides hardware Trojan detection, AdaTest allows the end users (or validation technicians) to verify the security of a given circuit by checking its behaviors against the expected ones. This capability is useful for other tasks such as built-in-self-test [McC85, AKS93]. Furthermore, AdaTest is very lightweight since: (i) A compact set of test inputs are generated in a sequential manner to attain high Trojan coverage rates; (ii) It leverages software/hardware co-design to accelerate the bottleneck of AdaTest (which is the process of obtaining the circuit's response on the given inputs and computing the reward).

1.4 Broader Impact

The broad goal of this dissertation is to provide a holistic solution to secure and reliable deep learning for both model developers and end users. More specifically, I have designed two model-level IP protection frameworks, DeepMarks [CRF$^+$19] and SpecMark [CRK20], that enable traceable DNN usage (i.e., user identification) and lightweight, training-free DNN watermarking. Besides IP protection for DL models, I proposed ProFlip [CFZK21] and revealed the vulnerability of DNNs to bit flipping-based Trojan attacks. The DeepInspect [CFZK19a] framework I proposed allows end users to assess the security of pre-trained DNNs against Neural Trojan attacks. Last but not the least, I have adapted deep learning to solve existing hardware security problems. My proposed framework GenUnlock [CFZK19b] demonstrates an effective and efficient attack against logic locking by leveraging genetic algorithms. AdaTest [CZHK22] employs reinforcement learning and achieves a high Trojan coverage rate with a compact test set for hardware Trojan detection.

1.5 Acknowledgements

Chapter 1, in part, has been published at has been published at (i) the Proceedings of the 2019 International Conference on Multimedia Retrieval (ICMR) and appeared as: Huili Chen, Bita Darvish Rouhani, Cheng Fu, Jishen Zhao, Farinaz Koushanfar, “DeepMarks: A Secure Fingerprinting Framework for Digital Rights Management of Deep Learning Models”, and (ii) 2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA) and appeared as Huili Chen, Cheng Fu, Bita Darvish Rouhani, Jishen Zhao, Farinaz Koushanfar, “DeepAttest: An End-to-End Attestation Framework for Deep Neural Networks”, and (iii) 2020 INTERSPEECH and appeared as Huili Chen, Bita Darvish, Farinaz Koushanfar, “SpecMark: A Spectral Watermarking Framework for IP Protection of Speech Recognition Systems”, and (iv) the Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV), 2021 and appeared as Huili Chen, Cheng Fu, Jishen Zhao, Farinaz Koushanfar,

"ProFlip: Targeted Trojan Attack with Progressive Bit Flips", and (v) the Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI), 2019 and appeared as Huili Chen, Cheng Fu, Jishen Zhao, Farinaz Koushanfar, "DeepInspect: A Black-box Trojan Detection and Mitigation Framework for Deep Neural Networks", and (vi) 2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD) and appeared as Huili Chen, Cheng Fu, Jishen Zhao, Farinaz Koushanfar, "GenUnlock: An Automated Genetic Algorithm Framework for Unlocking Logic Encryption", and (vii) 2022 ACM Transactions on Embedded Computing Systems (TECS) and appeared as Huili Chen, Xinqiao Zhang, Ke Huang, Farinaz Koushanfar, "AdaTest: Reinforcement Learning and Adaptive Sampling for On-chip Hardware Trojan Detection". The dissertation author was the primary author of these materials.

Chapter 2

Background

In this chapter, I first introduce the background of traditional digital watermarking and fingerprinting techniques in the multi-media domain (Section 2.1). Then, I discuss the mechanism of existing software attestation methods and the background of machine learning in Section 2.2 and Section 2.3, respectively. Last but not the least, I introduce three exemplar problems of hardware security in Section 2.4.

2.1 Multi-media Watermarking and Fingerprinting

A digital watermark (WM) is an invisible identifier that is embedded as an integral part of the host design and has been widely adopted in the multi-media domain for IP protection [Lu04, CMB$^+$07]. The host of the identifier can be images, video contents, and functional artifacts such as digital integrated circuits [FK04, HK99, QP07].

Conventional digital watermarking techniques have two phases: WM embedding and WM extraction. Figure 2.1 visualizes the workflow of a typical *constraint-based watermarking* system. The original problem (e.g., image classification) is used as the ‘cover constraints’ to hide the owner’s WM signature. To embed the watermark, the IP designer creates the stego-key and a set of additional constraints that do not conflict with cover constraints. Combining these two constraints yields the stego-problem that is solved to produce the stego-solution. It is worth noting that the stego-solution simultaneously satisfies both the original constraints

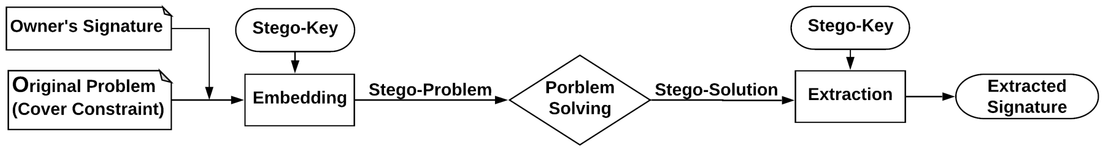


Figure 2.1. Constraint-based watermarking system.

and the watermarking-specific constraints. Therefore, the multi-media designer can extract the watermark from the stego-solution and claim the authorship. An effective watermarking scheme needs to meet a set of criteria including imperceptibility, robustness, verifiability, capacity, and low overhead [SC13, BP13].

Digital watermarking belongs to the wide area of *information hiding* that communicates information by embedding and retrieving it in digital data [PAK99, MO03]. There are various motivations for information hiding, such as protecting the digital object from malicious usage (i.e. digital watermarks), covert communication, or gaining side benefits for free [GGS14, Mem02]. Information hiding has four general properties:

- **Fidelity** describes the level of perceptual degradation induced by information embedding. For instance, an image with the secret encoded shall not have a visually recognizable difference from the original clean image.
- **Robustness** characterizes the resiliency of the embedded information against possible distortion/manipulation of the information-carrying object (i.e., stego-solution). For example, the image owner shall be able to recover his/her signature from the marked image even if it undergoes image compression or filtering.
- **Payload** describes the amount of information that can be reliably embedded and extracted from the marked object. It is worth noting that the payload of an information hiding scheme shall be decided while complying with the fidelity and robustness constraint.
- **Security** of an information hiding scheme depends on the actual application. For instance,

the security of digital watermarking requires that unauthorized users cannot detect/remove the watermark from the marked host.

Information hiding techniques can be categorized based on different criteria. Based on the data modality of the host object, it can be classified into images, texts, audio, software programs, etc. According to the assumption of secret information extraction, it can be categorized into blind detection (which does not need the original cover object) and non-blind extraction (which requires the authentic host signal as the reference). Based on the resiliency requirement of the embedded information, we can categorize information hiding into robust, fragile, and semi-fragile variants.

One can model information hiding using a communication system. Figure 2.2 shows the schematic diagram of a standard communication system. The input message m first undergoes channel encoding with the given key. Then, an additive noise n in the communication channel is applied, resulting in a noisy signal y. In the last step, the channel decoder recovers the message m' from y. Analogously, Figure 2.3 shows how a digital watermarking scheme can be modeled from the perspective of a communication system. In this scenario, a cover object c (or so-called the host signal) is involved in watermark embedding, which produces the stego solution (cw). This stego-solution undergoes unknown manipulation/distortion, which is modeled as the noise signal n. The watermark decoder then recovers the message m' using both the noisy stego-solution and the original cover object [MK01, Pat14].

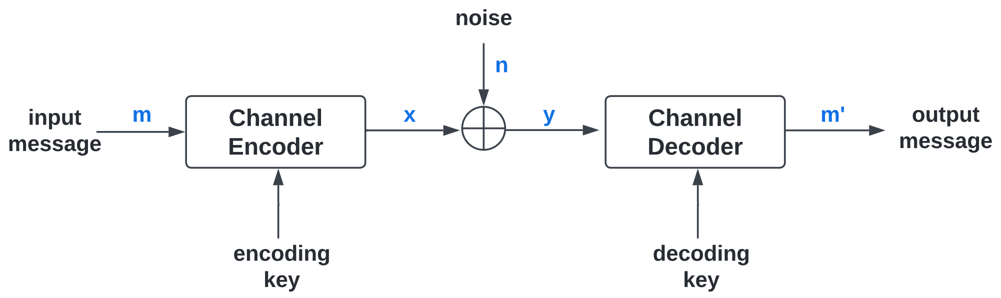


Figure 2.2. Demonstration of a standard communication system with key-based channel encoding [Pat14].

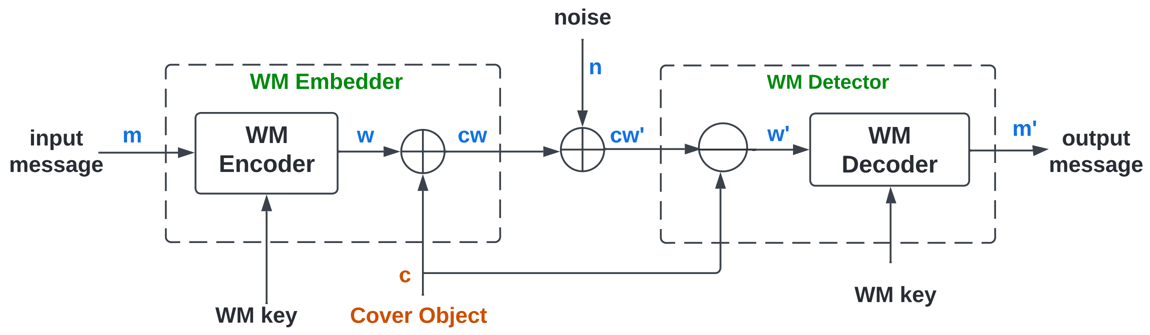


Figure 2.3. Modeling a digital watermarking scheme with non-blind detection using communication systems.

2.1.1 Watermarking in Transformed Domain

We discuss general concepts in conventional multi-media watermarking in the above section. Particularly, a digital watermarking scheme needs to be robust against perturbations while respecting the fidelity criteria (i.e., cannot degrade the 'quality' of the original cover object). Earlier works on digital watermarking focused on the *spatial domain* where the watermark information is embedded directly in the host signal via (scaled) addition [HK99, Lu04, CMB$^+$07]. These techniques typically use a pseudo-random signal with a small magnitude as the watermark. To avoid visible changes caused by watermark embedding, the magnitude of the watermark signal needs to be carefully designed.

Later works on multi-media watermarking have shown that watermark embedding can be performed in the *transformed domain* instead of the spatial one. This means that the watermark signal is encoded in the transformed host signal. For instance, the paper [CKLS96] proposes to perform digital watermarking in the largest components of the Discrete Cosine Transform (DCT) domain. Researchers have also investigated watermark embedding using Discrete Wavelet Transform (DWT) since it shares similarities with the theoretical model of the Human Visual System (HVS) [HN05, GM10, LZZ$^+$08]. Particularly, DWT provides a multi-resolution representation of an image by decomposing it into sub-bands of different frequencies/resolutions. To reduce visual degradation, the DWT-based watermarking technique [GM10] embeds the watermark in the high frequency sub-bands of the host signal since the majority energy of the cover object is distributed in the low frequency sub-band. Figure 2.4 shows the workflow of digital watermarking schemes in the transformed domain.

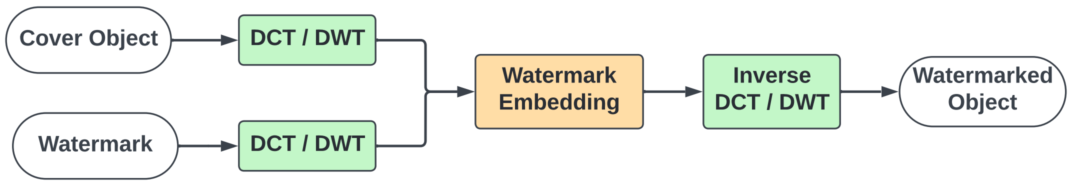


Figure 2.4. Demonstration of a DCT/DWT-based watermarking scheme.

To enhance the security of digital watermarking, CDMA-based *spread spectrum watermarking* is developed [CKLS96, CMB$^+$07, GM10]. Traditional spread spectrum communication transmits a narrow band signal over a much larger bandwidth so that the energy of the target signal is distributed across different frequencies, which makes it difficult to detect the presence of signal transmission. Inspired by the benefits of CDMA-based spread spectrum communication, prior works have proposed spread spectrum watermarking that considers the host signal (e.g., images) and the watermark as the communication channel and transmission signal, respectively [CKLS97, KM03, PFPG09]. Experimental results show that spread spectrum watermarking is robust and has strong anti-interference.

2.1.2 Multi-media Fingerprinting

While multi-media watermarking can address the IP ownership concern of the media owner, it is not able to distinguish multiple copies of the object that carry the same watermark. In the real-world setting, the IP owner also has the intention to identify and track each copy of the marked object. Another challenge in the multi-user environment (e.g., distributed systems) is that multiple users may collaborate to remove the watermark from the protected host object. Figure 2.5 shows an example of such collusion attacks against additive embedding-based digital watermarks. It is worth noting that a naive adaptation of the previous digital watermarking techniques cannot address the above challenge since watermarks have been shown to be susceptible to statistical

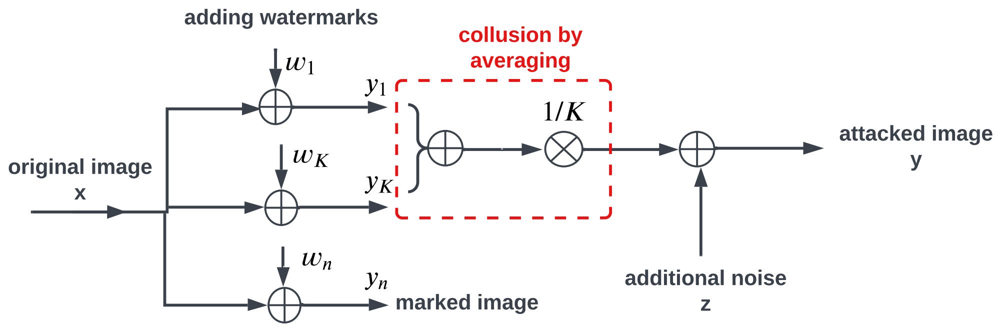


Figure 2.5. Demonstration of an averaging-based collusion attack against the digital watermarking system. [WTWL04].

attacks [KVH00, TTAH12, NR13]. Here, we discuss digital fingerprinting techniques that are able to provide user-level tracking and resistance against collaborative signature removal attacks.

The main difference between digital watermarking and fingerprinting is that the watermark remains the same for all copies of the IP while the fingerprint is unique for each copy. As such, digital fingerprinting address the ambiguity issue of the watermarking techniques and enables the owner to trace back IP misuse to the liable users, thus achieving Digital Right Management (DRM) [KK09, WTWL04, RET+17]. Figure 2.6 shows the usage of digital fingerprinting for user tracing and collusion-resistant IP protection. The top part of Figure 2.6 illustrates the workflow of multi-media fingerprinting with the assistance of the codebook. The middle row shows how two users can collaborate to construct a multi-media copy without any fingerprint in the multi-user collusion attack. The bottom row of Figure 2.6 shows how the digital fingerprinting scheme identifies the participants of collusion attacks based on the colluded multi-media copy.

Digital fingerprinting techniques can be categorized into orthogonal and code modulated fingerprinting based on the code construction methods. *Orthogonal (or independent) fingerprinting* deploys orthogonal signals as the fingerprints for individual users [WWZ+03, WWZ+05, KM09]. Prior works have suggested to use component-wise Gaussian distribution to generate orthogonal fingerprints [CKLS96, PZ98]. The resiliency of orthogonal fingerprints against collusion attacks are also theoretically studied in the papers [WWZ+03, WWZ+05, KM09]. Or-

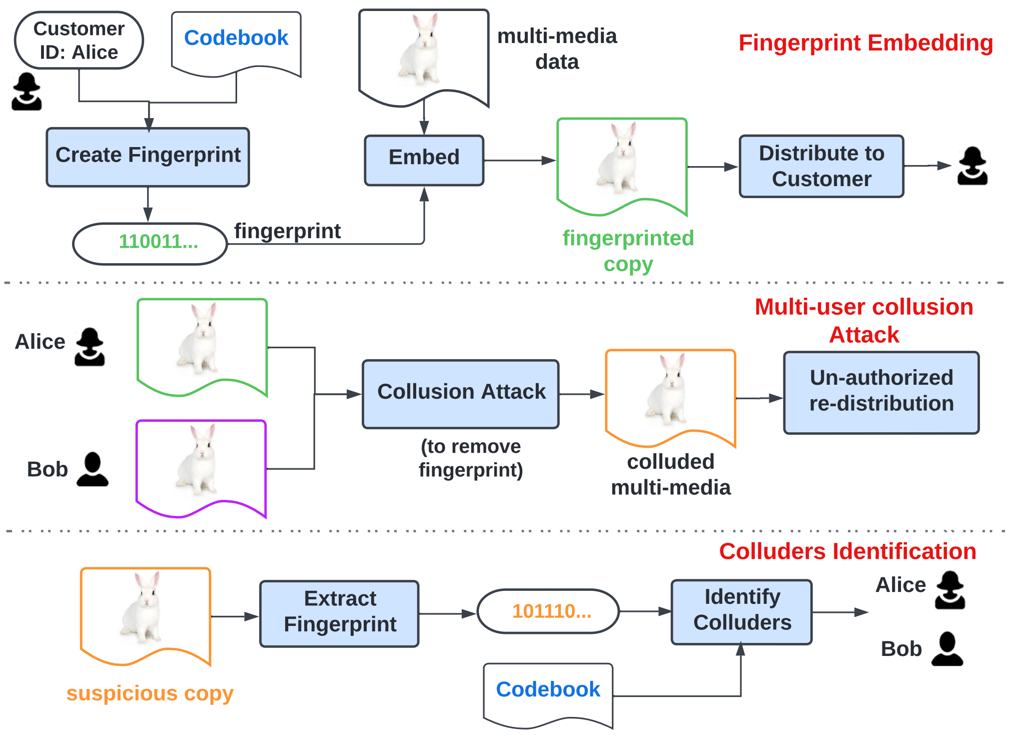


Figure 2.6. Using digital fingerprinting for digital right management of multi-media contents [WTWL04].

thogonal fingerprinting is attractive due to its simple implementation and suitability for systems with a small number of users. *Coded fingerprinting* constructs users' fingerprints using the linear combination of orthogonal basis signals. Code modulation has the following advantages: (i) It can support more users for a given fingerprint dimensionality; (ii) The deployment of Anti-Collusion Code (ACC) can improve the robustness of coded fingerprinting against fingerprint collusion attacks [TWL02, TWWL03, YLCZ10].

2.2 Software Attestation

Software attestation is a technique for proving the identify of a program. The objective of software-based attestation is to verify the integrity of code and data stored in the pertinent hardware [Bar06, ASSW13]. There are two parties involved in software-based attestation: prover and verifier. The prover wants to prove the identity of his code/data to the verifier by generating a ‘proof’ (or signature) based on the given program and its current state. The verifier is a trusted party and is responsible for checking whether the proof from the prover satisfies the expected condition [ASSW13, SL19]. Figure 2.7 shows the high-level workflow of a general remote attestation scheme that relies on challenge-response pairs for integrity verification. Software attestation is a sub-category of remote attestation where the main focus is the software program.

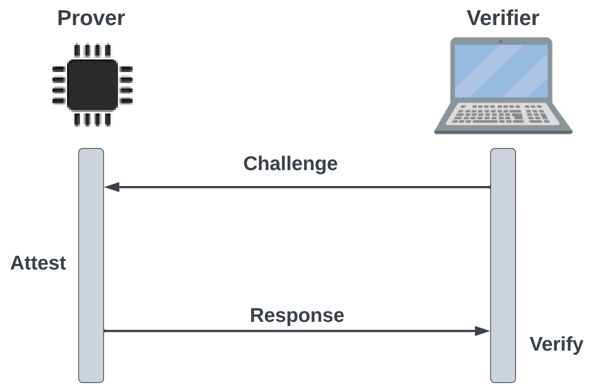


Figure 2.7. Demonstration of a remote attestation protocol.

Traditional software-based remote attestation is not concerned with the status of the underlying hardware that supports the program. Later works have extended attestation techniques to the Internet-of-Things (IoT) environment [SL19, ADD21] and embedded devices [SPVDK04, KKP$^+$14, NER$^+$19]. For instance, PUFatt [KKP$^+$14] binds software-based attestation to the inherent hardware properties by leveraging a processor-based Physically Unclonable Function (PUF). The integration of PUF enables secure time-constrained remote attestation for resource-constrained devices. VRASED [NER$^+$19] proposes a hybrid attestation scheme using software/hardware co-design. VRASED is ‘verifiable-by-design’ and obtains comparable security as hardware-based attestation. The paper [ADD21] provides a comprehensive survey of existing software-based attestation methods in the context of IoT and compares their advantages.

2.3 Machine Learning

In this section, we discuss background knowledge about deep learning, including deep neural networks, reinforcement learning, and genetic algorithms.

2.3.1 Neural Networks

A Neural Network (NN) consists of multiple intermediate layers residing between the input layer and the output layer. Each layer has various numbers of neurons. Consecutive layers are connected by wires and each wire is associated with a numeric value denoting its weight. Researchers have developed diverse layers types such as fully-connected layers, convolution layers, embedding layers, activation layers, and pooling layers [Wan03, AMAZ17, AJO$^+$18]. These layers can execute different kinds of computation on their inputs. As such, a neural network essentially performs a sequence of computations on the given input to obtain the final output. NNs are applicable in many problems such as regression, classification, and sequential decision-making [DOM02, MHZ$^+$08]. Depending on the requirement of the data labels, NNs can be trained in supervised, semi-supervised, and unsupervised settings [AMHH15, DZDW18].

Figure 2.8 shows an example of a neural network. The input to this NN is a vector of three elements $\mathbf{x} = (x_1, x_2, x_3)$ and the output is also a vector $\mathbf{z} = (z_1, z_2, z_3)$. The hidden layer has four neurons. We denote the weight connecting the i^{th} neuron in the input layer and the j^{th} neuron in the hidden layer as $w^1_{i,j}$. Then, the activation value of neurons in the hidden layer can be computed using the input and the weight values $y_j = \sigma(\sum_{i=1}^{3} x_i \cdot w^1_{i,j})$ where $j = \{1,2,3,4\}$ and $\sigma()$ is the non-linear activation function. Similarly, the output of the NN is computed as $z_k = \sigma(\sum_{j=1}^{4} y_j \cdot w^2_{j,k})$ where $k = \{1,2,3\}$. In the *forward pass*, the NN is given the input $\mathbf{x}$ and performs the computations as described above to obtain the output $\mathbf{z}$.

The 'quality' of a NN can be quantified by the loss and the goal of NN training is to minimize the *loss* of the neural network on the training set. For supervised learning, the training set $D = \{x_i, y_i\}_{i=1}^{n}$ (n is the number of data points) is a collection of (labeled) data samples

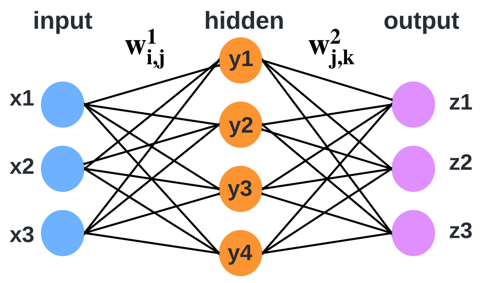


Figure 2.8. Illustration of an exemplar neural network.

where each sample is an input-output pair. Let us denote the function performed by the NN as f and the associated parameter set as θ. The loss $\mathscr{L}(f(\theta,x),y)$ describes how close the NN's output $f(\theta,x)$ is compared to the ground-truth/expected result y on the given input sample x. For instance, for a regression task, we can use Mean Square Error (MSE) as the loss function and compute the loss as:

$$\mathscr{L}(f(\theta,x),y) = \frac{1}{n}\sum_{i=1}^{n}(f(\theta,x_i)-y_i)^2. \tag{2.1}$$

For classification tasks, we can use Cross-Entropy (CE) loss as the loss function:

$$\mathscr{L}(f(\theta,x),y) = -\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{C} y_{i,j}\cdot log(p_{i,j}), \tag{2.2}$$

where C is the total number of classes, $p_{i,j}$ is the probability that the NN predicts class j for the i^{th} input data x_i, and $y_{i,j}$ is a binary value that indicates whether the predicted class j is correct for the input x_i. It is worth noting that the NN's output $f(\theta,x)$ in the loss evaluation is computed by forward pass as discussed above.

Training a neural network $f(\theta)$ is essentially the process of learning the model parameters θ by minimizing the empirical loss $\mathscr{L}(f(\theta,D))$. Recall that the data flow within the NN is analogous to a series of functions. Therefore, prior works have explored the chain rule and developed *Gradient Descent* as an iterative greedy algorithm to solve the optimization problem [Erb93, WM03]. Particularly, in the t^{th} iteration of NN training, the forward pass is performed and the loss of the current model parameters is computed on the training set.

To minimize the loss, the gradients of the loss with respect to each parameter in the NN are calculated using the chain rule. Then, the model parameter θ is updated along the opposite direction of the gradient with a proper step size α (which is called the 'learning rate') as shown in the equation below:

$$\theta_i^{t+1} = \theta_i^t - \alpha \cdot \frac{\partial \mathscr{L}}{\partial \theta_i^t}. \tag{2.3}$$

Here, the subscript i is the index of the parameter within the NN, and the superscript t is the iteration counter. This parameter update process (which is also called *backpropagation*) repeats until the loss of the model converges.

2.3.2 Generative Adversarial Network

Machine learning models can be categorized into two types, *generative* models and *discriminative* models, based on the probability distribution they capture. In terms of the functionality, generative models can produce new data samples, while discriminative models distinguish data instances from different classes [UB05, LBM06, Dev22]. Formally speaking, given a set of input data X and the corresponding labels Y, a generative model learns the joint distribution of the data $P(X,Y)$, or $P(X)$ if no labels are given. The discriminative model learns the conditional probability $P(Y|X)$. Therefore, the generative model describes the probability of the data (by estimating its distribution), while the discriminative model characterizes the probability of the data point coming from the specific class.

In the following of this section, I introduce a specific type of generative model called Generative Adversarial Network (GAN). A typical GAN consists of two components: a generator G and a discriminator D. The generator learns to approximate the distribution of real data and intends to produce feasible data samples. The discriminator learns to differentiate real inputs from the fake ones created by the generator [GPAM$^+$20, Dev22]. Note that both the generator and the discriminator are neural networks. These two components are trained simultaneously through backpropagation [CWD$^+$18, GPAM$^+$20]. Figure 2.9 shows the schematic view of a standard

GAN. The generator G takes the random noise vector $\mathbf{z}$ and optional auxiliary information (e.g., a specific class) as inputs, and returns synthetic data samples as outputs. The discriminator D tries to differentiate real inputs from 'fake' ones produced by the generator.

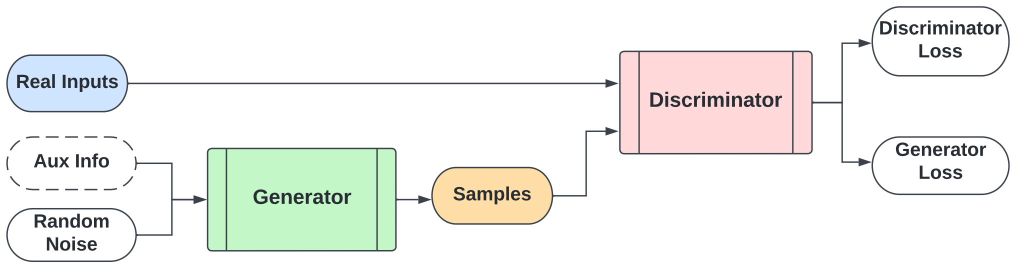


Figure 2.9. Structure of an exemplar generative adversarial network.

As shown in Figure 2.9, the generator maps the sampled random noise $\mathbf{z}$ to the input data $\mathbf{x}$ via computing $G(\mathbf{z};\theta_g)$ where θ_g is the parameter of the generator network. Meanwhile, the discriminator $D(\mathbf{x};\theta_d)$ outputs a single scalar (in the range of $[0,1]$) that quantifies the probability of its input coming from the real data instead of from the generator G. More specifically, the discriminator is trained to *maximize* the probability that it assigns correct labels to both the real training samples and the synthetic ones produced by G. The generator G is trained to *minimize* the value $log(1-D(G(\mathbf{z})))$. In other words, the generator tries to learn the mapping θ_g that makes the discriminator predicts 'real' on its output, i.e., forcing $D(G(\mathbf{z}))$ to be close to 1. Therefore, GAN training can be considered as a *minmax two-player game* where the generator and the discriminator are dynamically competing against each other [OSGP$^+$17, FG20, JWL$^+$20]. The minmax loss is computed as follows:

$$\mathbb{E}_{\mathbf{x}}[log(D(\mathbf{x})] + \mathbb{E}_{\mathbf{z}}[log(1-D(G(\mathbf{z})))]. \tag{2.4}$$

The optimization problem of GAN training can be formulated as:

$$\min_{G}\max_{D} V(D,\ G) = \mathbb{E}_{\mathbf{x}\sim p_{data}(\mathbf{x})}[log(D(\mathbf{x})] + \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}(\mathbf{z})}}[log(1-D(G(\mathbf{z})))]. \tag{2.5}$$

GAN training involves updating of two separate networks G and D as described in Equation (2.5). To solve this challenge, *alternative training* is proposed and works as follows: (i) The discriminator D is trained for certain epochs while fixing the parameters in G; (ii) The generator G is trained for certain epochs while freezing the discriminator D. These two steps are repeated until the GAN converges [GPAM+20, GSW+21]. Note that the generator is fixed during the training of the discriminator since D needs to learn how to identify the 'flaws' in the generator's output in contrast to the real data. This task is well defined only when the generator is kept constant. Similarly, the discriminator is fixed when training the generator since the goal of this stage is to train G such that it can successfully 'fool' the discriminator. Setting a dynamic discriminator will make a moving target for the generator to compete with, which will make the GAN training unstable [CWD+18, GPAM+20, Dev22].

2.3.3 Reinforcement Learning

Reinforcement learning [KLM96, WVO12, SB18] is a machine learning technique that is capable of solving complex problems in various domains. RL works *sequentially* in an environment by taking an action, evaluating its reward, and adjusting the following actions accordingly. In particular, an RL paradigm involves an *agent* that observes the environment and takes *actions* to maximize the *reward* determined by the problem of concern [SB18, MKS+13]. Figure 2.10 shows the interaction between the agent and the environment in the RL paradigm.

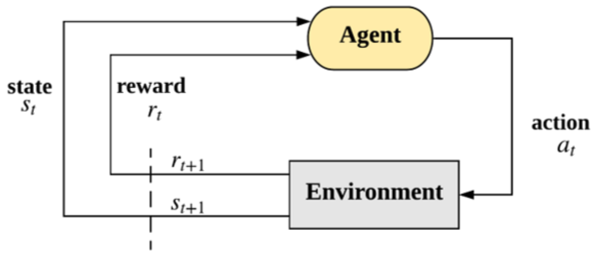


Figure 2.10. Illustration of the agent-environment interaction in reinforcement learning.

We introduce the key concepts in an RL system below:

- **State.** A state is a concrete and instantaneous situation in which the agent finds itself.

This can be an instant configuration, a particular place, and a moment that puts the agent in connection with other influential objects in the environment, such as the opponents or awards. It is noteworthy that a state needs to contain all information to ensure that the system satisfies the *Markov property* [PTLK18, Lei21].

▪ **Action Space.** The action space of an RL system is a set of possible moves that the agent can take to change to a new state. For example, in a video game, an action can be running left/right, or jumping high/low.

▪ **Environment.** The environment takes the agent's current state and the action as its inputs and returns the reward as well as the next state as the output. Depending on the problem domain, the environment might be a set of physical laws or chemical reaction rules that processes the actions and establish the corresponding outcomes.

▪ **Observations.** The agent can obtain observations (emission of states) from the environment. In particular, the observation is a (stochastic) function of the state.

▪ **Reward.** The reward is a numerical value that evaluates the fitness (measurement of success) of an agent's actions in the given state. From a given state, an agent takes action in the environment and acquires the new state as well as the reward from the environment. A *cumulative reward* is defined as the summation of discounted rewards: $G(t) = \sum_{k=0}^{n} \gamma^k R(t+k+1)$. The discount factor γ $(0 \leq \gamma \leq 1)$ tunes the importance of future rewards for the current state. The key idea of RL is to find a series of actions that maximize the expected cumulative reward.

▪ **Policy.** The policy of an RL algorithm is typically defined within the context of a Markov decision process [OW12, SB18]. Given the state information, a policy is the suggested action that the agent shall take in order to obtain a high reward.

2.3.4 Genetic Algorithms

Genetic Algorithms (GAs) are a popular subcategory of Evolutionary Algorithms (EAs) and have wide applications in searching-related problems such as feature selection, Knapsack problem, and hardware design optimization [TMKH96, DPAM02]. In GAs, a potential solution is called a chromosome and a set of possible solutions is called the population. The elements contained in the chromosome are referred to as genotypes and the value of the genotype is called phenotype [MG$^+$95, Obi98]. The ‘goodness’ of a solution is quantified using the fitness value where the definition of fitness is problem-specific. To model an optimization problem in the GA paradigm, the designer needs to *encode* the chromosome in specific format depending on the problem of interest. Common encoding forms include binary, integer, real, tree, and permutation [Shi99, Kum13].

Inspired by *natural evolution* in the real world, a GA routine typically involves the following four genetic operations: fitness evaluation, population selection, crossover, and mutation. Figure 2.11 shows the workflow of a standard genetic algorithm. We first initialize the solutions as the first population and encode the solutions as chromosomes. In each generation, a subset of chromosomes are selected as parents to participate in crossover and generate the new population. In the next step, the new population goes through mutation. Finally, the fitness of each solution in the population is evaluated. The solutions with high fitness scores have high probabilities of being selected and produce offsprings in the next round of crossover.

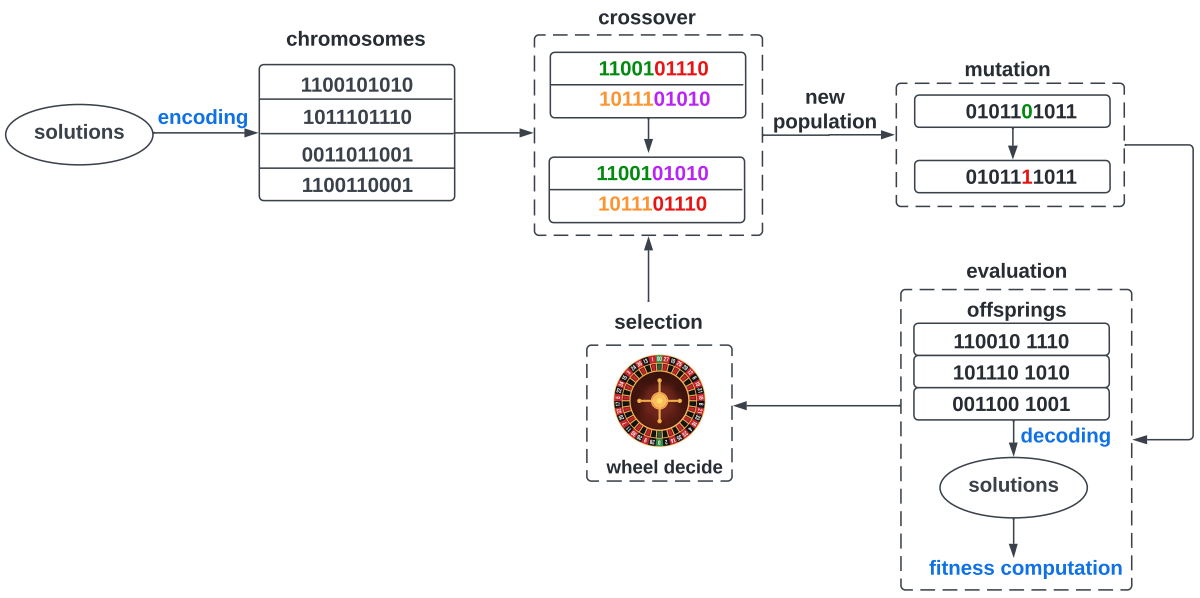


Figure 2.11. High-level workflow of genetic algorithm.

The convergence speed and quality of GAs depend on the distribution of the population. Crossover and mutation increase the diversity of the population, which reduces the probability of the GA getting trapped in local optima. In particular, the exploration and exploitation of GAs shall be balanced dynamically. To address this problem, *diversity* is introduced to evaluate the difference in individuals' gene representation and control the balance mentioned above [Shi99, Urs02, GG12].

As an instance of heuristic-based optimization methods, GAs feature the following advantages: (i) They are applicable when the objective function is not smooth (in which case derivative-based methods cannot be employed); (ii) They search from a population of solutions simultaneously, thus are able to avoid local optima; (iii) They always yield a solution and the solution gets better over time; (iv) The required computation is inherently parallel, which makes them suitable for hardware acceleration. Thanks to these advantages, GAs have been widely used in many real-world applications such as evolvable hardware [SKL06] and code-breaking [Del04]. GAs are also used in parameter selection with large design space, such as DNN structure exploration [SW15] and non-gradient-based training [SMC+17].

2.4 Hardware Security

In this section, we introduce preliminary knowledge about three hardware security problems that are relevant to the contribution of this dissertation.

2.4.1 Disturbance Errors in DRAMs

Memory storage such as Dynamic Random-Access Memory (DRAM) is indispensable for computing systems [DAR09, LNM+17]. Figure 2.12 shows the architecture of modern DRAM chips. A DRAM chip consists of two-dimensional DRAM cells as shown in Figure 2.12 (a) where each cell includes a capacitor and an access transistor. A cell has two states (charged or discharged) and each state denotes a binary value [KDK+14]. DRAMs are addressed using a hierarchical scheme that indicates their hardware organization: channels, DIMMs, ranks, banks,

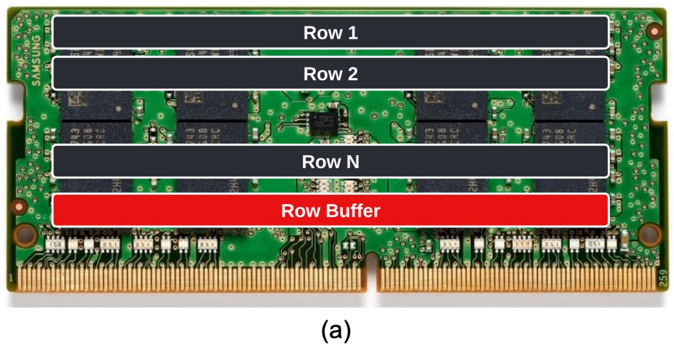


Figure 2.12. DRAM architecture. (a) A DRAM bank is consisted of multiple rows. (b) Internal structure of a DRAM bank.

rows, and columns [PGM+16]. Therefore, the DRAM address is multi-dimensional and can be represented as $< chan,\ DIMM,\ rank,\ bank,\ row,\ col >$.

The susceptibility of commercial DRAMs to *disturbance errors* has been demonstrated by Kim *et.al* in [KDK+14]. This paper finds out that repeated access to a DRAM row can corrupt data in the neighboring rows, i.e., causing bit flips '0' $\rightarrow$ '1' (anti-cell) or '1' $\rightarrow$ '0' (true cell). This disturbance error in DRAMs is called the *Rowhammer Attack (RHA)* [KDK+14, VDVFL+16, TGBR18]. Researchers have found that the disturbance errors in Rowhammer attacks are mostly *repeatable* and *stable* [KDK+14, RGB+16], which means that the locations of the vulnerable DRAM cells are fixed after device manufacturing. The root cause of RHAs is that frequent row activation results in voltage fluctuations, which leads to charge loss of adjacent rows. Exploiting the stability of RHAs, the adversary can perform precise bit flipping at the desired location by profiling the DRAM memory layout [KGGY20, YRF20]. RHAs pose severe security threats to computing platforms since they can evade common data integrity checks and error correction techniques [MK19, GLS+18].

Figure 2.13 shows the schematic of two variants of Rowhammer attacks. Single-sided RHAs perform frequent memory access (i.e., hammering) to one row (so-called the 'aggressor row') which is adjacent to the 'victim row' (where bit flips are desired). Double-sided RHAs simultaneously hammer two aggressor rows located on each side of the victim row [VDVFL+16, GLS+18]. Therefore, double-sided RHA needs to know the DRAM addressing scheme, which is

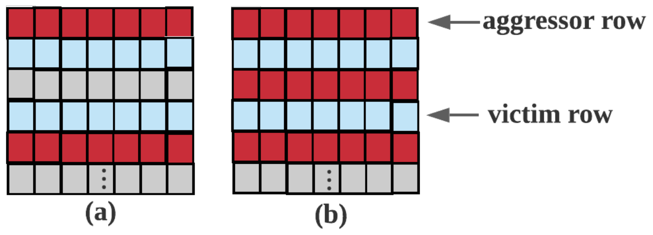


Figure 2.13. Illustration of single-sided (a) and double-sided (b) Rowhammer attacks. The aggressor rows and the victim rows are marked in red and blue color, respectively.

not required by single-sided attacks. Existing works find out the attacker can trigger more bit flips with double-sided [VDVFL$^+$16, TGBR18] and many-sided [FVH$^+$20, dRFV$^+$21] Rowhammer technique compared to the single-sided variant [KDK$^+$14, AAA17].

2.4.2 Circuit Obfuscation

There are two common circuit obfuscation techniques to protect the IP of modern ICs. *IC camouflaging* has been suggested to protect the layout design of the circuit against reverse engineering attacks. Existing IC camouflaging techniques include the insertion of dummy connections and/or cells, as well as doping modification [RSSK13, RPSK12]. The objective of IC camouflaging is to *decouple* the correlation between the appearance and the corresponding functionality of gates. Figure 2.14 shows an example of the layout-level circuit camouflaging. The structure of gates with different functionalities (e.g., NAND and NOR) are different by default as can be seen in Figure 2.14a. However, such a *layout disparity* can be hidden from the attacker by adding redundant connections to both standard gates, resulting in two gates with an identical appearance as shown in Figure 2.14b. As such, layout-level reverse engineering attacks are invalidated since no useful information is leaked from the appearance of the circuit.

As another example of Design-for-Security, *logic locking* has been proposed to protect the intellectual property of ICs by corrupting the functionality of the circuit when incorrect key values are applied to the additional key gates [YS17, HYR21, KAFT22]. Compared to IC camouflaging, logic locking is able to prevent attacks from untrusted fabrication foundries. Existing logic locking techniques include performing XOR/XNOR operations of wires with

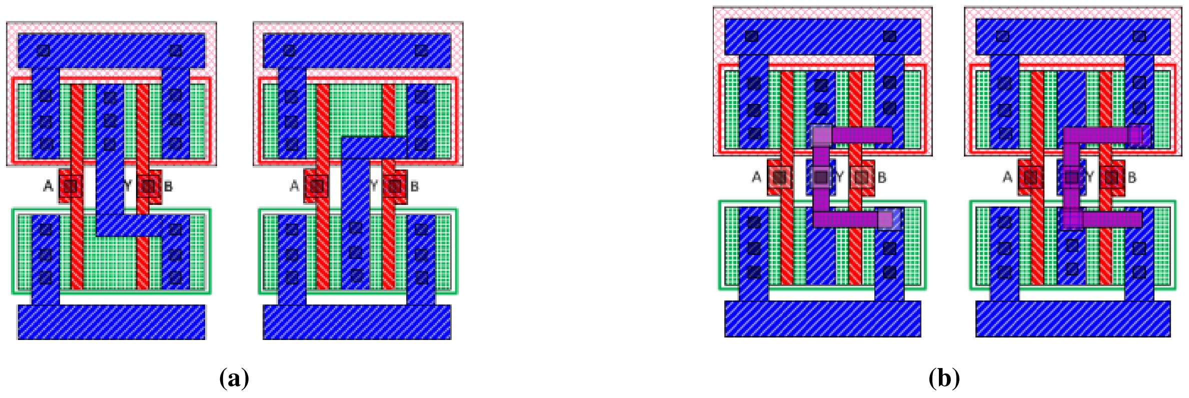


Figure 2.14. Demonstration of IC camouflaging [RSSK13]. (a) Original layouts of a regular 2-input NAND gate (left) and a NOR gate (right). (b) Camouflaged layouts of the corresponding standard cells in (a).

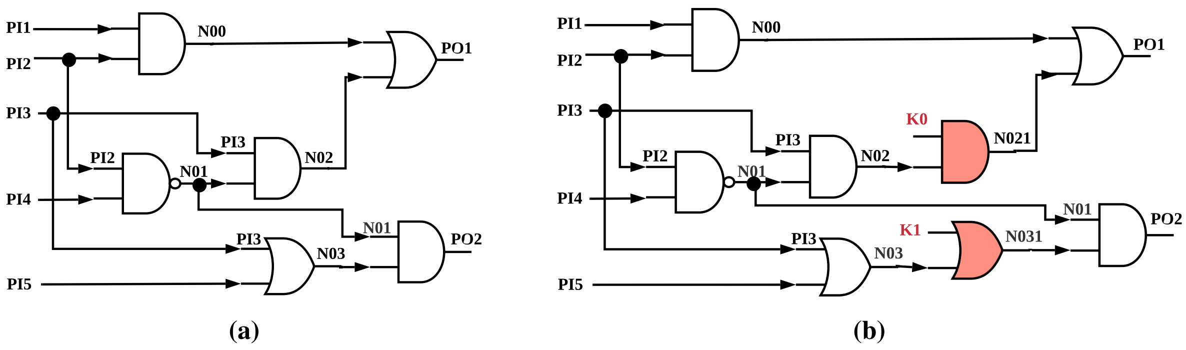


Figure 2.15. Example of logic locking on $c17$ benchmark. The encrypted circuit (b) yields consistent outputs as the original one (a) only when the two-bit key K_0K_1 is set to $2'b10$.

the key inputs [RKM08, RZZ$^+$15], substituting a subset of gates with look-up-tables (LUTs) that stores the key sequence [Bau09], and inserting multiplexers (MUXs) controlled by the key bits [RZZ$^+$15, RPSK12]. An example of logic locking is shown in Figure 2.15.

It is worth noticing that logic locking is applicable to both combinatorial and sequential circuits. For the latter case, the Finite State Machine (FSM) can be locked by adding new states to the original State Transition Graph (STG), adding trap states (i.e., black holes), or changing the deepest state of the circuit to degrade the timing performance [DF19, KAFT22]. To quantify the effectiveness of logic locking, *output corruptibility* is introduced as the metric that measures how the functionality of a circuit is impacted by logic locking. The corruptibility is measured from two aspects when the designer applies an incorrect key to the locked circuit: (i) The number of output pins that produce wrong values; (ii) The number of input patterns that will lead to incorrect primary outputs. The exact value of output corruption is typically quantified using Hamming Distance (HD) [CFZK19b, KAFT22].

While IC camouflaging and logic locking are effective in alleviating the vulnerability of ICs in the semiconductor supply-chain, various attacks have been demonstrated to invalidate these defense techniques. For instance, SAT-based attacks and their variants are able to find a valid decryption key to activate the locked circuit [SRM15, AKHS20]. Removal attacks take a different approach from SAT attacks and deobfuscate the circuit by identifying and removing the protection circuitry consisted of the additional key gates [YMSR17a, CCB19].

Conjunctive Normal Form. It has been proven in theory that every Boolean function can be converted into an equivalent formula in Conjunctive Normal Form (CNF) [BW05]. CNF takes the form of a sequence of *clauses* that are connected by the AND operator. All variables inside the same clause are connected by the OR operator. Representing a circuit netlist with CNF facilitates the verification of whether a given set of constraints are satisfied [FM07, MV07]. As such, SAT-based attacks typically use CNF of the circuit during computation [BW05, SLM+17, AKHS19]. Let us consider the following Boolean expression in the CNF form:

$$O = (x_1 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee \neg x_1). \tag{2.6}$$

This circuit is equivalent to the CNF statements (in dimacs format) shown in Equation (2.7) where each row of the numerical sequence corresponds to one clause in Equation (2.6).

$$\begin{array}{llll} p & \text{cnf} & 3 & 2 \\ 1 & -3 & 0 & \\ 2 & 3 & -1 & 0 \end{array} \tag{2.7}$$

We refer the readers to [Sim04] for detailed steps of CNF conversion and interpretation.

2.4.3 Hardware Trojans

The security of the third-party System-on-Chips (SoCs) has raised an increasing amount of concerns due to the contemporary outsourcing-based supply chain. Hardware Trojans (HTs) are malicious circuit modifications inserted in the circuit to perform the pre-defined adversarial task (i.e., ‘payload’) e.g., circuit malfunction or private information leakage when its control signal (which is called ‘trigger’) is activated [CNB09, BHBN14, MKG$^+$15]. Figure 2.16 shows an example of HT design where a logic-AND gate and an XOR-gate are used as the trigger and payload, respectively. The payload flips the output signal when the trigger is activated, thus disturbing the desired behavior of the original circuit.

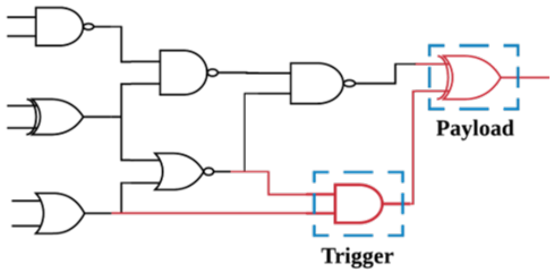


Figure 2.16. Demonstration of the Hardware Trojan attack.

Hardware Trojans can be categorized using different criteria [CNB09, TK10, KRRT10]. For instance, based on the abstraction level where the malicious modification is performed, HTs can be classified into system level, development environment level, register-transfer level, gate level, transistor level, and layout level. According to the activation mechanism, HTs can be categorized into always-on and trigger-controlled (via external or internal trigger signals). Additionally, Hardware Trojans might be inserted into different locations such as the processor, IO, and memory. The physical characteristics criterion defines the hardware realization of the HTs and can be further divided into distribution, size, type, and structure. Particularly, Hardware Trojans fall into two types of physical characteristics: functional class and parametric one. *Functional* Trojans are implemented by adding or removing transistors/gates, while *parametric* Trojans are realized by manipulating existing logic/wires in the circuit [CNB09, TK10].

The collaborative nature of the supply chain also determines that HTs may be inserted by different parties at different stages of the IC life cycle. For instance, the untrusted IP provider, the circuit designer, or the manufacturing party might insert HTs in the circuit. Hardware Trojans shall remain *dormant* in most cases to evade functional testing and HT detection, while it should be successfully activated by the trigger to perform the attack. For this purpose, stealthy HTs are designed with two main considerations: (i) Rare conditions are used to construct the trigger signal; (ii) The HT is placed in a non-critical path to minimize its impact on the side channels (e.g., delay, power, electromagnetic emission)

Detection of Hardware Trojans is challenging due to the facts that: (i) The number of IP cores integrated within contemporary SoCs is large and the complexity of IP blocks is increasing, which makes it difficult to identify small malicious changes introduced by HTs; (ii) The feature size of integrated circuits is getting smaller and smaller, making physical inspection infeasible; (iii) Hardware Trojans are designed to be active only in rare conditions, this intrinsic stealthiness property implies that detecting HTs using random input patterns or test patterns designed for detecting manufacturing faults is hard [CNB09, TK10].

2.5 Acknowledgements

Chapter 2, in part, has been published at has been published at (i) the Proceedings of the 2019 International Conference on Multimedia Retrieval (ICMR) and appeared as: Huili Chen, Bita Darvish Rouhani, Cheng Fu, Jishen Zhao, Farinaz Koushanfar, “DeepMarks: A Secure Fingerprinting Framework for Digital Rights Management of Deep Learning Models”, and (ii) 2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA) and appeared as Huili Chen, Cheng Fu, Bita Darvish Rouhani, Jishen Zhao, Farinaz Koushanfar, “DeepAttest: An End-to-End Attestation Framework for Deep Neural Networks”, and (iii) 2020 INTERSPEECH and appeared as Huili Chen, Bita Darvish, Farinaz Koushanfar, “SpecMark: A Spectral Watermarking Framework for IP Protection of Speech Recognition

Systems”, and (iv) the Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV), 2021 and appeared as Huili Chen, Cheng Fu, Jishen Zhao, Farinaz Koushanfar, “ProFlip: Targeted Trojan Attack with Progressive Bit Flips”, and (v) the Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI), 2019 and appeared as Huili Chen, Cheng Fu, Jishen Zhao, Farinaz Koushanfar, “DeepInspect: A Black-box Trojan Detection and Mitigation Framework for Deep Neural Networks”, and (vi) 2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD) and appeared as Huili Chen, Cheng Fu, Jishen Zhao, Farinaz Koushanfar, “GenUnlock: An Automated Genetic Algorithm Framework for Unlocking Logic Encryption”, and (vii) 2022 ACM Transactions on Embedded Computing Systems (TECS) and appeared as Huili Chen, Xinqiao Zhang, Ke Huang, Farinaz Koushanfar, “AdaTest: Reinforcement Learning and Adaptive Sampling for On-chip Hardware Trojan Detection”. The dissertation author was the primary author of these materials.

Chapter 3

DeepMarks: Secure Fingerprinting Framework for Deep Learning Models

Deep neural networks are revolutionizing various critical fields by providing an unprecedented leap in terms of accuracy and functionality. Because of the expensive training procedure, high-performance DNNs are considered the Intellectual Property (IP) of the model designer and need to be protected. While DNNs are increasingly commercialized, the pre-trained models might be illegally copied or redistributed after they are delivered to malicious users. In this chapter, I introduce DeepMarks, the first *end-to-end, collusion-secure fingerprinting framework* that enables the model owner to retrieve DNN authorship information and identify unique users associated with the models in the context of deep learning.

DeepMarks consists of two main modules: (i) Designing unique fingerprints using anti-collusion codebooks for individual users; and (ii) Encoding each constructed fingerprint (FP) in the probability density function of the weights by incorporating an FP-specific regularization loss during DNN training. We investigate DeepMarks' performance on various datasets and DNN architectures. Experimental results show that the embedded FP preserves the accuracy of the host DNN and is robust against different model modifications that might be conducted by malicious users. Furthermore, DeepMarks is scalable and yields perfect detection rates and no false alarms when identifying the participants of FP collusion attacks under the theoretical guarantee. The runtime overhead of FP retrieval from the marked DNN can be as low as 0.056%.

3.1 Introduction

Recent advances in Deep Learning (DL) have enabled the paradigm shift in diverse domains including autonomous transportation, nuclear engineering, and smart health [LBH15, Sch15, FDFC^{+}18]. Training a highly accurate DNN is costly since this requires: (i) Processing massive amounts of data acquired for the target application; (ii) Allocating substantial computing resources to fine-tune the topology and hyper-parameters of the deployed model. Given the costly process of designing/training, pre-trained DNNs are considered the Intellectual Property (IP) of the model owner and needs to be protected. As an increasing amount of pre-trained DNNs are open-sourced/distributed on the Internet [Caf17, Ama22], IP protection and Digital Right Management (DRM) of these public models are particularly important to maintain the competitiveness of the model owner and facilitate reliable technology transfer.

Digital watermarks and fingerprints have been immensely leveraged to protect the authorship of multi-media content and functional artifacts [KK04, QP07, RET^{+}17]. However, the extension of watermarking and fingerprinting techniques to the DL domain for reliable model distribution is still in its infancy. Developing a practical DNN fingerprinting technique has the following challenges: (C1) Fingerprint (FP) embedding shall not incur performance degradation of the original model; (C2) FP detection shall yield a minimal false alarm rate to avoid the incorrect accusation of innocent customers for misusing/stealing the model; (C3) The embedded FP shall be sustainable to withstand potential model modifications and FP deconstruction attacks conducted by malicious users. This chapter investigates how to tackle the above challenges and presents DeepMarks as a promising solution for large-scale model distribution systems.

A holistic IP protection technique is expected to provide the following two capabilities: **(i) DNN ownership proof.** The model owner shall be able to prove the authorship of her model after the DNN is distributed to the users; **(ii) Tracking/Identifying unique users.** The model owner can trace different customers that are using the same IP and determine which person has misused the model if IP infringement is detected. These two properties are the requirements of IP protection and DRM, respectively. DNNs can be leveraged in either a white-box setting (model internals are publicly known) or a black-box setting (only model outputs are known). DeepMarks aims to provide secure and robust DNN fingerprinting in the white-box scenario, which is a common practice considering the prevalence of DL models on the Internet.

Prior works have proposed DNN watermarking methodologies for model authentication in both the white-box [UNSS17, NUSS18, DRCK19] and the black-box setting [LMPT20, ABC$^+$18, ZGJ$^+$18]. However, all existing watermarking methods only address the first requirement of DNN IP protection (ownership proof) while ignoring the second one (tracking unique users). This is due to the fact that the above-mentioned DNN watermarking techniques typically are not concerned with the co-existence of multiple users that might employ the same model IP. This chapter demonstrates that the state-of-art DNN watermarking scheme [UNSS17] is deficient to provide a robust fingerprinting solution. More specifically, we show that multiple users can collaborate and construct an unmarked model that achieves a comparable accuracy as the baseline model using their individually watermarked models. This type of attack is called the *'FP collusion attack'* and can defeat the DNN watermarking approach in [UNSS17]. DeepMarks is motivated to overcome this vulnerability of DNN watermarking schemes.

This chapter presents DeepMarks, the first *provably secure DNN fingerprinting framework* that empowers coherent integration of robust digital markers into DL models. DeepMarks takes the pre-trained DNN (which is the owner's IP) together with a set of security parameters as its inputs. Multiple functionality-preserved variants of the original model are returned as the outputs, carrying unique fingerprints of individual users in the model distribution system. We address the challenges (C1-C3) by designing *collusion-aware* fingerprints and encoding the FP information in weights using a customized regularization loss during DNN re-training. The intuition behind DeepMarks is that there are abundant redundancies existing in the pre-trained DNN due to its high dimensionality. We leverage such redundancies and tackles DNN fingerprinting as an auxiliary task of the original data application. As such, the designed fingerprint is integrated as an inseparable part of the weight parameters, ensuring that the adversary cannot remove the FP without compromising the performance of the marked model.

DeepMarks framework is innovative in the sense that it is the first collusion-secure fingerprinting framework with theoretical guarantees on detection performance. Unlike prior works that only focus on addressing model ownership authentication for a single user, we consider a large-scale model distribution system where multiple users might perform collaborative FP deconstruction attacks. Such a scenario is more practical considering the real-world setting. Furthermore, DeepMarks provides model ownership proof and digital right management simultaneously, thus providing the first full-fledged IP protection solution in the deep learning domain. Our approach is generic and compatible with various applications as well as DNN architectures.

This work makes the following contributions:

- **Enabling robust IP protection and digital right management for DNNs in model distribution systems.** We propose DeepMarks, a novel fingerprinting methodology that encodes robust fingerprints in the probability density function (pdf) of weights for model ownership proof and unique user tracing. DeepMarks is provably more robust against FP collusion attacks compared to the state-of-the-art DNN watermarking scheme.

- **Characterizing the requirements for an effective fingerprinting methodology in the deep learning domain.** We introduce a comprehensive set of metrics to assess the performance of a DNN fingerprinting methodology. Such metrics provide new perspectives for model designers and facilitate a coherent comparison of current and pending DNN IP protection techniques.

- **Investigating the performance of DeepMarks on various DNN benchmarks.** We perform extensive proof-of-concept experiments to corroborate the efficacy and robustness of DeepMarks. Empirical results show that our framework yields perfect FP detection rates and no false alarms given the properly selected security parameters.

We emphasize that enabling the model owners to *prove model authorship and trace back illegal model usages* is important due to the prevalence of DNNs in critical fields. We are motivated to address the pressing concerns about the IP and digital right management of valuable DL models. DeepMarks opens a new axis for the growing research in secure deep learning and sheds light on the unexplored limitations of DNN watermarking techniques.

3.2 Related Works

IP protection of valuable DNN models has been a subject of increasing interest to both researchers and practitioners. Uchida *et al.* take the first step towards DNN watermarking and propose to embed the watermark (WM) by adding constraints to the weight parameters. The WM is later extracted from the marked layer assuming a white-box scenario [UNSS17]. To alleviate the constraint that the parameters of the queried model are available during WM extraction, several papers propose zero-bit watermarking techniques that are applicable in the black-box scenario [LMPT20, ABC$^+$18, ZGJ$^+$18]. These works suggest different methods to generate watermark images and labels as the 'trigger set', which is then used to tweak the decision boundary of the pre-trained model for WM embedding. In this scenario, the WM existence is determined by querying the remote model with the WM images and thresholding the accuracy on the trigger set. It is worth noting that all of the above-mentioned papers consider a single-user setting and are unaware of the potential collusion attacks in multi-user scenarios. In this chapter, we present a collusion-secure DNN fingerprinting framework to address the limitations of DNN watermarking, thus providing a holistic IP protection solution.

3.3 Problem Formulation

While cloud-based DNN services are widely adopted in various applications, white-box DL model deployment provides a more powerful utilization alternative that encourages research communities and industrial developers to improve existing DL techniques. DeepMarks is motivated to protect the IP of white-box DNNs in a model sharing system. To the best of our knowledge, there is no prior work on DNN fingerprinting. In this chapter, we define fingerprinting as the task of designing a v-bit binary code-vector $\mathbf{c_j} \in \{0,1\}^v$ for each user and embedding it in the parameters (e.g., weights) of one/multiple layers in the host neural network. Here, $j = 1,...,n$ is the index for each distributed user and n is the total number of users. The objective of fingerprinting is *two-fold*: (i) Claiming the ownership of a specific DNN, and (ii) Tracing the unintended usage of the model conducted by the distributed users.

In the following of this sections, we introduce a set of requirements for effective DNN fingerprinting and discuss potential attacks that might render the embedded FPs ineffective.

3.3.1 Requirements

Table 3.1 summarizes the requirements for an effective fingerprinting technique in the DL domain. In addition to fidelity, efficiency, security, reliability, integrity, and robustness requirements shared between fingerprinting and watermarking, a successful fingerprinting methodology should also satisfy uniqueness, scalability, and collusion resilience criteria. *Uniqueness* is the

Table 3.1. Requirements for an effective fingerprinting methodology of deep neural networks.

Requirements	Description
Fidelity	The accuracy of the target neural network shall not be degraded as a result of fingerprint embedding.
Uniqueness	The fingerprint need to be unique for each user to achieve unambiguous identification.
Efficiency	The overhead of fingerprint embedding and extraction shall be negligible.
Security	Fingerprint embedding shall leave no tangible footprint in the host neural network; thus, an unauthorized individual cannot detect the presence of a fingerprint in the model.
Robustness	The fingerprint shall be robust against potential fingerprint destruction and model modification attacks.
Reliability	Fingerprint extraction shall yield minimal false negatives to ensure high detection rates.
Integrity	The fingerprinting methodology should yield minimal false alarm (a.k.a., false positive). This means that the probability of an innocent user being accused as a colluder should be very low.
Scalability	The fingerprinting methodology should be able to support numerous users in the distributed system.
Generality	The fingerprinting technique should be applicable to various datasets and network architectures.

intrinsic property of fingerprints that enable unambiguous user identification. *Scalability* is a key factor to support model ownership authentication and DRM in large-scale systems. *Collusion resistance* is a desired property considering the practicality of collusion attacks.

3.3.2 Threat Model

Corresponding to the robustness requirement in Table 3.1, we discuss four types of DL domain-specific attacks that the DNN fingerprinting technique should be resistant to: model fine-tuning, parameter pruning, fingerprint collusion, and fingerprint overwriting attacks.

Parameter Pruning. Genuine users may leverage parameter pruning to reduce the memory and computation overhead of the DNN [HPTD15, LWL17] while adversaries may apply pruning to remove the FP. As such, an effective fingerprinting technique shall be resistant to parameter pruning that incurs the change of model parameters.

Model Fine-tuning. Fine-tuning might be performed by honest users for transfer learning, or by malicious attackers to remove the FP. Since the parameters that carry the FP are altered during fine-tuning, the embedded FP should be robust against this modification.

Fingerprint Collusion Attack. A group of users who have the same host neural network with different embedded fingerprints may perform collusion attacks to construct a functional model where no fingerprints can be detected by the owner. In this work, we focus on evaluating DeepMarks' robustness against the FP averaging attack.

Fingerprint Overwriting. Assuming an active adversary knows the deployed fingerprinting methodology, he may embed a new FP to destroy the original one inserted by the authentic model owner. While the location where the original FP is embedded shall be a secret to the malicious parties, it is conceivable that the attacker can embed the new FP into multiple layers of the target DNN to increase the success rate of destroying the original FP.

3.4 DeepMarks Framework

Figure 3.1 demonstrates the global flow of DeepMarks framework. DeepMarks performs DNN fingerprinting by embedding the designated fingerprint information in the probability distribution of weights at selected layers. The fingerprinted model is assumed to be deployed in a white-box setting where the model internals are transparent to the public. Such an assumption is practical considering the popularity of model sharing/distribution in the real-world setting. There are two types of FP modulation schemes in the multi-media domain: orthogonal modulation [WWZ$^+$05, KM09], and coded modulation [WTWL04, YLCZ10]. Since coded fingerprinting achieves better collusion resilience and can be considered a general case of orthogonal fingerprinting [TWL02, TWWL03], we focus on code-modulated fingerprinting in this work. Note that DeepMarks is orthogonal to the existing code modulation schemes and can be further augmented when advanced modulation methods are integrated. To enable model

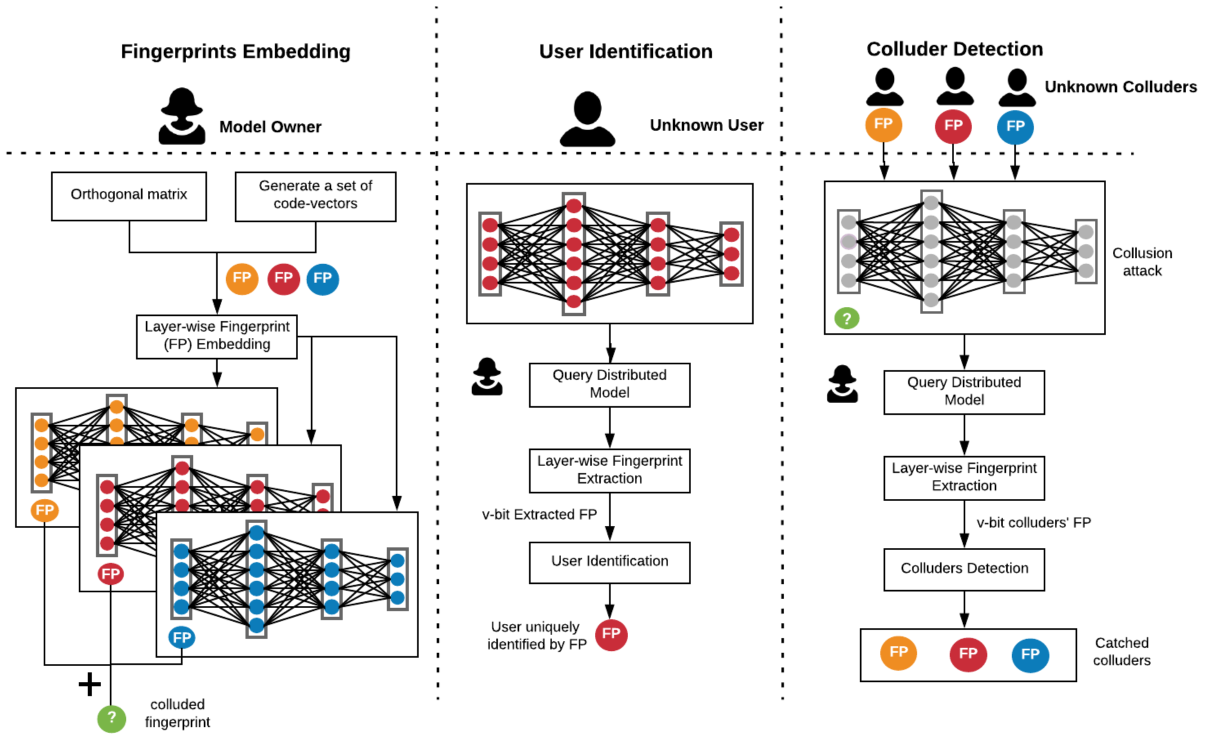


Figure 3.1. DeepMarks Global Flow. DeepMarks consists of three main modules: FP embedding, user identification, and colluder detection.

ownership authentication and digital right management, DeepMarks allows the model owner to retrieve the embedded fingerprints for user identification as well as colluder detection after distributing the fingerprinted models.

3.4.1 Fingerprint Embedding

DeepMarks' regularization-based FP embedding is inspired by *constraint-based watermarking systems* in the multi-media domain [KLMS+98, KLMS+01, AHTA03]. More specifically, the original problem (e.g., image classification) is used as the cover constraint and FP embedding is incorporated as the additional stego constraint. We leverage the *over-parameterization* of high dimensional DNNs to enforce the stego constraints. As such, DeepMarks helps to alleviate model over-fitting and preserve the performance of the original model. Embedding FPs in the training-from-scratch fashion is impractical for large-scale distributed systems since the fingerprinting process is required for each copy of the target DNN. DeepMarks tackles this viability concern by treating FP embedding as a *post-processing* step implemented via fine-tuning the pre-trained model with the FP-specific regularization loss. Particularly, FP embedding is formulated as an *off-line, one-time* process performed locally by the owner before model distribution. We demonstrate the construction and embedding of code-modulated FPs based on DeepMarks framework as follows.

(I) Fingerprint Construction. DeepMarks ensures provable collusion resilience by taking advantage of the *Anti-Collusion Code* (ACC) theory when constructing the codebook. ACC is proposed in [WTWL04] for collusion-resistant coded fingerprinting and has the following property: the composition of any subset of K or fewer code-vectors is unique. This property allows the owner to identify a group of K or fewer colluders from the composition precisely. A K-resilient AND-ACC codebook is a matrix where the element-wise composition is logic-AND and allows for the accurate identification of K unique colluders from their composition.

To generate ACC of binary values, DeepMarks deploys Balanced Incomplete Block Design (BIBD) [YLCZ10]. A (v,k,λ)-BIBD has $b = \lambda(v^2-v)/(k^2-k)$ blocks with block size k.

The BIBD can be represented by its corresponding incidence matrix $\mathbf{C}_{v\times b}$ where each element:

$$c_{ij} = \begin{cases} 1, \text{ if } i^{th} \text{ value occurs in } j^{th} \text{ block} \\ 0, \text{ otherwise.} \end{cases}$$

By setting the number of concurrent occurrences to one ($\lambda = 1$) and assigning the bit complement of columns of the incidence matrix $\mathbf{C}_{v\times b}$ as the code-vectors, the resulting $(v,k,1)$-BIBD code is $(k-1)$-resilient and supports up to $n = b$ users [TWWL03]. Note that DeepMarks is generic and compatible with other anti-collusion code design schemes. Here, we focus on illustrating the feasibility of DeepMarks and leave advanced codebook construction to future work.

DeepMarks generates code-modulated fingerprints as follows. Given the designed incidence matrix $\mathbf{C}_{v\times b}$, the coefficient matrix $\mathbf{B}_{v\times b}$ for FPs is computed from the linear mapping $b_{ij} = 2c_{ij} - 1$. The FP of the j^{th} user is then generated from an orthogonal matrix $\mathbf{U}_{v\times v}$ and the coefficient matrix $\mathbf{B}_{v\times b}$ as:

$$\mathbf{f_j} = \sum_{i=1}^{v} b_{ij}\mathbf{u_i}, \tag{3.1}$$

where $\mathbf{b_j} \in \{\pm 1\}^v$ is the coefficient of user j. $\mathbf{U}$ is generated from element-wise Gaussian distribution for security consideration [WTWL04].

(II) Fingerprint Insertion. The FP obtained from Equation (3.1) is embedded in the selected layers of the pre-trained model by incorporating an FP-specific embedding loss term to the conventional loss function ($\mathscr{L}_0$):

$$\mathscr{L} = \mathscr{L}_0 + \gamma\, MSE(\mathbf{f_j} - \mathbf{Xw}). \tag{3.2}$$

Here, MSE is the mean square error function, γ is the embedding strength that controls the contribution of FP embedding loss, $\mathbf{X}$ is the owner's secret projection matrix generated from standard normal distribution $\mathscr{N}(0,1)$. The vector $\mathbf{w}$ is the flattened averaged weights of the target layers that carry the FP information.

As a proof-of-concept analysis, we embed the FP ($\mathbf{f_j}$) in a convolutional layer of the host DNN. The weight is a 4D tensor $\mathbf{W} \in \mathbb{R}^{D \times D \times F \times H}$ where D is the kernel size, F and H is the number of input and output channels, respectively. We average the weight $\mathbf{W}$ over the output channel dimension and stretch the result to a vector $\mathbf{w}$. The FP is embedded in the vector $\mathbf{w} \in \mathbb{R}^N$ where $N = D \times D \times F$ is the embedding dimension. The additive FP embedding loss $\mathscr{L}_{FP} = MSE(\mathbf{f_j} - \mathbf{Xw})$ is minimized together with the conventional loss during DNN training to encode the FP $\mathbf{f_j}$ in the pdf of weights in the selected layer. We assume that the three matrices $\mathbf{U}$, $\mathbf{C}$, $\mathbf{X}$, and the layers selected for FP embedding are secret security parameters that are only known to the owner. The main difference between DeepMarks' FP embedding and *transfer learning* is that the latter one involves training with a new dataset.

3.4.2 Fingerprint Extraction

In this section, we described how DeepMarks extracts the embedded fingerprint from the marked model for two purposes: identifying unique users and detecting participants of fingerprint collusion attacks.

User Identification

DeepMarks uniquely identifies each individual user by recovering his/her associated code-vector assuming the availability of model parameters. To do so, DeepMarks undergoes four main steps: **(i)** Acquiring the weights in the marked layers to reconstruct the FP vector $\widetilde{\mathbf{f_j}} = \mathbf{X}\widetilde{\mathbf{w}}_\mathbf{j}$; **(ii)** Recovering the correlation score vector from the FP vector and the owner's secret basis matrix by computing $\widetilde{\mathbf{b_j}} = \widetilde{\mathbf{f_j}}^T \mathbf{U}$; **(iii)** Decoding the ACC code-vector $\widetilde{\mathbf{c_j}}$ from the element-wise hard-thresholding of $\widetilde{\mathbf{b_j}}$; **(iv)** Comparing the recovered code-vector $\widetilde{\mathbf{c_j}}$ with each column in the owner's codebook $\mathbf{C}$ where the matching position uniquely identifies the user. We use Bit Error Rate (BER) computed between the true code-vector and the recovered one to assess DeepMarks' performance of FP extraction. The user identification process is considered successful if there exists one unique matching (BER=0 for a column in $\mathbf{C}$).

To illustrate DeepMarks' workflow for user identification, let us consider a $(7,3,1)$-BIBD codebook shown in Equation (3.3). The FPs for 7 users (shown in Equation (3.4)) are constructed using the columns of the codebook $\mathbf{C}$ and the basis matrix $\mathbf{U}$ as described earlier.

$$\mathbf{C} = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}, \tag{3.3}$$

$$\begin{cases} \mathbf{f_1} = -\mathbf{u_1} - \mathbf{u_2} + \mathbf{u_3} - \mathbf{u_4} + \mathbf{u_5} + \mathbf{u_6} + \mathbf{u_7}, \\ \dots \\ \mathbf{f_6} = +\mathbf{u_1} + \mathbf{u_2} - \mathbf{u_3} - \mathbf{u_4} + \mathbf{u_5} + \mathbf{u_6} - \mathbf{u_7}, \\ \mathbf{f_7} = +\mathbf{u_1} + \mathbf{u_2} + \mathbf{u_3} - \mathbf{u_4} - \mathbf{u_5} - \mathbf{u_6} + \mathbf{u_7}, \end{cases} \tag{3.4}$$

Note that for user 1, her coefficient vector can be recovered by computing the correlation scores:

$$\widetilde{\mathbf{b_1}} = \mathbf{f_1}^T[\mathbf{u_1}, ..., \mathbf{u_7}] = [-1, -1, +1, -1, +1, +1, +1].$$

The corresponding code-vector is then extracted by the inverse linear mapping $c_{ij} = \frac{1}{2}(b_{ij} + 1)$, resulting in $\widetilde{\mathbf{c_1}} = [0, 0, 1, 0, 1, 1, 1]$. The recovered $\widetilde{\mathbf{c_1}}$ is exactly the same as the first column of $\mathbf{C}$, indicating the effectiveness of DeepMarks framework for user identification.

Colluder Detection

In this section, we describe how DeepMarks deploys the intrinsic asset of AND-ACC for colluders detection. We focus on the *FP averaging attack*, which is a typical and cost-effective FP collusion attack, in our evaluation. Furthermore, we consider the worst-case scenario where the colluders know the positions of the embedded layers. As a result, the colluders can perform element-wise average on their weights and produce $\widetilde{\mathbf{w}}_{\mathbf{avg}}$ to answer the owner's inquiry. The owner then computes the colluded correlation vector $\widetilde{\mathbf{b}}_{\mathbf{avg}}$ as follows:

$$\widetilde{\mathbf{f}}_{\mathbf{avg}} = \mathbf{X}\widetilde{\mathbf{w}}_{\mathbf{avg}}, \tag{3.5}$$

$$\widetilde{\mathbf{b}}_{\mathbf{avg}} = (\widetilde{\mathbf{f}}_{\mathbf{avg}})^{\mathbf{T}}\mathbf{U}. \tag{3.6}$$

DeepMarks leverages hard-thresholding detectors for colluder identification. The ACC code-vector is decoded from the correlation vector $\widetilde{\mathbf{b}}_{\mathbf{avg}} = [\widetilde{b}^1_{avg}, ..., \widetilde{b}^v_{avg}]$ by comparing each element with an owner-defined threshold τ:

$$\widetilde{c}^i_{avg} = \begin{cases} 1, & \text{if } \widetilde{b}^i_{avg} > \tau, \\ 0, & \text{otherwise.} \end{cases} \tag{3.7}$$

Given the AND-ACC code-vector of the colluders $\widetilde{\mathbf{c}}_{\mathbf{avg}}$, the remaining problem is to find the subsets of columns from the codebook $\mathbf{C}$ such that their logic-AND composition is equal to $\widetilde{\mathbf{c}}_{\mathbf{avg}}$. For a $(v,k,1)$-BIBD-ACC, at most $(k-1)$ colluders can be uniquely identified [TWWL03] with theoretical guarantee.

As an example, we demonstrate DeepMarks' colluder detection scheme using the codebook in Equation (3.3). Assuming user 6 and user 7 collaboratively generate the averaged fingerprint as follows:

$$\mathbf{f}_{\mathbf{avg}} = \frac{1}{2}(\mathbf{f_6} + \mathbf{f_7}) = \frac{1}{2}(2\mathbf{u_1} + 2\mathbf{u_2} - 2\mathbf{u_4}),$$

where individual FPs are defined in Equation (3.4). The owner computes the colluders' correlation vector as follows:

$$\mathbf{b}_{\mathbf{avg}} = (\mathbf{f}_{\mathbf{avg}})^{\mathbf{T}}\mathbf{U} = [1,1,0,-1,0,0,0].$$

The corresponding code-vector is then extracted according to decision rule in Equation (3.7), resulting in $\mathbf{c_{avg}} = [1,1,0,0,0,0,0]$ One can observe that the logic-AND composition of column 6 and column 7 in the codebook $\mathbf{C}$ is exactly equal to $\mathbf{c_{avg}}$, while all the other compositions (of two or more columns) do not satisfy the constraint. This example shows that DeepMarks correctly identifies all participants of the collusion attack without any false alarms.

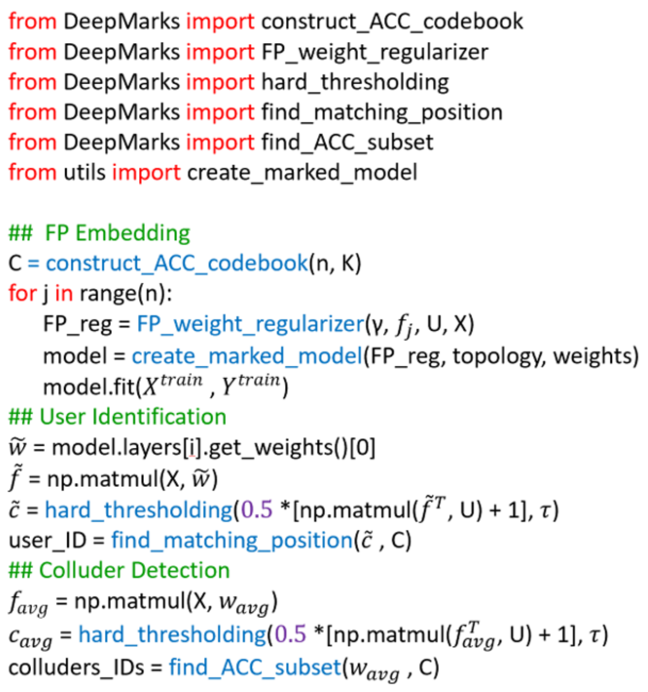
3.4.3 Computation Overhead Analysis

We discuss the overhead of a DNN fingerprinting technique from two perspectives: FP embedding, and FP extraction. Since FP embedding locally performed locally by the owner before model distribution, there is no communication overhead involved. The computation overhead is determined by the additional operations to compute the FP-specific loss $\mathscr{L}_{FP} = MSE(\mathbf{f}_j - \mathbf{X}\mathbf{w})$ during DNN training, which has complexity $\mathscr{O}(vN+v)$. To extract the FP, the queried user sends the vector $\widetilde{\mathbf{w}}_{N\times 1}$ of the marked layer to the owner, thus the communication overhead is $\mathscr{O}(N)$. The computation overhead is incurred by two matrix multiplications: $\widetilde{\mathbf{f}} = \mathbf{X}_{v\times N} \cdot \widetilde{\mathbf{w}}_{N\times 1}$, $\widetilde{\mathbf{b}} = \widetilde{\mathbf{f}}^T_{1\times v} \cdot \mathbf{U}_{v\times v}$ with complexity $\mathscr{O}(vN)$ and $\mathscr{O}(v^2)$, respectively. We provide the quantitative runtime overhead results in Section 3.6.

3.5 DeepMarks as a High-level API

DeepMarks minimizes the required data movement to ensure maximal data reuse and a minimal overhead caused by fingerprint embedding. To do so, we integrate the computation of additive loss term to the DNN tensor graph so that the gradients with respect to the FP embedding loss are computed during the regular back-propagation and all computation for FP embedding is performed homogeneously on GPU. Separately modeling the fingerprinting graph significantly slows the DNN training process since the weight parameters need to be completely transferred from the original DNN graph during the forward pass to compute the FP loss and update the parameters of the FP graph. This approach, in turn, further presses the already constrained memory. Our homogeneous solution reuses the weights within the original graph with minimal memory overhead.

Figure 3.2 shows the prototype of functions for the three main modules in DeepMarks framework. DeepMarks library enables the owner to construct an anti-collusion codebook by providing a function called *construct_ACC_codebook* that takes in the total number of users (n) and the desired resilience level (K) specified by the owner. To embed the FP in the pre-trained model, a customized weight regularizer *FP_weight_regularizer* is applied to compute $\mathscr{L}_{FP}$ and returns the total regularized loss. Our accompanying library is equipped with functions *hard_thresholding* and *find_matching_position* to identify the queried user and the function *find_ACC_subset* to detect colluders. DeepMarks' customized library supports acceleration on GPU platforms. Our provided wrapper can be readily integrated within well-known DL frameworks including TensorFlow, Pytorch, and Theano.



```
from DeepMarks import construct_ACC_codebook
from DeepMarks import FP_weight_regularizer
from DeepMarks import hard_thresholding
from DeepMarks import find_matching_position
from DeepMarks import find_ACC_subset
from utils import create_marked_model

##  FP Embedding
C = construct_ACC_codebook(n, K)
for j in range(n):
    FP_reg = FP_weight_regularizer(γ, $f_j$, U, X)
    model = create_marked_model(FP_reg, topology, weights)
    model.fit($X^{train}$ , $Y^{train}$)
## User Identification
$\widetilde{w}$ = model.layers[i].get_weights()[0]
$\tilde{f}$ = np.matmul(X, $\widetilde{w}$)
$\tilde{c}$ = hard_thresholding(0.5 *[np.matmul($\tilde{f}^T$, U) + 1], $\tau$)
user_ID = find_matching_position($\tilde{c}$ , C)
## Colluder Detection
$f_{avg}$ = np.matmul(X, $w_{avg}$)
$c_{avg}$ = hard_thresholding(0.5 *[np.matmul($f_{avg}^T$, U) + 1], $\tau$)
colluders_IDs = find_ACC_subset($w_{avg}$ , C)
```

Figure 3.2. DeepMarks library usage and resource management for FP embedding, user identification, and colluder detection.

3.6 Evaluation Results

In this section, we present the evaluation of DeepMarks on image classification tasks using two popular types of DL models: Convolutional Neural Networks (CNNs) and Wide Residual Networks (WRNs). The benchmark datasets and topologies are summarized in Table 3.2. We use ReLU as the activation function in all benchmarks. Recall that DeepMarks leverages strategical regularization during DNN training, thus is generic and applicable to various DNN architectures such as Multi-layer Perceptrons (MLP) and Recurrent Neural Networks (RNN). We want to emphasize that DeepMarks does *not require prior knowledge about the number of colluders* (k) in the detection stage. All participants of the collusion attack are automatically identified by DeepMarks as discussed in Section 3.4.2, rendering the detection scheme useful in practice.

Table 3.2. Benchmarks of DNN architectures. Here, $64C3(1)$ indicates a convolutional layer with 64 output channels and 3×3 filters applied with a stride of 2, $MP2(1)$ denotes a max-pooling layer over regions of size 2×2 and stride of 1, and $512FC$ is a fully-connected layer with 512 output neurons.

Dataset	Model Type	Architecture
MNIST	CNN	784-32C3(1)-32C3(1)-MP2(1)-64C3(1)-64C3(1)-512FC-10FC
CIFAR10	WRN	Please refer to [ZK16]

Experimental Setup. To evaluate the performance of DeepMarks coded fingerprinting scheme, we use a $(31,6,1)$-BIBD AND-ACC codebook that accommodates 31 users. We select the embedding strength γ in Equation (3.2) such that the embedding loss satisfies $\mathscr{L}_{FP} = 0.1 \cdot \mathscr{L}_0$ in the beginning of FP embedding. The total FP-regularized loss of the model is minimized during regular back-propagation. DeepMarks employs the BER computed between the code-vector recovered from the current weights and the ground-truth value as a 'monitor' to terminate FP embedding when BER=0. In our experiments, we use $\gamma = 0.1$ and retrain the target DNN for 5 epochs with the learning rate at the last stage of original training across all benchmarks. The threshold for code-vector extraction is set to $\tau = 0.85$ without explicit hyper-parameter tuning. We demonstrate a comprehensive examination of DeepMarks' performance (Section 3.6) and the comparison with the state-of-the-art DNN watermarking technique (Section 3.6.2) as follows.

3.6.1 DeepMarks Properties Evaluation

In the following of this section, we assess DeepMarks' performance based on the requirements discussed in Table 3.1.

Fidelity

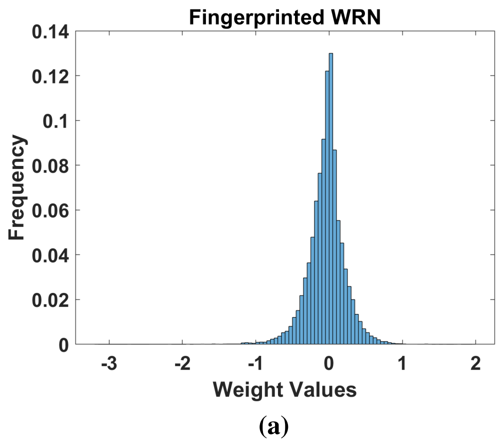
DeepMarks meets the fidelity criterion by preserving the model's functionality. To study the effect of FP embedding on the functionality of the original task, we compare the test accuracy of the pre-trained baseline model, the fine-tuned model with and without the FP embedding loss. The results are summarized in Table 3.3. One can see from the comparison that embedding FPs in the DNN does not induce accuracy drop and can even slightly improve the accuracy of the target DNN. This is due to the fact that the additive embedding loss in Equation (3.2) introduces regularization and alleviates model over-fitting.

Table 3.3. Fidelity requirement. The baseline accuracy is preserved after fingerprint embedding in the underlying benchmarks.

Benchmark	MNIST-CNN			CIFAR10-WRN		
Setting	Baseline	Fine-tune without fingerprint	Fine-tune with fingerprint	Baseline	Fine-tune without fingerprint	Fine-tune with fingerprint
Test Accuracy (%)	99.52	99.66	99.72	91.85	91.99	92.03

Security

DeepMarks respects the security criterion by preserving the intrinsic distribution of weights. To prevent the adversary from detecting the existence of an FP in the model, security requires the embedding of the fingerprint to leave no tangible changes in the distribution of the model parameters. Figure 3.3 shows the histograms of weights at the selected layer with and without the FP on the CIFAR10-WRN benchmark. The similarity between these two histograms corroborates DeepMarks security.



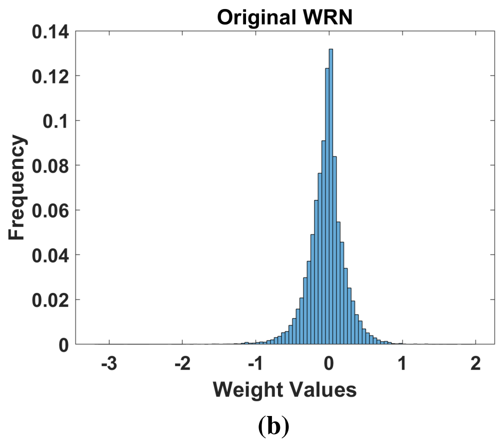
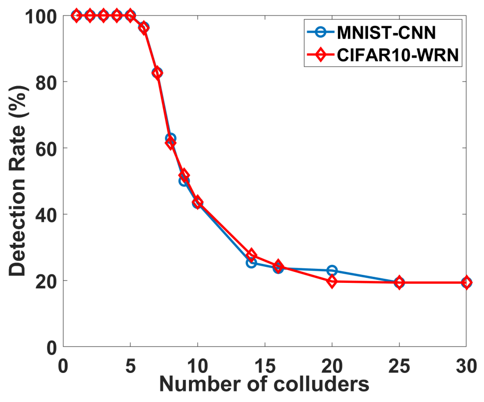


Figure 3.3. Histogram of the weights at the selected layer in the fingerprinted model (a) and the original model (b).

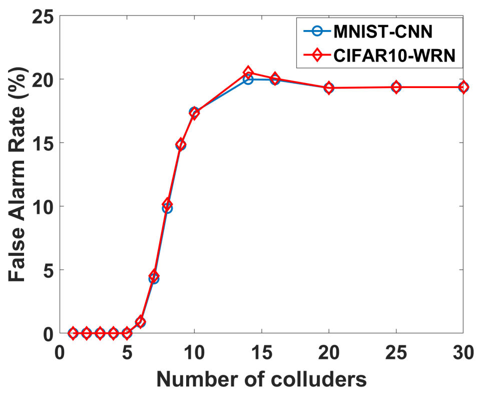
Robustness, Reliability, and Integrity

DeepMarks yields high detection rates and low false alarm rates for user identification and colluder detection under various attacks. We consider two FP deconstruction attacks: FP collusion and FP overwriting, and two model modification attacks: model fine-tuning and parameter pruning as discussed in Section 3.3.2. For a given number of colluders, we run $1,000$ random simulations to generate different colluders sets from all users and report the average performance. When the colluder set is too large to be uniquely identified by the property of ACC, we consider all feasible colluder sets that match the extracted code-vector resulting from FP collusion. We detail the settings and results of each attack as follows.

(I) Fingerprints Collusion. The FP averaging attack is described in Section 3.4.2. Figure 3.4 shows the detection (true positive) rates and false alarm (false positive) rates of DeepMarks when different numbers of users participate in the collusion attack. DeepMarks features ideal detection rates and false alarm rates when the number of colluders is smaller than or equal to the theoretical threshold $(k-1)$ guaranteed by the ACC codebook. As such, DeepMarks satisfies the *reliability* and *integrity* requirements in Table 3.1. The consistency with the theorem corroborates that DeepMarks provides provable collusion resistance.



(a)



(b)

Figure 3.4. Detection rates (a) and false alarm rates (b) of DeepMarks against fingerprint collusion attacks.

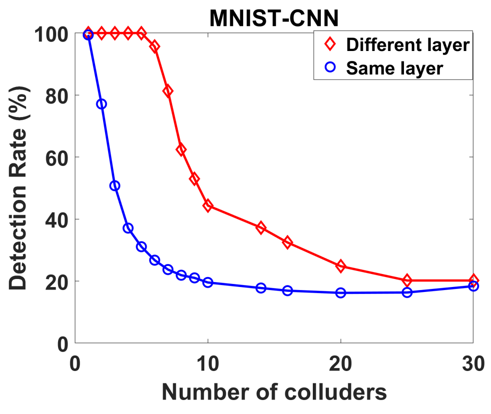
(II) Fingerprint Overwriting. Besides FP collusion, an active adversary that is aware of the fingerprinting method may try to destroy the original FP by embedding a new one in the marked DNN. To perform the attack, the adversary generates a new set of secret matrices (**C**, **U**, **X**), constructs his own FP, and randomly selects a layer in the distributed model to embed the FP as outlined in Section 3.4.1. In our experiment, we assume both the original FP and the new FP deploy single-layer embedding for simplicity.

Table 3.4 summarizes the results of DeepMarks' user identification performance when the FP overwriting attack is performed on a different layer or the same layer as the original FP. In our experiments, we assume all 31 users individually implement FP overwriting attacks on their fingerprinted models and report the average metrics. DeepMarks retains perfect user identification when the overwriting attack occurs at a different layer while incurs false negatives (non-zero BER) when the same layer is attacked.

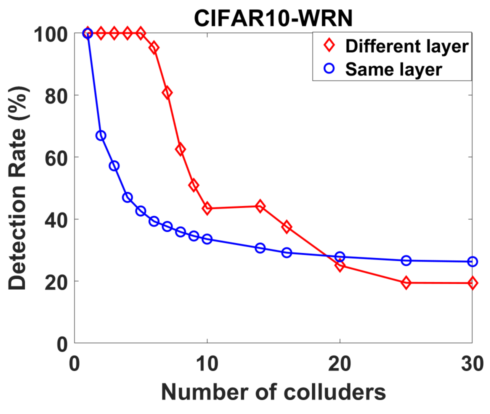
Table 3.4. DeepMarks' user identification in case of FP overwriting attack at a different or the same layer.

Overwrite Condition	Overwrite Different Layer		Overwrite Same Layer	
Metrics	Accuracy (%)	BER	Accuracy (%)	BER
MNIST-CNN	99.68	0	99.69	0.06
CIFAR10-WRN	91.90	0	91.96	0.01

We further assess DeepMarks' collusion resilience against FP overwriting attacks and show the results in Figure 3.5. In this case, the colluders first agree on which layers to embed their FPs and obtain the overwritten models. Then the weights at the attacked layers are averaged across colluders and used as the response to the owner's query. Comparing Figure 3.5 with Figure 3.4a, it can be seen that DeepMarks' colluders detection performance is not degraded by FP overwriting if the colluders embed their new FPs in different layers as the original one. When the originally marked layer is attacked, the detection rate has a significant drop in case of a small number of colluders. Although the colluders may embed new FPs in multiple layers to increase the chance of finding the secret embedding position of the original FP, such an approach introduces excessive regularization and might incur performance degradation.



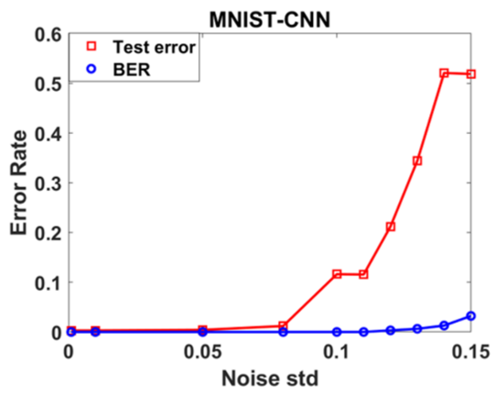
(a)



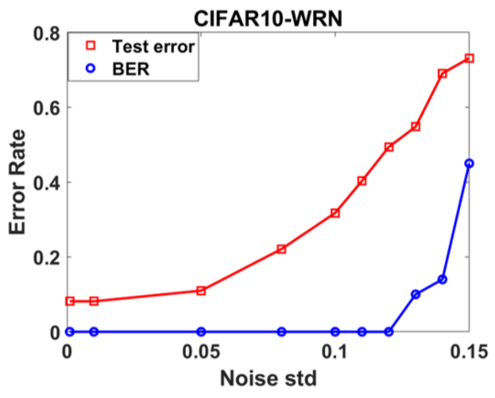
(b)

Figure 3.5. DeepMarks' robustness against FP overwriting at a different (red color) or the same layer (blue color).

(IV) Model Fine-tuning. Recall that the fine-tuning attack is implemented by re-training the fingerprinted model on the user's new dataset using only the conventional cross-entropy loss. We simulate this process by adding random Gaussian noise with zero mean and different standard deviations (std) to the weights of the marked DNN and extract the code-vector from the noisy weights. Figure 3.6 shows the test error and BER of FP detection after injecting noise on the fingerprinted model. Our key observation is that test error is more sensitive to noise compared to BER, thus the malicious user cannot remove the FP while preserving the model's performance.



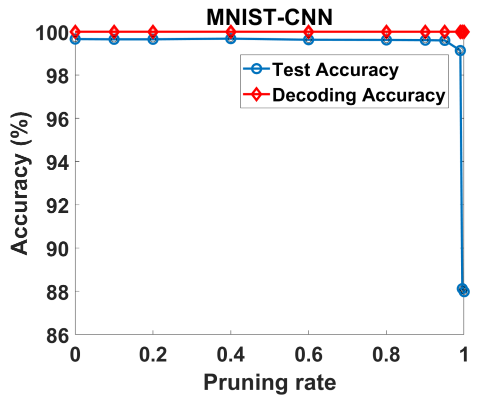
(a)



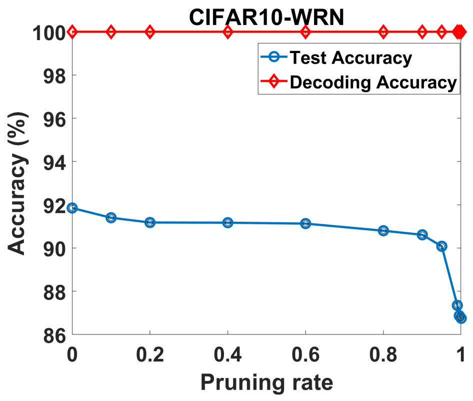
(b)

Figure 3.6. DeepMarks' robustness against model fine-tuning. Adding excessive noise incurs large increase of test error while the embedded FP might be removed.

(III) Parameter Pruning. To prune the target layer, we use the pruning method in [HPTD15] and set $\alpha\%$ of the weights that possess the smallest absolute values to zero. The obtained mask is then used to sparsely fine-tune the fingerprinted model on the training data with the conventional cross-entropy loss to compensate for the accuracy drop induced by pruning. We first assess the code-vector extraction (decoding) accuracy for individual users under different pruning rates and show results in Figure 3.7. One can see that increasing the pruning rate leads to a drop in the test accuracy, while the code-vector can always be decoded with 100% accuracy. The perfect FP



(a)



(b)

Figure 3.7. Code-vector extraction accuracy (red color) and test accuracy (blue color) for MNIST-CNN (a) and CIFAR10-WRN (b) benchmark under different pruning rates.

decoding suggests that: (i) DeepMarks is robust against pruning attacks and reliably identifies the queried user; (ii) DeepMarks has no false alarms and satisfies the integrity criteria.

We further assess the robustness of colluder detection against parameter pruning. Figures 3.8 shows the detection rates and false alarm rates of DeepMarks under three different pruning rates. Comparing Figure 3.8 with Figure 3.4, we can see that DeepMarks is robust and tolerates up to 99% parameter pruning for both MNIST and CIFAR10 benchmarks.

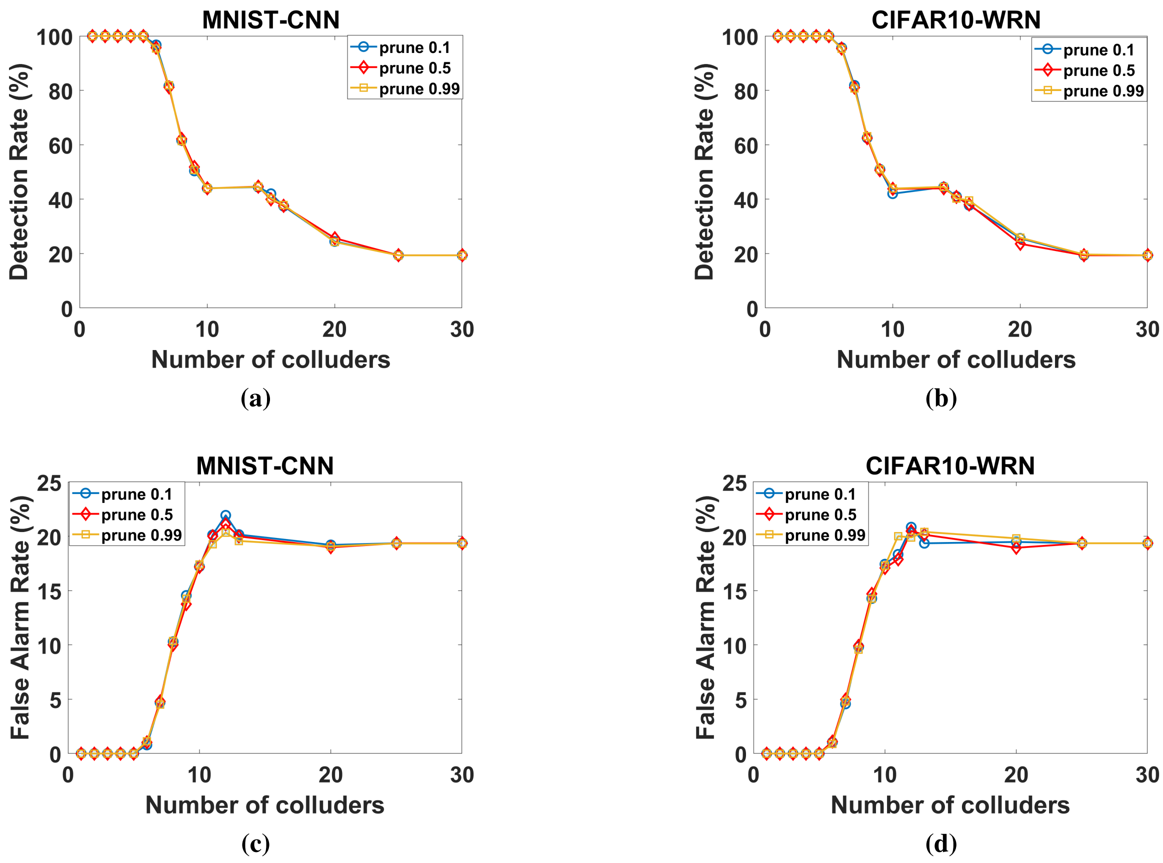


Figure 3.8. DeepMarks' robustness of colluders identification against parameter pruning attack. Detection rate (a, b) and false alarm rate (c, d) of DeepMarks framework are not affected by a wide range of pruning rates.

In summary, DeepMarks achieves high FP detection rates and low false alarm rates when confronted with various FP deconstruction and model modification attacks. Therefore, DeepMarks satisfies the **robustness, reliability** and **integrity** criteria in Table 3.1. The performance consistency across various benchmarks corroborates the **generality** of our framework.

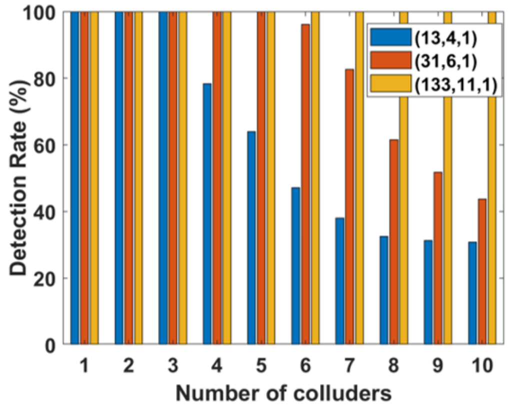
Scalability

DeepMarks is applicable to large-scale distribution systems. We define scalability of a fingerprinting technique as the number of supported users per code bit: $\beta = \frac{n}{v}$. For a $(v,k,1)$ codebook, the maximum number of users is determined by the code-vector length v and the block size k by $n = \frac{v(v-1)}{k(k-1)}$. Thus, the scalability metric can be computed as follows:

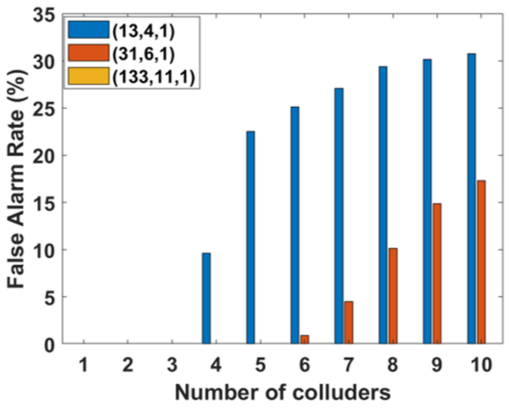
$$\beta = \frac{v-1}{k(k-1)}. \tag{3.8}$$

It is straightforward to see that longer FPs provide better scalability for a fixed block size. Equation (3.8) also shows the trade-off between the length of the code-vector v and the collusion resilience level $(k-1)$. When the scalability is fixed, a higher resistance level requires longer fingerprinting codes. Systematic approaches to construct various BIBDs have been developed [CD06], providing a vast supply of ACCs for DeepMarks.

We assess DeepMarks' performance with three different codebooks $(13,4,1)$, $(31,6,1)$, $(133,11,1)$ BIBD, and illustrate the comparison results in Figure 3.9. **DeepMarks provides various levels of user capacity and collusion resistance by allowing the owner to specify**



(a)



(b)

Figure 3.9. Effect of codebook design. DeepMarks is scalable and provides various levels of detection performance.

codebook parameters. Particularly, a larger codebook (e.g., $(133,11,1)$-BIBD ACC) accommodates more customers in the distribution system and yields better detection metrics. As such, DeepMarks framework can be customized to provide guaranteed performance based on the requirements of the IP owner.

Efficiency

DeepMarks fingerprinting framework incurs negligible overhead and is highly efficient. We define the efficiency of embedding and extracting an fingerprint as the normalized runtime overhead of retraining the target DNN and recovering the code-vector from the weights, respectively. To quantitatively evaluate DeepMarks' overhead as discussed in Section 3.4.3, we measure the ratio of the FP embedding time to the original DNN training time, and the ratio of the code-vector extraction time to the DNN prediction time. Table 3.5 summarizes the results of DeepMarks' normalized runtime, suggesting our efficiency.

Table 3.5. Efficiency evaluation of DeepMarks' FP embedding and extraction in terms of normalized runtime overhead.

Normalized Runtime Overhead (%)	FP Embedding	FP Extraction
MNIST-CNN	5.214	0.006
CIFAR10-WRN	2.562	0..056

3.6.2 Comparison with the State-of-the-art

In this section, we compare DeepMarks with the state-of-the-art DNN watermarking method in literature. The work of [UNSS17, NUSS18] proposed a white-box digital watermarking technique for DL models using constraint-based watermarking. The authors evaluate their approach on CIFAR10-WRN benchmark and show that the embedded watermark tolerates up to 65% parameter pruning. For fair comparison, we also assess the performance of DeepMarks on CIFAR10-WRN benchmark and encode the FP information in the same layer as reported in the paper [UNSS17]. Compared to the prior watermarking method, **DeepMarks is more robust against parameter pruning attack** since the embedded fingerprint remains even after 99%

parameters are removed (shown in Figure 3.7b).

We further demonstrate the superior collusion resilience of DeepMarks compared to the fingerprinting scheme that employs multiple distinct watermarks constructed by [UNSS17]. In our experiments, we assume there are 31 customers in the model distribution system and use the open-source code [Uch17] to implement WM signature generation, WM embedding, and WM extraction. The $(31,6,1)$-BIBD ACC codebook is selected for DeepMarks. Table 3.6 summarizes the comparison results. The test accuracy of the original unmarked CIFAR10-WRN and the marked one are shown in Column 2 and Column 3, respectively. The BER of WM/FP extraction is shown in Column 4, indicating that both methods can retrieve the digital marker correctly for authenticating model authorship.

Table 3.6. Performance comparison between DeepMarks and the state-of-the-art DNN watermarking technique [UNSS17].

Method	Baseline Accuracy	Accuracy with WM/FP	Average BER	Colluded Accuracy	Colluders Detection Rate
Uchida et.al [19]	91.85%	92.15%	0	92.18%	**3.84%**
Ours	91.85%	92.03%	0	92.14%	**100%**

To assess the robustness of these two marking methods against FP averaging attack, we assume that the first three users collude and average the weights in the embedded layer. The result $\widetilde{\mathbf{w}}_{\mathbf{avg}} = \frac{1}{3}(\widetilde{\mathbf{w}}_{\mathbf{1}} + \widetilde{\mathbf{w}}_{\mathbf{2}} + \widetilde{\mathbf{w}}_{\mathbf{3}})$ is used as the response to the owner's query. In our experiment, we assume user 1 produces the colluded DNN by replacing the weights in the marked layer $\widetilde{\mathbf{w}}_{\mathbf{1}}$ with $\widetilde{\mathbf{w}}_{\mathbf{avg}}$ while keeping the weights in the other layers unchanged. The test accuracy of the resulting colluded model is shown in Column 5 of Table 3.6, which is comparable to the baseline and makes the collusion attack effective. Note that the DNN watermarking method in [UNSS17] is unaware of potential collusion attacks and does not propose any collusion detection approach. We implement colluder identification in the watermarking setting by randomly selection from all feasible colluder sets that satisfy the constraint obtained from the colluded watermark. We compare the detection rate of the collusion attack in the last column of Table 3.6.

In summary, DeepMarks outperforms the fingerprinting extension built on the state-of-the-art DNN watermarking technique [UNSS17] by achieving 100% detection accuracy, which is significantly higher than 3.84%.

3.7 Summary

DNNs are facilitating breakthroughs in various fields and are increasingly commercialized. Systematic IP protection and digital right management for pre-trained, ready-to-deploy models has been a standing challenge. We take the first step to tackle this problem by proposing DeepMarks, an efficient, end-to-end framework that is functionality-preserving and enables coherent fingerprint insertion in the distribution of weights within the target DNN. We introduce a comprehensive set of requirements for DNN fingerprinting and empirically corroborate that DeepMarks respect all criteria. Evaluation results show that DeepMarks yields high detection rates and low false alarm rates for model ownership proof and user tracing. Furthermore, DeepMarks is the first framework that is *provably collusion-secure* in a large-scale model distribution system and is robust against various attacks. Our technique can be seamlessly integrated within existing DL frameworks (e.g., TensorFlow, PyTorch, Theano), thus paving the way for model designers to achieve reliable technology transfer. Future research directions include developing advanced codebook construction schemes to further improve collusion resistance, and investigating collaborative fingerprint embedding among multiple users to improve efficiency.

3.8 Acknowledgements

This chapter, in part, have been published at the Proceedings of 2019 International Conference on Multimedia Retrieval (ICMR) and appeared as: Huili Chen, Bita Darvish Rouhani, Cheng Fu, Jishen Zhao, Farinaz Koushanfar, “DeepMarks: A Secure Fingerprinting Framework for Digital Rights Management of Deep Learning Models” [CRF+19]. The dissertation author was the primary author of this conference paper.

Chapter 4

DeepAttest: End-to-End Attestation of Deep Neural Networks

Emerging hardware architectures for deep neural networks are increasingly commercialized and shall be considered as the hardware-level *Intellectual Property* (IP) of the device providers. However, these intelligent devices might be abused and such vulnerability has not been identified. The unregulated usage of intelligent platforms and the lack of hardware-bounded IP protection impair the commercial advantage of the device provider and prohibit reliable technology transfer. This chapter is motivated to design a systematic methodology that provides *hardware-level IP protection* and *usage control* for DNN applications on various platforms.

To address the hardware IP concern, we propose *DeepAttest*, the first on-device DNN attestation technique that certifies the legitimacy of the DNN program mapped to the device of interest. DeepAttest works by designing a device-specific *fingerprint* which is encoded in the weights of the DNN deployed on the target platform. The embedded fingerprint (FP) is later extracted with the support of the Trusted Execution Environment (TEE). The existence of the pre-defined FP is used as the attestation criterion to determine whether the queried DNN is legitimate. Our attestation framework ensures that only authorized DNN programs yield the matching FP and are allowed for inference on the target device. DeepAttest provisions the device provider with a practical solution to limiting the application usage of his/her manufactured hardware and prevents unauthorized or tampered DNNs from execution.

We take an Algorithm/Software/Hardware co-design approach to optimize DeepAttest's overhead in terms of latency and energy consumption. To facilitate the deployment, we provide a high-level API of DeepAttest that can be seamlessly integrated into existing deep learning frameworks and TEEs for hardware-level IP protection and usage control. Extensive experiments corroborate the fidelity, reliability, security, and efficiency of DeepAttest on various DNN benchmarks and TEE-supported platforms.

4.1 Introduction

Deep Neural Networks (DNNs) are increasingly adopted in various fields ranging from biomedical diagnosis and nuclear engineering to computer vision and natural language processing due to their unprecedented performance [CW08, FDFC$^+$18]. Methodological and architecture-level advancements have been proposed to improve the performance and efficiency of DNN training/execution on diverse platforms [HMD15, CES16, SPS$^+$18]. While the distribution of intelligent devices facilitates DNNs' deployment in the real world, IP concerns may arise in the supply chain. The customers might misuse the device for illegal/unauthorized DNN applications. In this work, we are motivated to provide *hardware-level IP protection* and *usage control* via on-device DNN attestation to protect the commercial advantages of the device providers.

Prior works have identified the IP concern when deploying contemporary Deep Learning (DL) models. Various DNN watermarking techniques have been proposed to prevent copyright infringement of software-level neural IP (consisting of the topology, parameters, and configuration of the DNN) [UNSS17, DRCK19, CRF$^+$19]. The watermark is embedded in the distribution of model weights/activations [UNSS17, DRCK19], or the decision boundary [ABC$^+$18, CRK19]. Existing DNN watermarking techniques provide model ownership proof at the *software/functionality* level while the authentication overhead and the potential misuse of the underlying computing platform are not taken into account. Developing an efficient and effective on-device DNN attestation methodology is challenging since the attestation scheme is required to: (i) Preserve the performance (e.g., accuracy) of the deployed DNN; (ii) Provide reliable and secure attestation decision; (iii) Incur low latency and power consumption to ensure its applicability in real-time DNN applications and resource-constrained systems.

We develop an *end-to-end, on-device attestation* framework called *DeepAttest* to address the above challenges. DeepAttest, for the first time, extends IP protection to the hardware/device-level by taking advantage of the Trusted Execution Environment (TEE). Figure 4.1 illustrates the usage of our framework. DeepAttest takes the pre-trained model and security parameters from the device provider as its inputs, thus can provide a trade-off between security level and attestation overhead. A set of verifiable, functionality-preserved DNNs that carry the device-specific fingerprint are returned as the outputs. Only the 'marked' models can pass our customized attestation and are allowed to run inference on the pertinent device. DeepAttest effectively detects malicious modifications and prevents unauthorized models from execution.

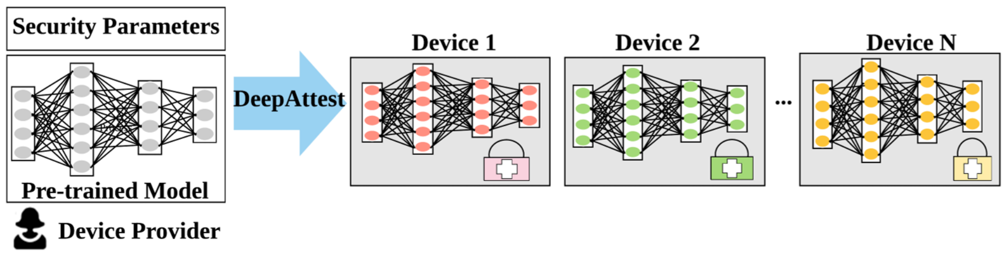


Figure 4.1. DeepAttest provides device-level IP protection and usage control for DNN applications.

DeepAttest framework consists of two key phases: (i) **Off-line marking stage**: DeepAttest generates a device-specific fingerprint associated with each target hardware for the device provider, and embeds it in the probabilistic distribution of the selected weighted within the deployed DNN. (ii) **Online attestation stage**: DeepAttest designs a hybrid trigger to control the activation of DNN attestation, thus can detect both static and dynamic data tampering. When the attestation is triggered, DeepAttest securely extracts the fingerprint (FP) from the deployed DL model with TEE's support and compares it with the true value stored in the secure memory. The queried DNN is determined to be legitimate and permitted for normal inference if it yields a matching FP. Otherwise, the DNN program fails the attestation and its execution is aborted.

By introducing DeepAttest, this work makes the following contributions:

- **Enabling effective on-device attestation for DNN applications.** The proposed end-to-end attestation framework is capable of verifying the legitimacy of an unknown DNN with high reliability (preventing unauthenticated DNNs from execution) and high integrity (allowing legitimate DNNs to run normal inference).

- **Characterizing the criteria for a practical attestation scheme in the domain of deep learning.** We introduce a comprehensive set of metrics to profile the performance of pending DNN attestation techniques. The introduced metrics allow DeepAttest to provide a trade-off between security level and attestation overhead.

- **Leveraging an Algorithm/Software/Hardware co-design approach to develop an efficient attestation solution.** Our device-aware framework is equipped with careful design optimization to ensure the minimal overhead and enhanced security of attestation. As such, our solution provides a lightweight on-device DNN attestation scheme that is applicable to resource-constrained platforms.

- **Investigating DeepAttest's performance on various DNN benchmarks and TEE-supported platforms.** We perform extensive experiments on DNNs with different topologies using TEE-supported CPU (Intel SGX) and GPU (via simulation) platforms.

DeepAttest opens a new axis for the growing research in secure DL. Our approach is orthogonal to existing secure DL methods that aim to verify the correctness of DNN execution or preserve the privacy of sensitive data. DeepAttest paves the way for on-device attestation and platform-aware usage control for DNN applications.

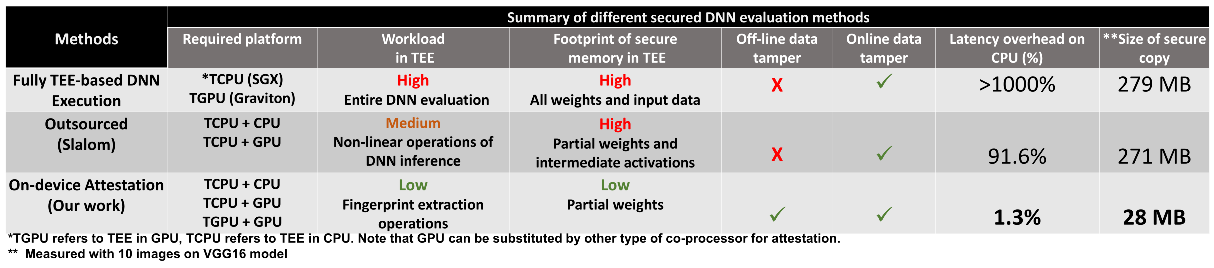
4.2 Related Works

4.2.1 Secure DNN Evaluation on Hardware

TEE Protection Mechanism. Modern CPU hardware architectures provide TEEs to ensure secure execution of confidential applications using program isolation. Intel SGX [XSLH16], ARM TrustZone [LIM09] and Sanctum [CLD+17] are examples of TEEs. TEEs are called *enclaves* in SGX. To prevent malicious programs from interfering executions in TEE, data is encrypted by Memory Encryption Engine (MEE) before it is put into the Enclave Page Cache (EPC) located in the Processor Reserved Memory (PRM). We refer to this process as *secure memory copy*. Programs inside the TEE can read or write data outside of the TEE, while programs outside of the TEE is not allowed to access the EPC. TEEs on other platforms utilize similar mechanisms to isolate the execution of the protected program by securing memory access to the code and data of the confidential program. Besides the CPU-level TEE support, Graviton [VVB18] proposes a new GPU architecture design that provides the TEE capability.

Comparison between Secure DNN Techniques. Figure 4.2 illustrates the comparison between the state-of-the-art secure DNN techniques and DeepAttest in terms of platform requirement, incurred workload in TEE, resistance to off-line/online data tampering, and capability of verifying DNNs' inference results. A quantitative overhead comparison is given in the last two columns. Detailed explanations about the overhead are given in Section 4.7.6. Existing secure DNN inference can be divided into two categories: full execution inside the TEE, and outsourcing partial computations from the TEE to untrusted environments. *DeepAttest identifies a new security dimension named 'device-level' IP protection and usage control.* Note that DeepAttest is orthogonal to the techniques that provide verifiable results and privacy-preserving property. We discuss the limitation of the contemporary secure DNN techniques below.

▪ **Fully TEE-based DNN Evaluation.** A naive way to ensure trusted DNN inference is to run all computations within the TEE [GTS+18, LLP+19]. However, such an approach incurs a prohibitive overhead due to: (i) Limited PRM size in TEEs for execution. For instance,



Methods	Summary of different secured DNN evaluation methods						
	Required platform	Workload in TEE	Footprint of secure memory in TEE	Off-line data tamper	Online data tamper	Latency overhead on CPU (%)	**Size of secure copy
Fully TEE-based DNN Execution	*TCPU (SGX) TGPU (Graviton)	High Entire DNN evaluation	High All weights and input data	X	✓	>1000%	279 MB
Outsourced (Slalom)	TCPU + CPU TCPU + GPU	Medium Non-linear operations of DNN inference	High Partial weights and intermediate activations	X	✓	91.6%	271 MB
On-device Attestation (Our work)	TCPU + CPU TCPU + GPU TGPU + GPU	Low Fingerprint extraction operations	Low Partial weights	✓	✓	**1.3%**	**28 MB**

*TGPU refers to TEE in GPU, TCPU refers to TEE in CPU. Note that GPU can be substituted by other type of co-processor for attestation.
** Measured with 10 images on VGG16 model

Figure 4.2. Comparison of existing secure DNN techniques and DeepAttest.

the enclave memory size of Intel SGX is 128MB, which is much smaller than the parameter size of a contemporary DL model (e.g., 189MB for ResNet-101). As such, the weights need to be reloaded for different inputs. (ii) Encryption and decryption by MEE. Data that is communicated with the secure memory needs to be encrypted/decrypted; (iii) Additional hardware behaviors due to CPU‘s context switch [Har17]. Operations such as flushing Translation Lookaside Buffer (TLB) [ACH$^+$10] and out-of-order execution pipeline are necessary for security consideration [CLD$^+$17], which incurs extra overhead.

▪ **Outsource-based Secure DNN Inference.** Slalom [TB18] is a framework for secure DNN execution on trusted hardware that guarantees integrity. It partitions DNN computations into non-linear and linear operations. These two parts are then assigned to the TEE and the untrusted environment for execution, respectively. Freivalds' algorithm [MR95] is used to verify the integrity of linear computations performed on the untrusted GPU. Slalom reduces the overhead compared to fully TEE-based inference. However, the main disadvantage of Slalom is that it need to transfer intermediate results into the TEE to complete DNN forward propagation, thus incurring large communication overhead.

▪ **TEE-based Attestation (Our Work).** Unlike the previous methods, DeepAttest is the first framework that can prevent both off-line and online data tampering. More specifically, Slalom [TB18] and fully TEE-based DNN evaluation [GTS$^+$18, LLP$^+$19] can guarantee result integrity and protect the weight parameters from online data tamper. However, these two methods are vulnerable to off-line data tamper (e.g., fault injection) where the attacker modifies the data stored in the untrusted memory before it is used in the secure DNN inference.

4.2.2 DNN Watermarking

A line of research has focused on addressing the soft-IP concern of DL models using digital watermarking [UNSS17, ABC+18, DRCK19, LMPT20]. The authors of [UNSS17] encode the watermark (WM) in the transformation of model weights by adding constraints to the original objective function. The works [ABC+18, LMPT20] extend DNN watermarking to remote cloud service. Particularly, they design specific image-label pairs as the watermark set and embed the WM in the model's decision boundary. DeepSigns [DRCK19] presents the first data-aware watermarking approach by embedding the WM in the dynamic activation maps.

All of the above-mentioned DNN watermarking techniques focus on *software-level model authorship proof*. Note that a naive implementation of DNN watermarking on the hardware is inadequate to provide an efficient and trustworthy attestation solution due to the unawareness of resource management and potential attacks. As such, these methods are not suitable for hardware-level IP protection. The works [ABC+18, LMPT20] require DNN inference of multiple inputs on the local device and TEE-supported WM checking, which is prohibitively costly. Compared to weight-based watermarking [UNSS17], DeepAttest's fingerprint extraction from the DL model involves fewer computations since no extra sigmoid function is required. In this work, we develop an efficient on-device attestation scheme that ensures the legitimacy of the deployed DNN with negligible overhead.

4.2.3 Trusted Execution Environment

Previous research [BW12, TMLL06] has paved the path for secure isolated execution on general-purpose processors. Intel SGX [XSLH16] is the most widely used TEE with a user interface. The vulnerabilities of SGX to potential attacks such as spectre[KHF+19], cache[GESM17] or other side-channel attacks are later identified. Besides TEE for CPU platforms, a growing amount of research has been done to provide TEE for other hardware platforms. For instance, Ty-

Tan [BEMS+15] and TrustLite [KSSV14] are TEEs proposed for embedded systems. DeepAttest is generic and can be extended to these computing platforms with TEE support.

4.2.4 Privacy-Preserving DNN

Beyond integrity violation and data tampering, privacy is another critical concern in the DL domain. Various techniques have been suggested for Privacy-Preserving Machine Learning (PPML) [RWT+18, HRGK18, HK19, RSC+19]. CryptoNet [GBDL+16] leverages Homomorphic Encryption (HE) to achieve PP-inference with prohibitive latency due to extensive computations. Gazelle [JVC18] accelerates the Levelled HE-based DL inference by exploring SIMD optimization. Garbled Circuit (GC) is an alternative approach [SHS+15, RRK18] for PPML. Compared to HE, GC-based PPML schemes have a smaller computation overhead but require more communication [HTGW18, ARC19].

4.3 Motivation

Prior works have focused on model ownership proof using software-level DNN watermarking [UNSS17, ABC+18, DRCK19]. Existing watermarking techniques are *oblivious* of the computing platform and verification overhead, thus the security and efficiency of their execution on the hardware are not guaranteed (detailed in Section 4.2). DeepAttest is motivated to address the above deficiencies. We provide an attestation-based IP protection technique that is bounded to hardware and restrict device usage for DNN applications. We identify three challenges of developing a practical on-device DNN attestation method and detail each one below.
(C1) Functionality-preserving. The attestation scheme shall not degrade the performance of the original DL model. Since authenticated DNNs are allowed for normal inference, their functionality (e.g., accuracy) shall be preserved to provide the desired service.
(C2) Security and Reliability. On-device attestation shall be secure and yield reliable decisions under a strong threat model. Note that the attack surface of device-level attestation is larger than the one of software-level attestation.

(C3) Low Overhead. The attestation protocol shall incur negligible overhead to ensure its applicability in real-time data applications and resource-constrained systems.

The constraint *C1* imposes the algorithm/software-level challenge on the design. Challenges *C2* and *C3* need to be resolved from algorithm/software/hardware all three levels. We explicitly develop systematic design principles to tackle the identified challenges C1-C3 as detailed in Section 4.4.

4.4 DeepAttest Overview

DeepAttest is the first DNN attestation framework for device IP protection & usage control and is applicable to any computing platform with TEE support. Figure 4.3 illustrates the global flow of DeepAttest. In the off-line marking stage, the device manufacturer obtains the secret FP keys and a set of marked DL models. The FP keys are then stored in the secure memory of the TEE on the target device. The user is required to purchase the marked DNN from the device provider to pass the online attestation and execute normal inference. Deployment of unauthorized DNN programs and malicious fault injection will be detected. The target *device* can be used with a co-processor (e.g., ASIC, FPGA), in which case usage control can be extended to it. On-device attestation can be performed on the co-processor if it has TEE support.

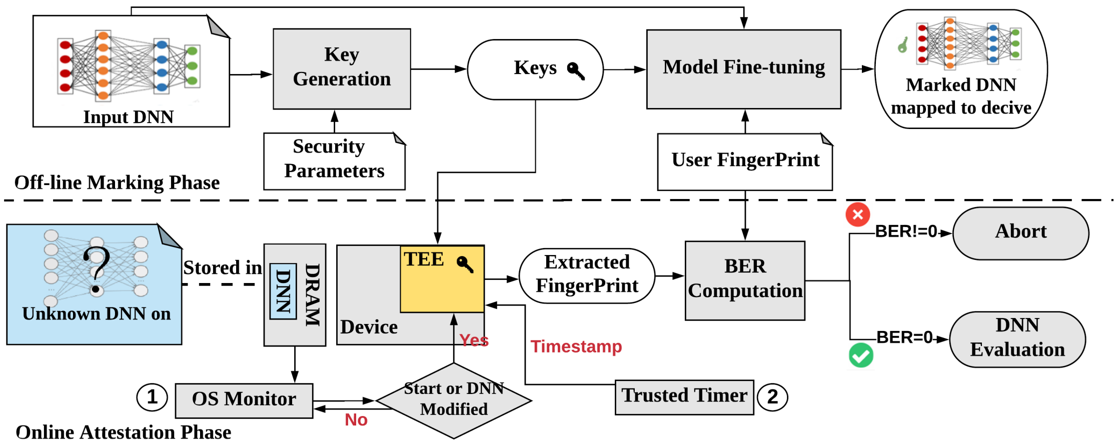


Figure 4.3. DeepAttest's global flow for on-device DNN attestation.

In the following of this section, we present the *Design Principles* (DPs) of DeepAttest to address the corresponding challenges in Section 4.3.

(DP1) Regularization-based DNN Fingerprinting. To preserve the functionality of the pertinent DNN program (C1), DeepAttest explores the over-parameterization of high dimensional DNNs and utilizes regularization to encode the device-specific FP in the DL model. Regularization [BNS06, SS01] is a common approach to alleviate model over-fitting [SHK+14, RSJ+18]. We detail the two key phases of DeepAttest's algorithm/software design below.

▪ **Off-line DNN Marking.** DeepAttest takes the pre-trained DL model and the owner-defined[1] security parameters as its inputs. DeepAttest then outputs the FP secret keys along with the corresponding set of marked DNNs that are ready-to-be-deployed on the target device. Note that FP embedding is a one-time task performed by the owner before the authorized models are deployed on the target device. Furthermore, the secret FP keys stored in the secure memory can be updated after device distribution. This is feasible since current TEEs typically support remote attestation (RA) that allows secure memory update of the TEE. Details about off-line DNN marking are given in Section 4.5.1.

▪ **Online DNN Attestation.** DeepAttest utilizes a hybrid triggering scheme where a TEE-based attestation process is instantiated when the static or the dynamic trigger is activated. During the attestation phase, the marked weights data that carries the FP is copied to the secure memory inside the TEE. The FP is then extracted from the weights within the TEE. Finally, the Bit Error Rate (BER) between the recovered FP and the ground-truth one is computed. The verified DNN with zero BER is allowed to run normal inference. Illegitimate DL models with non-zero BERs are aborted. Details about the attestation protocol are discussed in Section 4.5.2.

(DP2) TEE-based Attestation. Hardware-bounded attestation has larger attack surface compared to the one in software-level [MGDC+18]. The program might be corrupted by the adversary in an untrusted execution environment. As such, the computation involved in attestation shall

[1]We use owner and device provider interchangeably in the paper.

be performed securely. DeepAttest utilizes TEE-supported trusted hardware to guarantee the security and reliability of the attestation result (addressing C2, detailed in Section 4.5.2).

(DP3) Algorithm/Software/Hardware Co-design. We present multiple design optimization techniques to enhance the efficiency and security of DeepAttest. As a result, our framework is applicable to real-time data applications and resource-constrained systems. DeepAttest's hardware optimization includes: (i) Data pipeline that hides the majority of the TEE latency during attestation; (ii) Early termination that avoids unnecessary computation; (iii) Shuffled data storage that provides stronger security against fault injection. These optimization address challenge C3 as detailed in Section 4.6.

4.4.1 DNN Attestation Metrics

We introduce a comprehensive set of criteria to evaluate the performance of a DNN attestation technique. Table 4.1 details the criteria for an effective DNN attestation methodology. *Fidelity* requires that the functionality (e.g., accuracy) of the pre-trained model shall not be degraded after the off-line DNN marking. *Reliability* and *integrity* means that the attestation approach shall prevent unauthorized DNNs from executing (low false alarm rate of FP detection) and allow normal inference of legitimate DNNs (high detection rate of the embedded FP), respectively. A reliable attestation method is also desired to satisfy the *security* requirement such that the attestation decision is trustworthy. *Efficiency* requires that the overhead (i.e., latency, power consumption) incurred by attestation shall be negligible. *Scalability* and *generalizability* ensure that the attestation method can be applied to DNNs of various size and diverse TEE-supported hardware devices, respectively. DeepAttest satisfies all the requirements listed in Table 4.1 as shown in Section 4.7.

4.4.2 Assumptions and Threat Model

DeepAttest's Assumptions. We aim to design a robust attestation scheme that yields reliable decisions in various situations. More specifically, we consider the following three adversarial

Table 4.1. Requirements for an effective and practical on-device attestation technique for deep neural networks.

Requirements	Description
Fidelity	Functionality of the deployed DNN shall not be degrade as a result of FP embedding in the marking stage.
Reliability	Online attestation shall be able to prevent unauthorized DNN programs (including full-DNN program substitution and malicious fault injection) from executing on the specific device.
Integrity	Legitimate DNN programs shall yield the matching FP with high probability and run normal evaluation.
Efficiency	The online attestation shall yield negligible overhead in terms of latency and energy consumption.
Security	The attestation method shall be secure against potential attacks including fault injection and FP forgery.
Scalability	The attestation technique shall be able to verify DNNs of varying sizes.
Generalizability	The DNN attestation framework shall be compatible with various computing platforms.

levels: (i) Operating System (OS) is trusted and can lock the pages allocated for the weight data. In this case, the weights in the main memory will not be tampered or evicted. (ii) OS cannot lock the memory but is able to provide information about the pages associated with the weights (e.g., page fault or page modified); (iii) Hardware provides a trusted timestamp while the OS might be corrupted (thus does not satisfy the requirements in (i) or (ii)). DeepAttest tackles with the above scenarios by designing a hybrid trigger scheme using two sources, i.e., the OS and the secure timer, as detailed in Section 4.5.2.

Threat Model. The adversary might try to bypass on-device attestation for gaining illegal profits. We detail three potential attacks below and demonstrate the experimental results of DeepAttest's security in Section 4.7.

(i) Full-DNN Program Substitution. Untrusted users may attempt to misuse the distributed device by mapping illegitimate DL model to it. In this case, the adversary is assumed to know the physical address of the deployed DNN (e.g., by eavesdropping) and substitutes the original content with his target unauthorized DNN. DeepAttest is motivated to address the susceptibility of the intelligent device to such attacks.

(ii) Fingerprint Forgery Attack. In order to successfully pass the attestation, the attacker might attempt to forge the device-specific signature stored in the secure memory inside the TEE. More specifically, the adversary may use brute-force searching and try to find the exact secret key used in FP embedding to reconstruct the device's fingerprint and yield zero BER.

(iii) Fault Injection. Besides the program-level replacement, the attacker may also conduct more fine-grained memory content modification attacks. We assume a strong attack model where the adversary might know the memory allocation on the target device and randomly selects memory blocks for malicious purpose (e.g., malware, pirating sensitive information). Such local memory modification is stealthier than the DNN-program substitution discussed above and poses potential threats to the intelligent device.

4.5 DeepAttest Design

DL models typically feature non-convex loss functions with many local minima that are likely to yield similar accuracy [CHM$^+$15]. DeepAttest takes advantage of the non-uniqueness of non-convex problems to embed the device's FP in the distribution of the selected weights. The embedded FP is later extracted in the attestation phase as the identifier to determine the legitimacy of the DNN. We use Convolution Neural Networks (CNNs) and Residual Networks to illustrate DeepAttest's workflow. Note that DeepAttest is generic and can be applied to other network architectures. We detail the two key stages discussed in Section 4.4 below.

4.5.1 Off-line DNN Marking

Algorithm 1 outlines the steps involved in DeepAttest's off-line DNN marking (i.e., FP embedding) for one intermediate layer. The extension to multi-layer fingerprinting is straightforward. DeepAttest's DNN marking consists of the following three steps:

❶ **Key Generation.** Besides the position of the target layer that carries the FP, DeepAttest's FP keys consist of three components: a codebook C, an orthogonal basis matrix U, and a projection matrix X. We explain the design of each component as follows:

(i) Devices Codebook: Given the code length v and the maximal number of supported users b

ALGORITHM 1: Fingerprint embedding for one hidden layer.

```
INPUT: Pre-trained unmarked DNN ($\mathcal{T}$); Training data
    ($\{X^{train}, Y^{train}\}$); Location of the target layer ($l$) with
    embedding dimension ($N$); Code length ($v$); Resilience
    level ($k$); Embedding strength ($\gamma$).
OUTPUT: A set of marked DNNs ($\{\mathcal{T}_1^*, ..., \mathcal{T}_b^*\}$); FP keys.
1 Key Generation:
        $C_{v\times b} \leftarrow Construct\_Codebook\ (v, k, 1)$
        $U_{v\times v} \leftarrow Generate\_Basis\_Matrix\ (v)$
        $X_{v\times N} \leftarrow Generate\_Projection\_Matrix\ (v, N)$
2 Fingerprint Construction:
        $F_{v\times b} \leftarrow Construct\_Fingerprints\ (C, U)$
3 Model Fine-tuning: For each user $j$ ($j = 1, ..., b$), train the DNN
  on $\{X^{train}, Y^{train}\}$ with the corresponding FP-specific loss:
      $\mathcal{L} = \mathcal{L}_0 + \gamma \cdot Mean\_Square\_Error(\mathbf{f_j} - X\mathbf{w_j})$.
Return: Marked DNNs ($\{\mathcal{T}_1^*, ..., \mathcal{T}_b^*\}$), FP keys
    ($l$, $C_{v\times b}$, $U_{v\times v}$, $X_{v\times N}$).
```

specified by the owner, DeepAttest generates a codebook $C \in \mathbb{B}^{v\times b}$ for the device provider. The codebook is randomly generated where each column of C is a unique code-vector associated with a specific device. The code-vector is stored in the secure memory within the TEE on the target hardware.

(ii) Orthogonal Basis Matrix: The orthogonal matrix $U_{v\times v}$ is generated from element-wise Gaussian distribution for security consideration [WTWL04]. The columns of U are used as the basis vectors for FP construction in step 2.

(iii) Projection Matrix: The owner's secret projection matrix $X_{v\times N}$ is generated from standard normal distribution $\mathcal{N}(0,1)$ where N is the embedding dimension of the target layer. We explicitly illustrate the design of the FP carrier for convolutional (conv) layers and fully-connected (FC) layers below:

▪ **Convolutional Layer.** The weight matrix of a conv layer is a 4D tensor $W \in \mathbb{R}^{D\times D\times F\times H}$ where D is the kernel size, F and H is the number of input and output channels, respectively. We average the weight W over the output channel dimension and stretch the result to a vector $\mathbf{w}$. The FP is embedded in the projection of the vector $\mathbf{w} \in \mathbb{R}^N$ (detailed in step 3) where the embedding dimension is $N = D \times D \times F$.

■ **Fully-Connected Layer.** The weights of the FC layer is a 2D matrix $W_{F\times H}$ where F is the input dimension and H is the number of units. Similar to the processing for conv layers, we average W over the last dimension and use the resulting vector $\mathbf{w} \in \mathbb{R}^N$ to carry the FP. Note that the embedding dimension $N = F$ here.

❷ **Fingerprint Construction.** DeepAttest constructs code modulated FPs as follows. Given the codebook $C_{v\times b}$ obtained in step 1, the coefficient matrix $B_{v\times b}$ for FPs is computed from the linear mapping $b_{ij} = 2c_{ij} - 1$ where $c_{ij} \in \{0,1\}$. The FP of the j^{th} user is crafted as the linear combination of basis vectors in U (obtained in step 1) with $\mathbf{b_j} \in \{\pm 1\}^v$ as the coefficient vector:

$$\mathbf{f_j} = \sum_{i=1}^{v} b_{ij}\mathbf{u_i}, \tag{4.1}$$

❸ **Model Fine-tuning.** The FP designed from Equation (4.1) is embedded in the weight parameters of the selected layer in the pre-trained model by incorporating the FP-specific embedding loss to the conventional loss function ($\mathscr{L}_0$):

$$\mathscr{L} = \mathscr{L}_0 + \gamma \cdot Mean\_Square\_Error(\mathbf{f} - X\mathbf{w}). \tag{4.2}$$

Here, γ is the embedding strength that controls the contribution of the additive FP embedding loss $\mathscr{L}_{FP} = Mean\_Sqaure\_Error(\mathbf{f} - X\mathbf{w})$. The vector $\mathbf{w}$ is the flattened averaged weights of the target layers that carry the FP information. DeepAttest minimizes the FP-specific loss $\mathscr{L}_{FP}$ together with the conventional loss during DNN training to enforce the FP constraint in the distribution of weights in the selected layer.

4.5.2 Online DNN Attestation

The secret FP keys generated in the off-line marking stage are stored in the secure memory inside the TEE. The returned DL models that carry the device-specific FPs are deployed on the target platform and stored in the untrusted memory for later execution. Algorithm 2 outlines the two main steps in the online attestation stage. It takes the FP keys and the weights in the marked layers as the inputs. The BER between the extracted FP from the queried weights and the ground-truth FP (included in FP keys) is returned as the output. We detail each step below.

1 **Hybrid Attestation Trigger.** DeepAttest leverages a *hybrid trigger* mechanism for activating DNN attestation as shown in Figure 4.3. A *static* trigger signal is generated when the OS detects that a DNN program requests to start. it enables DeepAttest to prevent off-line data tamper if the attacker tries to modify the memory content stored in untrusted environment. During program execution, the *dynamic* trigger is generated from two sources: (i) Memory change signal provided by OS monitoring. OS keeps monitoring the status change of pages allocated for the DNN program and raises a dynamic trigger signal if any online data pages modification is detected; (ii) A timestamp signal from the trusted timer [Int17]. The dynamic trigger from the secure timestamp has a fixed interval and provides enhanced security when the OS memory monitoring signal is tampered. Incorporating the dynamic triggering scheme is important to

Algorithm 2 Fingerprint extraction in the attestation stage.

```
INPUT: Queried DNN ($\mathscr{T}'$); Decoding threshold ($\tau$);
    Owner's FP keys ($\{l, C, U, X\}$).
OUTPUT: Computed $BER$ of fingerprint matching.
1  Trigger Generation: Produce a hybrid trigger signal:
        $S_{hybrid} \leftarrow S_{static} \vee S_{dynamic}$
2  if $S_{hybrid}$ == True then
        Acquire weights in the marked layer:
            $\mathbf{w}' \leftarrow Get\_Marked\_Weights\ (\mathscr{T}', l)$
        Extract the FP vector: $\mathbf{f}' \leftarrow X\mathbf{w}'$
        Recover the coefficient vector: $\mathbf{b}' \leftarrow \mathbf{f}'^T U$
        Decode the code-vector:
            $\mathbf{c}' \leftarrow Hard\_Thresholding\ (\mathbf{b}', \tau)$
        Check FP matching:
            $BER \leftarrow Compute\_BER(\mathbf{c}', \mathbf{c})$
        Return: Obtained $BER$ for FP matching.
3: else
        Go back to step 1
```

enable online data tamper detection. The final trigger signal is the logic-OR of the static and the dynamic signal. DeepAttest is able to detect both off-line and online data tamper due to the incorporation of the static and dynamic trigger, respectively.

❷ **Secure Fingerprint Detection.** When the trigger is enabled, the queried DNN program is suspended until the attestation finishes. The fingerprint of the DL model is extracted with the TEE support and compared with the true value stored in the secure memory. More specifically, DeepAttest first acquires the weights of the marked layer by moving them from the untrusted memory into the secure memory within the TEE. Meanwhile, OS locks the pages allocated for the queried DNN program if it has the capability. The core of the online attestation is recovering the code-vector $\mathbf{c}^{'} \in \{0,1\}^{v}$ from the weight data W. This reconstruction involves matrix multiplication and an element-wise hard-thresholding as shown in Algorithm 2. Note that the recovering of each bit in $\mathbf{c}^{'}$ is independent, DeepAttest leverages this observation from two perspectives (detailed in Section 4.6.2): (i) *Data pipeline*: DeepAttest utilizes data independence for parallel computation, thus reduces the attestation latency; (ii) *Scalability*: DeepAttest allows transferring partitioned blocks of the weight data into the secure memory in TEE. As such, DeepAttest attestation can be applied to arbitrary large DL model and TEE platforms with limited secure memory.

■ **Attestation Case Study:** We demonstrate how DeepAttest authenticates a DNN program using the extracted FP below. Let us consider a codebook $C_{7\times7}$ shown in Equation (4.3). The FPs for 7 users shown in Equation (4.4) are constructed using the columns of the codebook C and the basis matrix U as described in Section 4.5.1.

$$C = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}, \tag{4.3}$$

$$\begin{cases} \mathbf{f_1} = -\mathbf{u_1} - \mathbf{u_2} + \mathbf{u_3} - \mathbf{u_4} + \mathbf{u_5} + \mathbf{u_6} + \mathbf{u_7}, \\ \dots \\ \mathbf{f_6} = +\mathbf{u_1} + \mathbf{u_2} - \mathbf{u_3} - \mathbf{u_4} + \mathbf{u_5} + \mathbf{u_6} - \mathbf{u_7}, \\ \mathbf{f_7} = +\mathbf{u_1} + \mathbf{u_2} + \mathbf{u_3} - \mathbf{u_4} - \mathbf{u_5} - \mathbf{u_6} + \mathbf{u_7}, \end{cases} \tag{4.4}$$

Let us take the first device FP as an instance where the ground-truth FP ($\mathbf{f_1}$) is stored in the secure memory. For a legitimate DNN program whose weights carry the FP vector $\mathbf{f}^{'} = \mathbf{f_1}$, the corresponding coefficient vector can be recovered by computing the correlation of the fingerprint with the basis vectors:

$$\mathbf{b}^{'} = \mathbf{f}^{'T}[\mathbf{u_1}, ..., \mathbf{u_7}] = [-1, -1, +1, -1, +1, +1, +1].$$

The code-vector is then extracted by the inverse linear mapping $c_{ij} = \frac{1}{2}(b_{ij} + 1)$, resulting in $\mathbf{c}^{'} = [0, 0, 1, 0, 1, 1, 1]$. Since the recovered code-vector $\mathbf{c}^{'}$ exactly matches $\mathbf{c_1}$ (which is the first column of the codebook in Equation (3.3)), DeepAttest returns $BER = 0$ and allows the queried DNN program to execute normal inference. This example shows that DeepAttest respects the *integrity* requirement in Table 4.1 and can effectively detect the embedded FP in the legitimate DNN. Furthermore, the computation required to recover the FP code-vector is simple, rendering DeepAttest lightweight.

Note that for the first device that has the device-specific code-vector $\mathbf{c_1}$), any DNN program that cannot yield a matching code-vector (which implies a non-zero BER) will be aborted by our attestation protocol. As such, DeepAttest also satisfies the *reliability* requirement by terminating unauthorized DNN programs. The secret FP keys are stored in the secure memory inside the TEE and is tamper-resistant, suggesting DeepAttest's *security*.

4.6 DeepAttest Optimization

DeepAttest framework integrates innovative hardware optimization techniques to ensure high security (Section 4.6.1) and efficiency (Section 4.6.2). We explicitly discuss two design optimization below.

4.6.1 Shredder Storage

DeepAttest utilizes a *'shredder'* storage format instead of continuous storage to provide stronger security against the fault injection attack. More specifically, DeepAttest shuffles the weights and stores the resulting data in the untrusted memory. Note that the shuffling pattern is determined by the owner in the off-line stage, thus the attacker has no knowledge about the locations of the marked weights. This method is intrigued by the idea of Oblivious RAM (ORAM) where the memory blocks are duplicated and shuffled to hide the memory access pattern from adversary [GO96]. However, we consider a different scenario where data shuffling is performed inside the model parameters to prevent fault injection.

Figure 4.4 illustrates the intuition of high attestation security using shredder storage. When the marked weights are stored continuously, the adversary can easily find a safe position to inject malicious memory blocks without overlapping with the marked region (shadowed area). If the blocks containing the marked weights are shuffled, our online attestation scheme is more likely to yield a non-zero BER and abort the program.

DeepAttest enforces a theoretical upper bound on the success rate (η) of the attacker who aims to perform fault injection while ensuring $BER = 0$ for attestation. We formulate the mathematical problem as follows. Assuming all weights of the deployed DNN takes N memory

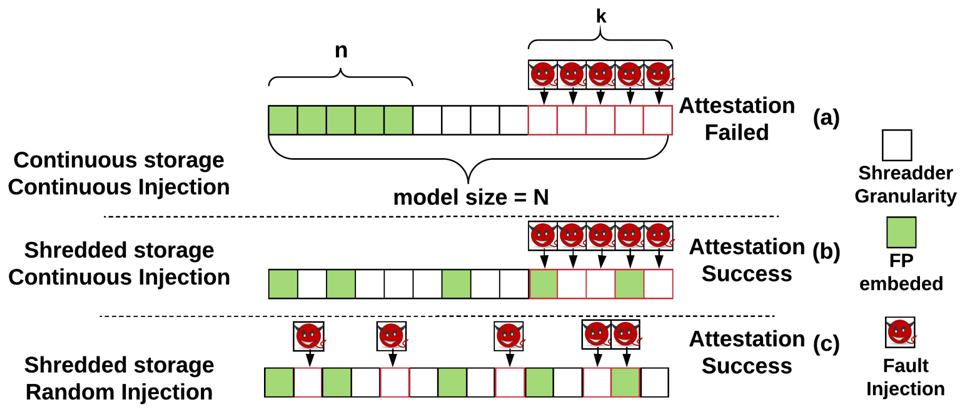


Figure 4.4. Security optimization using shredder storage

blocks where n of them carry the device-specific FP information. The attacker tries to inject k segments of equal size (s blocks) into the memory while still pass online attestation. We assume the attacker does not know the storing pattern of weights and randomly inserts his malicious blocks into the memory. One can see that the intervals between the adjacent marked blocks determine the success probability of the attacker. DeepAttest's shredder storage independently and randomly allocates each marked block, thus the distribution of the locations of the marked blocks can be modeled as a *Poisson Process* with the rate $\lambda = \frac{n}{N}$. As such, the interval X between two neighboring marked blocks is a random variable with the distribution function:

$$P(X \geq s) = e^{-\lambda s}. \tag{4.5}$$

For n marked blocks, the corresponding interval sequence has length $n+1$, thus can be denoted as $S_X = \{X_1, X_2, ..., X_{n+1}\}$. Recall that X_i are i.i.d and satisfies exponential distribution parameterized by λ. To successfully insert a single segment, the adversary can only select the intervals which have values equal or larger than the segment size s. The number of such intervals satisfying the constraints $\{X_i \in S_X, X_i \geq s\}$ is a random variable with *Binomial Distribution*. More specifically, B is a binomial distributed random variable where $n+1$ independent trials are conducted with the success rate $p = P(X \geq s)$ given in Equation (4.5) for each trial:

$$P(B = b) = C_{n+1}^{b} p^{b} (1-p)^{n+1-b}. \tag{4.6}$$

Combining the above analysis, the success rate of the attacker (parameterized by s and k) can be computed as follows:

$$P_a = \sum_{b=k}^{n+1} P(B = b) \cdot \frac{C_b^k}{C_{n+1}^k}. \tag{4.7}$$

DeepAttest provides a tunable security level against fault injection by selecting the parameter λ and the upper bound on the attack success rate $P_a < \eta$. Figure 4.5 illustrates the detection performance of DeepAttest's shredder storage. The x-axis is the injection ratio defined as $\phi = \frac{k \cdot s}{N}$ and the y-axis is the marked ratio $\lambda = \frac{n}{N}$. We can see from the figure that a larger injection ratio or a larger marked ratio results in a lower attack success rate (i.e., data tampering is more likely to be detected by DeepAttest). Figure 4.5 suggests the trade-off between the resistance against fault injection (measured by the tolerated injection ratio) and the attestation overhead (characterized by the marked ratio used in DNN fingerprinting).

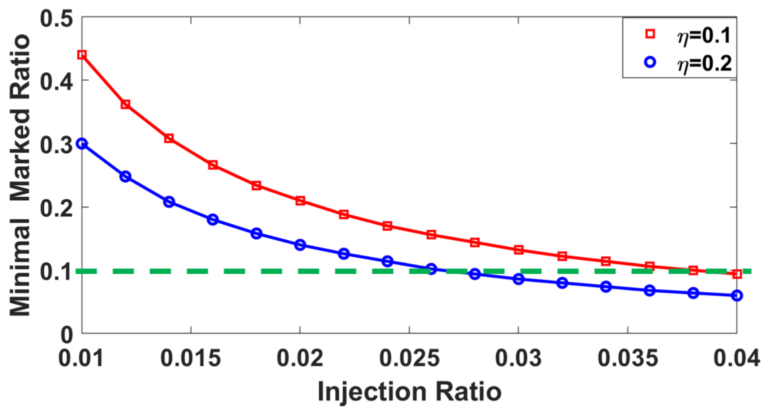


Figure 4.5. DeepAttest's detection performance of fault injection. The relation between the injection ratio and the minimal marked ratio with varying attack success rate is shown.

4.6.2 Efficient Attestation

▪ **Customized Attestation Interval.** As described in Section 4.5.2, DeepAttest utilizes both static and dynamic trigger signals to activate on-device DNN attestation. Such a hybrid triggering mechanism provides a trade-off between security and efficiency. Intuitively, checking the FP with a smaller interval gives stronger security while incurring larger overhead. The device provider can leverage this trade-off to customize the configuration of the attestation trigger in her device based on her resource budget and the desired security level.

▪ **Data Pipelining.** Due to the limited size of enclave memory, we pipeline secure memory copy and the FP computation in TEE as shown in Figure 4.6. To this end, we create two pipelined TEE threads to move the partitioned weight data into the TEE and extract the FP, respectively. The enclave memory occupied by FP extraction is freed once the computation is finished and no intermediate results need to be stored. As such, the weight parameters of large sizes can be easily fitted into the enclave memory. Note that our pipeline optimization is feasible since the reconstruction of each bit in the FP is *independent* and *parallelizable* as discussed in Section 4.5.2. Such a data partitioning scheme further improves DeepAttest's *scalability*.

▪ **Early Termination.** To further reduce the attestation overhead, we avoid unnecessary computation and communication using early termination. More specifically, the online attestation terminates and yields the abortion command once a mismatch between the extracted FP segment and the pre-specified device-specific FP is detected as shown in Figure 4.6.

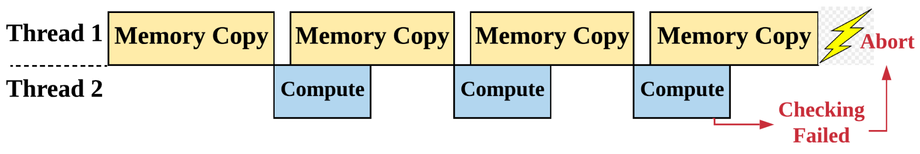


Figure 4.6. Illustration of DeepAttest's data pipeline and early termination for TEE-based attestation.

4.7 Evaluation Results

We assess the performance of DeepAttest according to the requirements discussed in Table 4.1. A codebook $C_{31\times31}$ that accommodates 31 users is used in our experiments. Without explicit hyper-parameter tuning, we set the embedding strength to $\gamma = 0.1$ and fine-tune pre-trained DNN for 5 epochs with the learning rate in the last stage for off-line DNN marking. The threshold for code-vector extraction is set to $\tau = 0.85$. We investigate DeepAttest's performance on Intel-SGX (TEE-support CPU platform) and Graviton-based TEE simulation (GPU platform) [VVB18]. DeepAttest is orthogonal to the existing secure DNN evaluation techniques shown in Figure 4.2 and we provide a horizontal overhead comparison in Section 4.7.6. Details about the hardware platforms and DNN benchmarks are discussed below.

Experimental Setup. To evaluate DeepAttest on TEE-supported CPUs, we use a secure container called SCONE [ATG^{+}16] that is built upon Intel SGX [XSLH16] execution support. When the hybrid trigger is activated, DeepAttest instantiates an attestation process as an enclave inside the SGX engine. We use a host desktop with a i7-7700k processor and measure the energy consumption using *pcm-monitor* utility.

For evaluations on trusted GPUs, we build a TEE simulator for GPUs based on the architecture design proposed in Graviton [VVB18] since there are no existing TEE-supported GPUs available. GPUs can use the device driver to monitor the state of pages inside its memory, thus providing DeepAttest with the trigger signal for attestation. Our TEE-supported GPU simulator restricts the secure memory size to 300MB using memory partition techniques. To ensure isolation, our simulator performs encryption and decryption on the data interacting with the secure memory. More specifically, the weight data stored in the untrusted DRAM is first encrypted by our GPU simulator using authenticated encryption (AES in GCM mode) and copied to the secure memory [VVB18]. The resulting data is then decrypted before it is used in TEE-based FP extraction in the online attestation stage. The encryption and decryption latency follows [Syn17]. We use Nvidia RTX 2080 as the GPU base and measure the power consumption using *nvidia-smi* utility.

DNN Benchmarks Summary. We corroborate DeepAttest's effectiveness on various DNN benchmarks and summarize them in Table 4.2. Since DeepAttest utilizes a hybrid trigger whose overall activation interval is uncertain, we assume the average trigger interval is $f = 100$, meaning that the attestation is run once every 100 images. We emphasize that DeepAttest enables the owner to customize the trigger configuration and show the attestation overhead under different intervals in Section 4.7.5. We set the minimal marked ratio to $\lambda_m = 0.1$, ensuring a maximal success rate of fault injection $\eta = 0.1$ under injection ratio $\phi > 0.04$ (Figure 4.5).

Table 4.2. Summary of the evaluated benchmarks.

Benchmark	Dataset	Model Size (MB)	Multiply-Add Operations (Mops)	Marked Layer Size (MB)
MNIST-CNN	MNIST [LBBH98]	1.3	24	0.13 (10.1%)
CIFAR-WRN	CIFAR10 [KH$^+$09]	2.4	198	0.29 (12.3%)
VGG16	ImageNet [DDS$^+$09]	276.7	25180	28.3 (10.2%)
MobileNet	ImageNet [DDS$^+$09]	8.4	569	1.05 (12.6%)

DeepAttest API. Our end-to-end solution provides a highly-optimized API compatible with current DL frameworks and can perform the two key phases in Algorithm 1 and 2. Furthermore, DeepAttest API provisions *tunable security level and attestation overhead* by allowing the owner to specify the security parameters including the TEE platform (CPU/GPU/other co-processors) for attestation, upper bound on fault injection success η, tolerated injection ratio ϕ, code-vector length v, and the trigger configuration.

4.7.1 Fidelity

Table 4.3 shows the test accuracy of the baseline model and the corresponding marked model for each benchmark in Table 4.2. The marked accuracy is the average value of the total 31 fingerprinted models. One can see that the accuracy of the marked model is comparable to the one of the baseline model, indicating that DeepAttest's off-line marking phase preserves the functionality of the pre-trained model. Slight accuracy improvement can be observed in several benchmarks. This is due to the fact that adding regularization to the training process helps to mitigate model over-fitting [SHK$^+$14, DRCK19].

Table 4.3. Fidelity requirement. The baseline accuracy is preserved after fingerprint embedding in the underlying benchmarks.

Benchmark	**MNIST-CNN** [Kat16]		**CIFAR-WRN** [UNSS17]		**VGG16** [SZ14a]		**MobileNet** [HZC$^+$17]	
Setting	Baseline	Marked	Baseline	Marked	Baseline	Marked	Baseline	Marked
Test Accuracy (%)	99.52	99.66	91.85	92.03	91.20	91.23	85.83	85.75

4.7.2 Reliability and Integrity

▪ **Reliability.** The *reliability* criterion requires that the unauthenticated DNN program shall not be allowed for execution, which is equivalent to yielding a non-zero BER for the queried model in the online attestation stage. We consider the following two sources of an illegitimate DNN: (i) Arbitrary unmarked DNN programs. The malicious user may intend to overuse the device by executing a DL model that is not authorized by the owner; (ii) Fault injection into an authenticated DNN program. The adversary may perform fault injection on the legitimate DNN for malicious purpose (e.g.,malware insertion). Note that the first scenario can be considered as a special case of the second one where the faulty injection level is sufficiently large.

To evaluate DeepAttest's reliability under the above unintended modifications, we add random Gaussian noise with zero mean, different standard deviation (magnitude of noise) and different spatial range (percent of modified elements) to the weight matrix in the marked layer. Figure 4.7 shows the resulting BER of the extracted FP after adding noise to the weights in the marked conv layer and the marked FC layer. One can see that the extracted BER becomes non-zero for small values of noise range and noise magnitude, indicating that DeepAttest can effectively forbid the maliciously modified DNN program from execution.

▪ **Integrity.** The *integrity* criterion means that legitimate DNNs shall pass the attestation and run normal inference. Such a requirement suggests that the attestation protocol shall yield a high FP detection rate (BER=0) for marked models. Figure 4.7 indicates that DeepAttest respects the integrity criterion since the BER is zero when no noise is added to the marked weights.

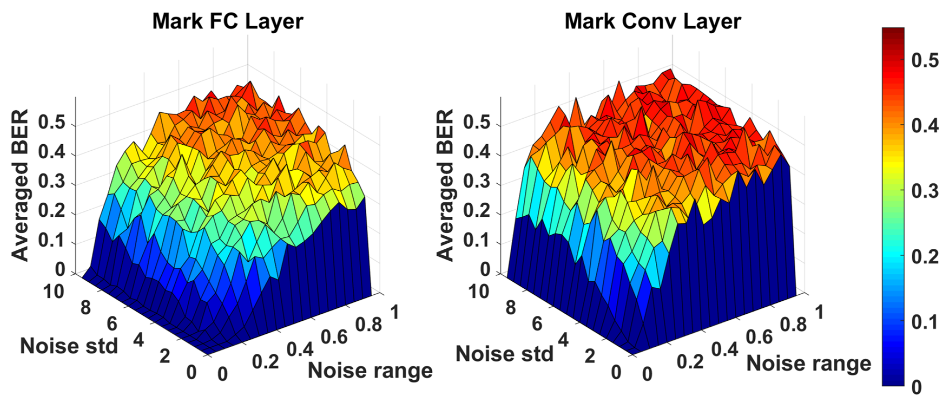


Figure 4.7. Reliability and integrity assessment of DeepAttest under noise.

4.7.3 Security

DeepAttest is secure against fingerprint forgery attack. To construct a DNN program that has the same device FP as the one stored in the secure memory inside the TEE, the adversary needs to know: (i) The DNN marking method (i.e., Algorithm 1); (ii) The secret projection matrix X, orthogonal matrix U, and the code-vector $\mathbf{c}$; (iii) Memory addresses of the marked weights stored in shredder storage format. Using brute force search to find all the above information is prohibitively expensive. DeepAttest might be compromised if the underlying TEE is attacked. For instance, Intel SGX has been identified to be vulnerable to side-channel attacks [LSG$^+$17, BMD$^+$17]. To address the susceptibility, hardware- and software-based defenses have been proposed [LSG$^+$17, MAK$^+$17]. DeepAttest is orthogonal to these methods and can be further secured when integrated with them.

4.7.4 Qualitative Overhead Analysis

We provide a qualitative analysis of the DeepAttest's overhead. Since the DNN marking is an off-line, one-time process, we focus on the overhead in the online attestation phase here. Recall that the weights in the marked layers are transferred from the untrusted memory to the secure memory inside the TEE to extract the FP as outlined in Algorithm 2. The data communication overhead is $\mathcal{O}(NH)$ where N is the embedding dimension and H is the number of output channels/units as described in Section 4.5.2. To reduce attestation computation overhead, DeepAttest *pre-computes* the product $X^T U$ used in FP extraction ($b \leftarrow \mathbf{w}^T \cdot X^T U$ in Algorithm 2). As such, the computation complexity of online attestation is $\mathcal{O}(vN)$. The above overhead analysis holds for both conv and FC layers.

4.7.5 Efficiency

We use the latency and energy consumption per image on the untrusted CPU/GPU as the base value and measure the relative overhead of DeepAttest. As shown in Figure 4.8, DeepAttest incurs on average 7.2% and 4.4% relative latency overhead on the TEE-support CPU and GPU platforms across all benchmarks, respectively. The average energy overhead of DeepAttest is 4.1% and 1.2% for CPU and GPU devices. The normalized energy overhead incurred by DeepAttest is low since the power in the attestation is much smaller compared to one of DNN inference on a given platform. The normalized overhead of DeepAttest depends on the DNN architecture for both TEE-supported CPU and GPU platforms. More specifically, DeepAttest incurs smaller latency and energy overhead on DL models with larger size (parameter count) and more operations. This is due to the fact that small DNNs (e.g., MNIST-CNN) have lower

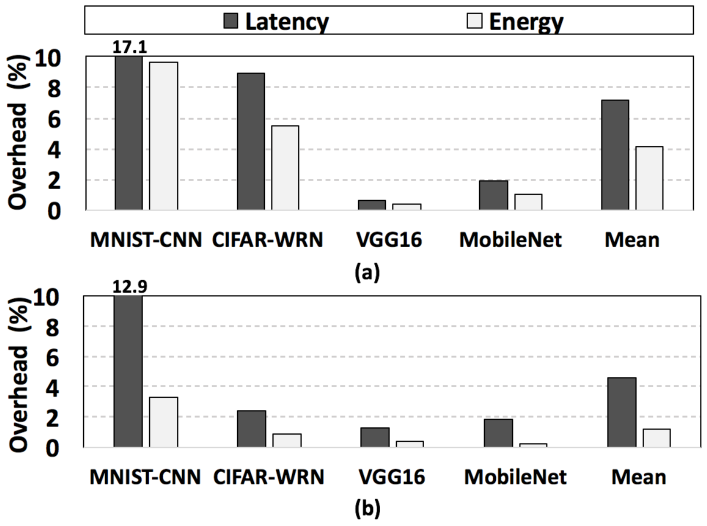


Figure 4.8. DeepAttest's normalized latency and energy overhead on TEE-supported (a) CPU and (b) GPU platforms.

base overhead compared to large models (e.g., VGG16). Comparing the overhead on different platforms, DeepAttest is more efficient when executed in the TEE in GPUs than CPUs.

Overhead Breakdown

To better understand the source and bottleneck of attestation overhead, we analyze the individual runtime of secure memory copy and FP extraction computation. Figure 4.9 shows the runtime contribution of these two processes. One can see that secure memory copy dominates DeepAttest's overhead in small benchmarks. This is due to the fact secure memory copy involves data loading/encryption and assistant operations executed when entering and exiting an enclave [Har17]. DeepAttest's secure computation is lightweight since: (i) Secure memory reading is faster than writing [WBA17]; (ii) The involved operations are simple (described in Algorithm 2). The overhead of secure FP computation is affected by the dimensionality of the marked weights, thus varies across different benchmarks as detailed in Section 4.7.5.

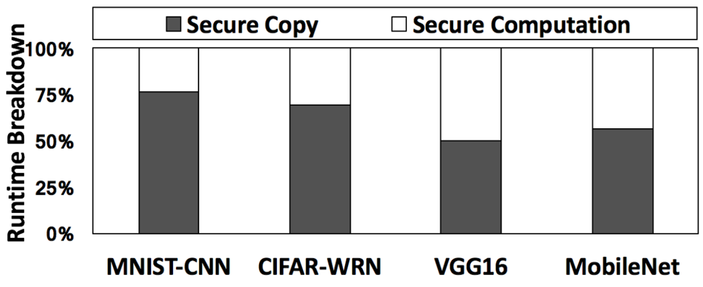


Figure 4.9. Runtime contribution breakdown of DeepAttest on Intel SGX without dataflow optimization.

Optimization Improvements

We optimize DeepAttest's dataflow using data pipeline as discussed in Section 4.6.2 to hide the latency of secure FP computation Figure 4.10 illustrates the effectiveness of data pipeline to reduce the attestation latency. On average, DeepAttest's optimized dataflow engenders 1.42$\times$ speedup and further improves its efficiency. Early termination optimization also helps to reduce the overhead. According to Algorithm 2, it is intuitive that the amount of overhead saving benefited from early termination is approximately linear with respect to the position of the first identified FP mismatch segment.

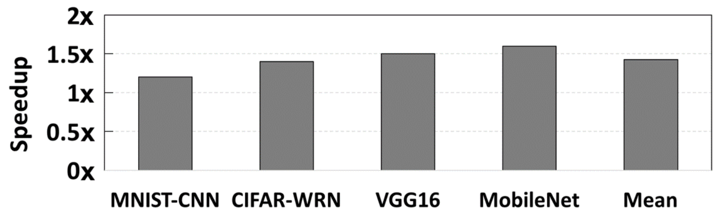


Figure 4.10. Speedup of DeepAttest's data pipeline optimization for secure FP extraction on Intel SGX.

Sensitivity Analysis

▪ Sensitivity to attestation interval.

DeepAttest leverages a hybrid trigger mechanism to activate the attestation as discussed in Section 4.5.2. Recall that we denote the average activation interval of the hybrid signal as f (one round of attestation every f inputs). Figure 4.11 shows DeepAttest's overhead with various attestation interval f on CIFAR-WRN benchmark and Intel-SGX platform. One can see that DeepAttest's overhead decreases linearly when f increases. Higher trigger interval results in smaller normalized attestation overhead for arbitrary DNNs. For instance, the relative latency overhead drops to 1.1% on CIFAR-WRN when $f = 800$.

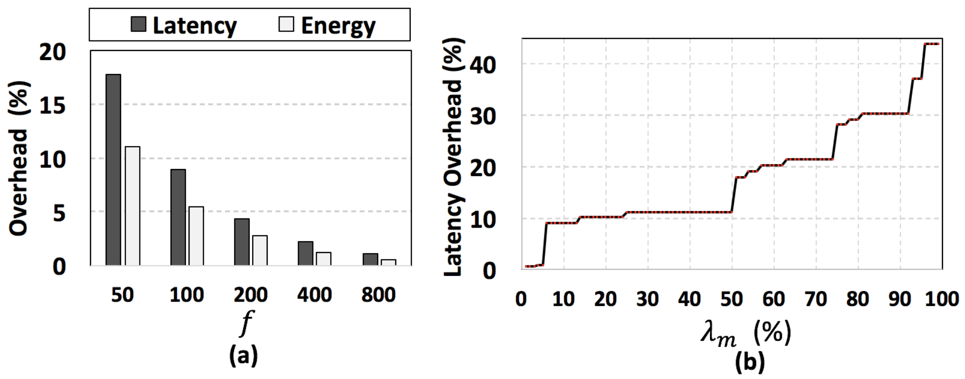


Figure 4.11. Sensitivity of DeepAttest's normalized overhead to the (a) attestation interval f, (b) minimal marked ratio λ_m on CIFAR-WRN (tested on Intel-SGX).

■ **Sensitivity to the the marked ratio.** The possible values of marked ratio λ for a specific DNN are discrete since DeepAttest performs FP embedding with the granularity of a single layer. Given the owner-specified security level η and the tolerant injection ratio ϕ, DeepAttest finds the minimal marked ratio λ_m (Section 4.6.1) and the optimal combination of layers that yields the minimal latency. A large marked ratio λ (i.e., percentage of marked weights) results in a larger latency overhead as shown in Figure 4.11 (b). Note that λ also impacts the security level of DeepAttest against fault injection as discussion in Section 4.6.1, thus providing a trade-off between overhead and security.

■ **Sensitivity to kernel size.** DeepAttest's overhead is affected by the kernel dimension of the marked layer as we analyze in Section 4.7.5. Figure 4.12 shows the breakdown of relative runtime overhead for TEE-based attestation and evaluation process as the kernel size changes. For attestation of a conv layer, the runtime overhead is dominated by secure memory copy. However, the runtime discrepancy between secure copy and secure computation becomes smaller as the kernel size increasing since the contribution of the fixed overhead in secure copy (extra hardware operations) is reduced. For TEE-based inference, the runtime of conv kernels is dominated by secure computation. As for FC layers, secure memory copy is the performance bottleneck for both TEE attestation and evaluation due to the large parameter size.

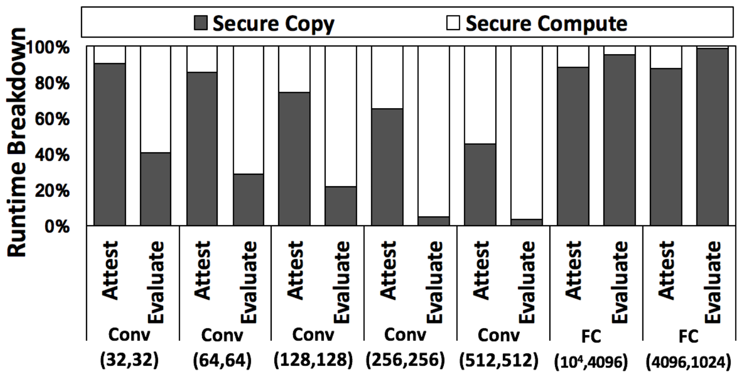


Figure 4.12. Runtime (relative) breakdown of TEE attestation and evaluation with varying kernel size on Intel-SGX without dataflow optimization. We use conv(F,H) and FC(F,H) to denote a convolutional layer with size $(3,3,F,H)$ and a fuly-connected layer with size (F,H), respectively.

4.7.6 Comparison with Related Works

In this section, we compared DeepAttest with the state-of-the-art secure DNN evaluation techniques listed in Figure 4.2. Note that DeepAttest aims to address a new security concern (i.e., hardware-level IP protection and usage control) for DNN applications that has not been identified by previous works. DeepAttest is orthogonal to the existing secure DNN techniques that target at different vulnerabilities of DL models and can be easily integrated within them. As such, we present a horizontal performance comparison to demonstrate the relative overhead required by different security/privacy-protection DNN methods.

In our experiments, we use the open-sourced code of Slalom [TB18] to evaluate its performance of verifying the integrity of DNN evaluation. Slalom requires the DNN weights and the input data to be quantized in order to satisfy the finite-field assumption. We adhere to the quantization technique and the pre-processing method that yields the highest throughput in [TB18] throughout our experiments. We emphasize that our framework does not require any model compression. As such, our assessment of DeepAttest's overhead is conservative. We use the trusted GPU simulator discussed in Section 4.7 in the experiments requiring TEE-supported GPU. The design optimization discussed in Section 4.6 are used. We detail the comparison between DeepAttest and related works below.

Comparison of Secure Memory Copy

Figure 4.13 illustrates the theoretical (minimal) size of secure memory copy required by different secure DNN techniques assuming the TEE is not memory-bounded. Slalom [TB18] incurs large overhead of secure memory copy since it outsources linear operations of DNN inference to the untrusted GPU. Therefore, all intermediate activations need to be transferred into the TEE to complete non-linear operations. This results in an approximately linear secure copy size with respect to the number of evaluated images as shown in Figure 4.13 (a) and (b). Fully TEE-based DNN evaluation only requires to transfer all weight data and input data, thus is less sensitive to the number of inputs. DeepAttest's memory copy size is not sensitive to the number

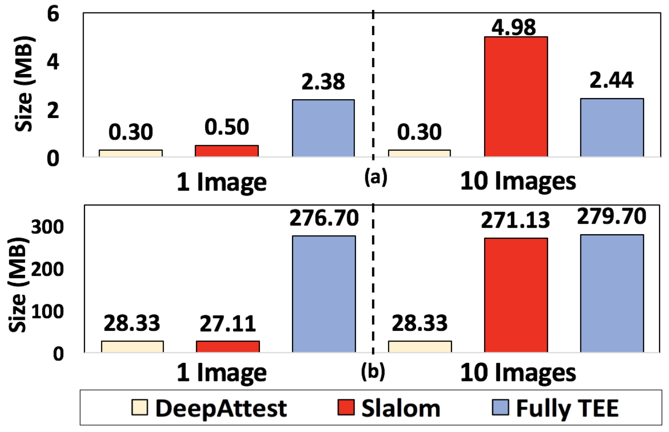


Figure 4.13. Comparison of the theoretical secure memory copy size to the TEE required by different secure DNN techniques on (a) CIFAR-WRN and (b) VGG16 benchmark.

of inputs since it adopts a hybrid triggering scheme where the attestation is performed every batch of f images. Furthermore, the secure copy size of DeepAttest is small for a given attestation interval due to the deployment of shredder storage optimization, which ensures security for a smaller value of the marked ratio λ.

Comparison of Latency

▪ **Overhead on CPU-based Inference.** Figure 4.14 shows the normalized latency required by different secure DNN evaluation methods and DeepAttest where the baseline inference is performed on the untrusted CPU. Implementing DNN inference fully inside TEE is on average

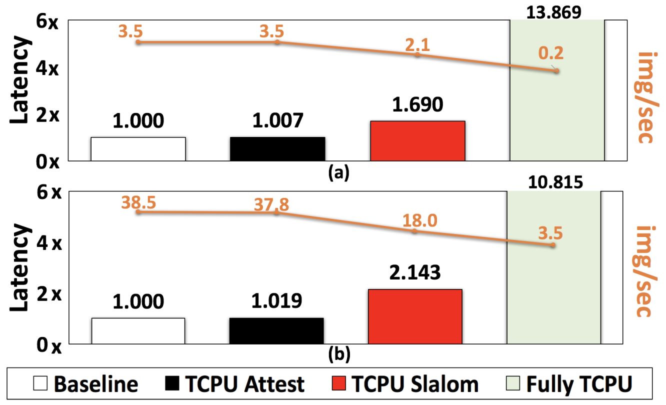


Figure 4.14. Comparison of relative latency between different secure DNN techniques when the inference is run on CPU. VGG16 (a) and MobileNet (b) are evaluated.

12.34× slower than the baseline evaluation on the untrusted CPU. Slalom [TB18] outsources linear operations to the untrusted CPU and non-linear parts to Intel-SGX, resulting in an average normalized latency of 1.72× to provide verifiable results. DeepAttest incurs negligible relative latency of 0.7% and 1.9% on VGG16 and MobileNet respectively, thus is highly efficient.

■ **Overhead on GPU-based Inference:** Figure 4.15 shows the normalized overhead of different secure DNN methods and DeepAttest where baseline inference is performed on untrusted GPUs. Full DNN inference in TEE-supported GPUs results in an average normalized latency of 8.75× due to the overhead of isolated execution and secure memory access. Slalom [TB18] outsources linear operations to the untrusted GPU and computes the nonlinear part in Intel-SGX (TCPU), resulting in a normalized latency of 6.43× and 5.69× on VGG16 and MobileNet, respectively.

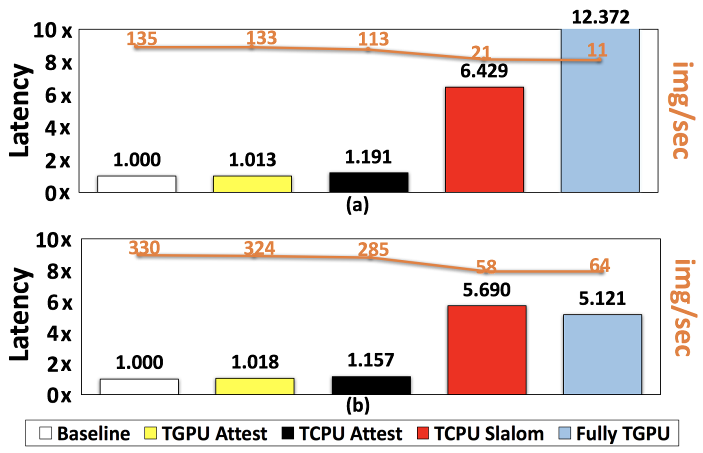


Figure 4.15. Comparison of normalized latency incurred by different secure DNN methods when the inference is run on GPU. VGG16 (a) and MobileNet (b) are assessed here.

DeepAttest can perform the FP extraction computation either in the trusted CPU or the trusted GPU if the TEE exists on the pertinent GPU. More specifically, DeepAttest results in 19.1% and 15.7% additional latency overhead when attesting VGG16 and MobileNet on the TEE-supported CPU (Intel-SGX), respectively. Alternatively, DeepAttest can attest the deployed DNN program using the TEE support inside the GPU to avoid data communication between CPU and GPU. In this case, DeepAttest incurs only 1.3% and 1.8% extra latency on VGG16 and MobileNet, respectively.

4.8 Summary

In this chapter, we present a systematic solution to device-level IP protection and usage control for DNN applications. We propose DeepAttest, the first on-device DNN attestation framework that verifies the legitimacy of the deployed DNN before allowing it to execute normal inference. DeepAttest leverages an Algorithm/Software/Hardware co-design principle and incorporates various design optimization techniques to minimize the overhead. Our framework allows the device providers to explore the trade-off between security level and attestation overhead by specifying security parameters including tolerance level of fault injection, marked ratio, and trigger configuration. Extensive experimental results corroborate that DeepAttest satisfies all criteria for a practical attestation scheme including fidelity, reliability, integrity, security, scalability, and efficiency.

4.9 Acknowledgements

This work was supported in part by the Office of Naval Research (ONR) under award number N00014-17-1-2500, and Multidisciplinary University Research Initiative (MURI) under award number FA9550-14-1-0351. This chapter, in part, has been published at 2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA) and appeared as Huili Chen, Cheng Fu, Bita Darvish Rouhani, Jishen Zhao, Farinaz Koushanfar, “DeepAttest: An End-to-End Attestation Framework for Deep Neural Networks” [CFR$^+$19]. The dissertation author was the primary author of this material.

Chapter 5

SpecMark: Spectral Watermarking for Automatic Speech Recognition

Automatic Speech Recognition (ASR) systems are widely deployed in various applications due to their superior performance. However, obtaining a highly accurate ASR model is non-trivial since it requires the availability of a massive amount of proprietary training data and enormous computational resources. As such, well-trained ASR models shall be considered the intellectual property (IP) of the model designer and need to be protected against copyright infringement attacks.

In this chapter, I introduce SpecMark, the first spectral watermarking framework that seamlessly embeds a watermark (WM) in the spectrum of the ASR model for *ownership proof*. SpecMark identifies the significant frequency components of model parameters and encodes the owner's WM in the corresponding spectrum region before sharing the model with end-users. The model builder can later extract the spectral WM to verify his/her ownership of the marked ASR system. We evaluate SpecMark's performance on DeepSpeech model with three different speech datasets. Empirical results corroborate that SpecMark preserves the recognition accuracy of the original system and incurs negligible overhead for both spectral WM embedding and extraction. Furthermore, SpecMark can sustain diverse model modifications, including parameter pruning and transfer learning. The *training-free* watermark embedding scheme in SpecMark makes it applicable to resource-constrained systems.

5.1 Introduction

Automatic Speech Recognition (ASR) is a technology that allows humans to interact with machines using their voices. The emergence of Deep Learning (DL) techniques has revolutionized ASR systems and enabled their commercialization. Voice assistants including Google Home, Amazon Alexa, Microsoft Cortana, and Apple Siri are examples of ASR's wide deployment [HLZ$^+$15, LSN$^+$17, Hoy18, KB18]. The success of modern ASR systems relies on the superior performance of the underlying DL models [HCC$^+$14, AAA$^+$16]. While current research in this field mainly focuses on increasing the accuracy of ASR models, we take an orthogonal perspective to ASR applications and investigate the *copyright concerns* of pre-trained models. Training a highly accurate ASR model is expensive since this process requires: (i) Access to an enormous amount of proprietary training dataset; (ii) Allocating extensive computing resources and time [SK16, JPM$^+$18]. As such, the resulting ASR system shall be considered the Intellectual Property (IP) of the model developer and needs to be protected to preserve the competitive advantage of the owner.

Regularization is a typical approach to increasing the generalization capability of a DL model to unseen datasets [YYS$^+$13, GHZZ18]. Prior works have explored regularization and adapted digital watermarking for *ownership proof* of Deep Neural Networks (DNNs). Existing DNN watermarking techniques can be categorized into two types based on the model deployment scenario. A line of works assumes the model internals are known in the watermark (WM) extraction stage (i.e., *'white-box'* setting) and inserts the WM by training the DL model with additional *regularization* loss terms [UNSS17, CRF$^+$19, CFR$^+$19, DRCK19]. In this case, the WM is typically a binary sequence. For instance, [UNSS17] modulates the distribution of static weights to encode the WM information. DeepSigns [DRCK19] explores the distribution of the dynamic activation maps as the WM carrier.

Another line of research assumes the DL model is employed as a remote service (i.e., 'black-box' setting) where only the input-output behavior of the model is known [ABC$^+$18, ZGJ$^+$18, CRK19, GP19, LMPT20]. Particularly, the model owner generates a secret WM key set (i.e., input-output pairs) and uses it to finetune the model. In this case, the WM takes the form of *statistically biased responses* and is encoded in the decision boundary of the model. Note that both white-box and black-box watermarking methods discussed above require expensive model re-training. These DL watermarking techniques are also shown to be vulnerable to strategic model disturbance [CWB$^+$21, XWL21]. Such limitations motivate us to design a more efficient and resilient watermarking scheme for DNN IP protection.

Contributions. In this work, we propose SpecMark, the first systematic *model-level spectral watermarking* framework that protects the IP of contemporary ASR systems. SpecMark encodes the ownership information in the *spectrum characteristics* of the ASR model while preserving the task accuracy of the marked model. More specifically, we propose to *spread* the watermark over multiple random subsets of the significant spectra components of the model parameters to ensure that SpecMark is *robust* and *secure*. Furthermore, our framework is highly *lightweight* since it embeds the WM strategically in the *spread spectrum (SS)* of the ASR model *without re-training* it. We validate the feasibility and robustness of SpecMark using DeepSpeech v2 [AAA$^+$16] on AN4, Command Voice, and LibriSpeech datasets. Our spectral watermarking technique is compatible with existing DL-based ASR systems and paves the way for safe and reliable deployment of ASR models.

5.2 Related Works

Prior works have identified the IP concern of DNNs and adapted digital watermarking techniques for ownership authentication. We categorize existing methods into two types based on the application scenarios of the DL model. We introduce each type in detail as follows.

White-box Watermarking. In the white-box setting,the pre-trained DL model for the intended task (computer vision, speech recognition, etc.) is shared with the end-users. This means that model internals including weight parameters and activation maps are publicly accessible. Such a deployment scenario is common with the increasing trend of knowledge exchange among the research community. Uchida *et al.* [UNSS17] take the first step of DNN watermarking and develops a customized regularization loss to embed the watermark in the weight distribution of the selected layer. To improve security and robustness, DeepSigns [DRCK19] proposes to insert the WM in the distribution of dynamic activations corresponding to the secret key input. DeepMarks [CRF$^+$19] uses weight regularization and incorporates anti-collusion codes for WM design to enhance the watermark's resistance against collusion attacks.

Black-box Watermarking. In the black-box setting, the pre-trained DL model is employed as a remote service where the customer sends his data to the cloud server and receives the corresponding output. Since the DL model is only available as an oracle, prior works suggest to craft secret input-output pairs as the WM. To insert the WM in the model's decision boundary, the WM key set is used to finetune the model. As an example, the paper [LMPT20] proposes to craft adversarial samples as the WM set, which results in high false alarm rates due to the transferability of adversarial examples. To resolve the issue, DeepSigns [DRCK19] generates random inputs and random labels as the WM key set.

Existing DL watermarking techniques have the following constraints: *(i) Application domain.* All of the above-mentioned DNN watermarking techniques demonstrate their effectiveness on image classification tasks. However, the intrinsic time-evolving nature and the representation form of speech signals distinguish ASR from image tasks. Such a discrepancy might render the watermarking techniques less effective or invalid for ASR systems; *(ii) High WM embedding overhead.* Current DNN watermarking primitives embed the WM via model re-training, which might be prohibitively costly; *(iii) Robustness.* Current watermarking schemes are susceptible to strategic model disturbance such as transfer learning [CWB$^+$21]. To address the above limitations, we propose SpecMark, the first practical and resilient model-level watermarking framework that is suitable for ASR systems.

5.3 Problem Statement

We define the problem of ASR model watermarking in this section. SpecMark assumes a *white-box* scenario where the model internals are known to the public. We formulate model-level watermarking as a *one-time, post-training* step where the objective is to embed a WM (a binary sequence in our work) in the parameter distribution of the ASR model. To be *practical* and effective in real-world ASR systems, the watermarking technique shall satisfy a set of criteria. We summarize these fundamental requirements in Table 5.1 and present a quantitative assessment of SpecMark's performance in Section 5.5.

Potential Attacks. The model owner inserts a secret watermark in his trained ASR model and shares the marked variant with the public. However, the marked model might undergo unintentional or deliberate model modifications in a practical deployment setting. The robustness criterion in Table 5.1 requires that the WM shall be resistant to potential disruptions and remains detectable. We consider three types of model disturbance attacks: *(i) Parameter pruning:* Parameters with small magnitudes can be zeroed out for computation savings without significant accuracy degradation [HMD15, HKM$^+$17, LWL17]; *(ii) Model fine-tuning:* The converged model can be fine-tuned to find better local optima [TSG$^+$16, NKFL18, CWB$^+$17]; *(iii) Transfer learning:* A pre-trained model might be re-trained on a new dataset for the intended task [CMS12, TSK$^+$18, SRG$^+$16]. We corroborate the robustness of SpecMark spread spectrum watermarking against these attacks in Section 5.5.

Table 5.1. Requirements for an effective watermarking method of ASR systems.

Requirement	Definition
Fidelity	Preserve the functionality of the original model.
Robustness	WM sustains possible model modifications.
Efficiency	Low overhead for WM embedding and detection.
Reliability	High detection rates of the embedded WM.
Integrity	Low false positive rates of WM detection.
Security	WM carrier is difficult to identify.

5.4 SpecMark Methodology

Figure 5.1 shows the global workflow of SpecMark. From the high-level overview, SpecMark takes the pre-trained ASR model and a set of secret WM keys as the inputs. The marked variant of the ASR model is returned as the output. Our spectral watermarking framework consists of two main phases: offline WM embedding and online WM detection. We detail the procedures of each stage below.

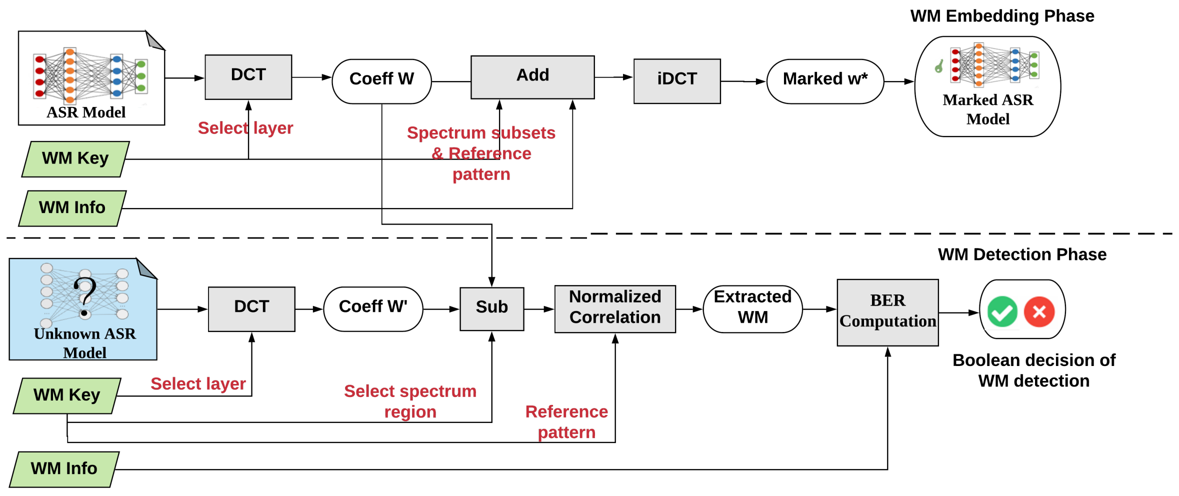


Figure 5.1. Global workflow of SpecMark watermarking framework for ASR systems.

5.4.1 Spectral WM Embedding

SpecMark *spreads* the WM information in the *significant spectrum components* of the target ASR model. Such an embedding mechanism features two advantages: *(i) Security*: Spreading the WM information over many frequency bins ensures that the energy change on a single bin is small and undetectable. The insertion location and content of the WM are only known to the owner, making it difficult to find out by the attacker using random guesses. *(ii) Robustness*: SpecMark's WM is encoded in the important frequency regions of the ASR model. Since feasible model modifications have to leave the significant spectra components intact to maintain high accuracy, the attacker cannot remove our WM without performance degradation.

We define the WM as a binary bit sequence $\mathbf{b}$ of length T where $b_k = \{-1, +1\}, k = 1, ..., T$. To provide security guarantee, SpecMark's WM key has three components: (i) The layer position (denoted by l) whose parameters are selected to carry the WM information; (ii) The secret random seeds ($\mathbf{s}$) that are used to determine the frequency bins modulated by the WM; (iii) The secret reference pattern matrix $\mathbf{U}_{T \times M}$ where T is the length of the WM sequence and M is the number of frequency bins controlled by each WM bit. The k^{th} row of $\mathbf{U}$ is used as the reference vector $\mathbf{u_k}$ to carry the WM bit b_k. Note that elements in each row of $\mathbf{U}$ have equal probabilities of taking two values: $u_{i,j} = \{-\sigma_u, +\sigma_u\}$. We detail each step of SpecMark's WM embedding stage shown in Figure 5.1 below.

■ **Identify Significant Spectra Components.** Given the layer position l in the WM key, SpecMark performs *DCT transformation* on the corresponding weight parameter w and obtains the frequency coefficients $\mathscr{W} = DCT(w)$. Since large values are less sensitive to additive alternations than small values, we select the top N largest elements of $\mathscr{W}$ as the tentative WM insertion locations and denote the resulting index set as I_N. Note that $M \ll N$ such that the spectra components controlled by each WM bit do not overlap with each other.

■ **Encode WM in Random Spectra Subsets.** To enhance watermarking security, SpecMark embeds each WM bit in a random subset of spectra components with the highest values (found by I_N). The insertion location I_k (with size M) for the bit b_k is determined by Equation (5.1) where s_k is the random seed from the WM key. To make the element-wise addition of frequency components feasible, we then use I_k^c to zero-pad the secret reference vector $\mathbf{u_k}$ as shown in Equation (5.2). Here, I_k^c is the complement set of I_k where the whole set is the index range of $\mathscr{W}$. The padded variant $\tilde{\mathbf{u_k}}$ takes the corresponding value from $\mathbf{u_k}$ only when its current index exists in I_k. Finally, the entire WM sequence $\mathbf{b}$ is embedded in the significant part of the DCT coefficients $\mathscr{W}$ using Equation (5.3).

$$I_k = RandomSelect(I_N, M, s_k), \tag{5.1}$$

$$\tilde{\mathbf{u_k}} = ZeroPad(\mathbf{u_k}, I_k^c), \tag{5.2}$$

$$\mathscr{W}^* = \mathscr{W} + \sum_{k=1}^{T} b_k \tilde{\mathbf{u_k}}, \tag{5.3}$$

▪ **Perform Inverse Frequency Transformation.** After embedding the WM in the selected bins of the important spectrum of the ASR model, we convert the resulting frequency map back to the spatial domain using inverse DCT: $w^* = iDCT(\mathscr{W}^*)$. The original weight parameter w of the secret layer l is replaced with w^* to obtain the marked ASR model.

5.4.2 Spectral WM Detection

In the online detection phase, the model owner queries the unknown ASR system and obtains its internal weights. Since the owner knows the WM insertion locations and content, he can concentrate the 'weak' WM signals spread over the particular frequency bins and extract the WM for authorship proof. We detail each step of WM detection shown in Figure 5.1 below.

▪ **Transform Queried Data to Frequency Domain.** Given the WM key, the model owner performs DCT on the weight parameter of the layer l of the queried model $\mathscr{W}^{'} = DCT(w^{'})$.

▪ **Compute Normalized Correlation.** As the developer of the original ASR system, the model owner has the DCT values $\mathscr{W}$ of the unmarked weights w. As such, he can compute the spectral difference $\Delta\mathscr{W}$ between the queried weight and the unmarked one in the DCT domain using Equation (5.4). Then, the normalized correlation between $\Delta\mathscr{W}$ and each reference vector $\mathbf{u_k}$ is computed using Equation (5.5). Note that the vector norm is $||\mathbf{u_k}|| = \sigma_u^2$, since each element in $\mathbf{u_k}$ can only take the value of $-\sigma_u$ or $+\sigma_u$.

$$\Delta\mathscr{W} = \mathscr{W}^{'} - \mathscr{W}, \tag{5.4}$$

$$r_k^{'} = \frac{\Delta\mathscr{W} \cdot \tilde{\mathbf{u_k}}}{||\tilde{\mathbf{u_k}}||}. \tag{5.5}$$

▪ **Determine WM Existence.** After computing the normalized correlation r_k $(k = 1, .., T)$ individually, the corresponding binary WM bit b_k is extracted by taking the sign of the correlation statistics as shown in Equation (5.6). Finally, we compute the Bit Error Rate (BER) between the ground-truth WM sequence $\mathbf{b}$ and the extracted one $\mathbf{b}^{'}$. SpecMark's WM is successfully detected for ownership authentication only when $BER = 0$.

$$b_k^{'} = sign(r_k^{'}). \tag{5.6}$$

5.5 Evaluations

We present a comprehensive assessment of SpecMark's performance according to the watermarking requirements discussed in Table 5.1. The results are summarized in this section.

Experimental Setup. We demonstrate the effectiveness of SpecMark using the DeepSpeech v2 model [AAA$^+$16] and three different speech datasets: AN4, Command Voice, as well as LibriSpeech [Nar20]. To implement SpecMark's spread spectrum watermarking (detailed in Section 5.4.1), we use the following configuration: WM sequence length $T = 16$, candidate range of significant spectra components $N = 5000$, number of frequency bins controlled by each WM bit $M = 20$, and reference strength $\sigma_u = 0.5$. The hidden-hidden weights of the third LSTM layer of DeepSpeech is selected to carry the WM. Similar results are obtained when other layers are used for SpecMark's watermarking. We emphasize that *no model re-training is required* by SpecMark to embed the WM, making our framework lightweight. We use the same hyper-parameters (e.g., learning rate, batch size, and optimization level) as [Nar20] for three WM removal attacks. We perform experiments on Nvidia Titan Xp with 12 GiB memory. We repeat each set of experiments for 10 runs and report the average values in the following section.

5.5.1 Fidelity and Efficiency

Recall that fidelity requires the watermarking technique to preserve the accuracy of the pre-trained model. For ASR tasks, we use Word Error Rate (WER) and Character Error Rate (CER) as the performance metrics. Table 5.2 summarizes the performance comparison results of the ASR system before and after SpecMark's WM embedding. The last two rows show the

Frobenius norm of the weight perturbation introduced by WM insertion in the spatial and the DCT domain, respectively. One can see that SpecMark's spread spectrum watermarking primitive does **not impact the accuracy** of the original model, thus respects the fidelity criterion. This is due to the fact that our framework induces negligible disturbance on the weight parameters (small $||\Delta\mathbf{w}||$ in Table 5.2).

Table 5.2. Fidelity evaluation of SpecMark. The WER and CER of the pre-trained baseline model and the watermarked variant are compared across different datasets.

Datasets	**AN4**		**Command Voice**		**LibriSpeech**	
Models	**Baseline**	**Marked**	**Baseline**	**Marked**	**Baseline**	**Marked**
WER (%)	11.38	11.38	26.72	26.72	18.09	18.09
CER (%)	6.81	6.81	11.63	11.63	7.32	7.32
$\|\|\Delta\mathbf{w}\|\|$	0.20		0.20		0.16	
$\|\|\Delta\mathscr{W}\|\|$	9.11		9.11		7.07	

As for the watermarking efficiency, we analyze the *runtime overhead* of SpecMark's WM embedding and detection procedure. According to SpecMark's mechanism outlined in Section 5.4, we can see that SpecMark has a *fixed computational overhead* for a specific watermarking configuration and a given target ASR system. This implies that SpecMark's overhead is *independent* of the dataset dimensionality, suggesting that our framework is **scalable** to large ASR tasks. In our experiments, the WM embedding and detection time is 97.67 and 10.98 millisecond for all three datasets, respectively. Compared with existing DL watermarking techniques [DRCK19, UNSS17, CRF$^+$19], SpecMark features the highest **efficiency** since no model re-training is required.

5.5.2 Robustness

We discuss three possible attack scenarios in Section 5.3: parameter pruning, model fine-tuning, and transfer learning. In the following of this section, we validate SpecMark's robustness against these attacks with empirical results.

Robustness against Parameter Pruning

We perform standard parameter pruning (i.e., zero-out elements with the smallest magnitudes [HMD15]) on all convolutional and LSTM layers of the marked ASR model. Acceleration-oriented pruning pipeline conducts model re-training to compensate for accuracy loss induced by pruning. In our case, the attacker intends to use pruning for WM removal. It is very unlikely that the attacker has the original training data and the computing power to perform model re-training (otherwise he has less incentive to steal the ASR model.) As such, we measure the test accuracy and BER (for WM detection) of the pruned model without re-training. Figure 5.2 shows SpecMark's robustness against parameter pruning on LibriSpeech dataset. We can see that SpecMark's BER is **less sensitive** to parameter pruning compared to the accuracy metric (WER and CER). As such, the adversary cannot remove the WM by excessive pruning while acquiring a functional ASR model. In our experiments, SpecMark can tolerate up to 99%, 90%, and 90% parameter pruning on AN4, Command Voice, and LibriSpeech datasets, respectively.

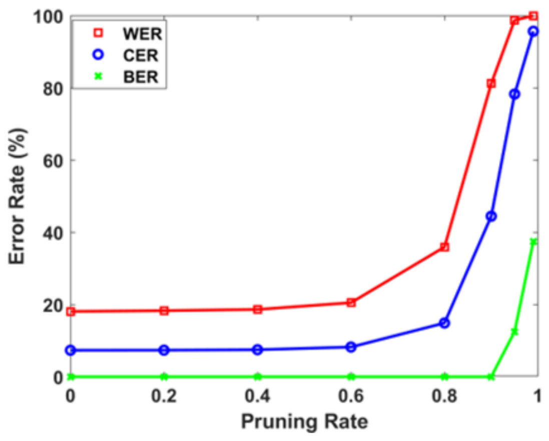


Figure 5.2. SpecMark's robustness against parameter pruning.

Robustness against Transfer Learning

Transfer learning is a popular practice that leverages the features extracted by a pre-trained DL model for a new task [CMS12, TSK+18]. More specifically, the user performs model re-training on his new dataset instead of training a model from scratch. In our robustness evaluation, the DeepSpeech model is first pre-trained on LibriSpeech dataset and marked by SpecMark. The transfer learning attack is then performed by re-training the marked model on AN4 dataset using the same configurations in [Nar20]. Figure 5.3 shows the test accuracy of the marked DeepSpeech model on the new dataset (AN4) and the BER of WM detection during the transfer learning process. We can see that SpecMark's SS WM remains detectable (i.e., BER=0) even if the marked ASR model undergoes transfer learning. This **transferability** of SpecMark's WM makes it suitable for reliable technology exchange in the speech recognition domain.

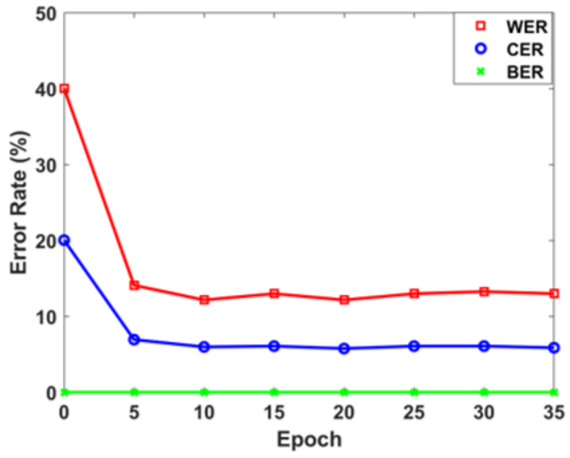


Figure 5.3. SpecMark's robustness against transfer learning.

Robustness against Model Fine-tuning

The nature of model fine-tuning determines that it introduces a *smaller* amount of perturbation to the marked weights compared to parameter pruning and transfer learning. Our evaluation results show that SpecMark still yields *zero BER* for the fine-tuned marked model across all three datasets, thus is **resilient** against model fine-tuning attacks. The detailed results are not shown here for simplicity.

5.5.3 Integrity

Recall that integrity requires the WM detection process to yield small false positive rates (see Table 5.1). This property is important since falsely claiming the ownership of an ASR model might lead to law disputes. To assess the integrity of SpecMark, we extract the watermark from unmarked ASR models following the procedures in Section 5.4.2. Table 5.3 shows the integrity evaluation results on LibriSpeech dataset while similar results are obtained on the other two datasets. ‘Unmarked1’ and ‘Unmarked2’ are models trained on the *same dataset* as the marked one (LibriSpeech in this case). ‘Unmarked3’ and ‘Unmarked4’ are models trained on different datasets (AN4 and Command Voice, respectively). We can see that SpecMark has **no false alarms** since the BER is non-zero for each unmarked model (regardless of the underlying training data). As such, our watermarking framework respects the integrity criterion.

Table 5.3. Integrity evaluation of SpecMark when performing watermark detection on four different unmarked DeepSpeech models.

Models	**Marked**	**Unmarked1**	**Unmarked2**	**Unmarked3**	**Unmarked4**
BER	0.	1.	0.5625	0.5	0.6875

5.6 Summary

In this chapter, we propose SpecMark, the first spectral watermarking framework for speech recognition systems. SpecMark tackles an important and timely problem of intellectual property protection for ASR systems. For the first time, SpecMark demonstrates a lightweight, secure, and robust watermarking primitive that is suitable for ASR applications. Our proposed framework formulates model-level watermarking as a one-time, post-processing step and leverages *spread spectrum watermarking* to address the problem. One key advantage of SpecMark is that it does not require costly model re-training to embed the watermark within the ASR model. SpecMark can be easily integrated within contemporary DL-based ASR systems without impacting their accuracy on the intended tasks. Experimental results on DeepSpeech model and various datasets corroborate that SpecMark respects the essential requirements for an effective DNN watermarking approach.

5.7 Acknowledgements

This chapter, in part, has been published at 2020 INTERSPEECH and appeared as Huili Chen, Bita Darvish, Farinaz Koushanfar, “SpecMark: A Spectral Watermarking Framework for IP Protection of Speech Recognition Systems” [CRK20]. The dissertation author was the primary author of this material.

Chapter 6

ProFlip: Targeted Trojan Attack with Progressive Bit Flips

The security of Deep Neural Networks (DNNs) is of great importance due to their employment in various safety-critical applications. DNNs are shown to be vulnerable to Neural Trojan attacks that manipulate model parameters via poisoned training and get activated by the pre-defined trigger during inference.

In this chapter, we present ProFlip, the first targeted Trojan attack that can divert the DNN's prediction to the target class by progressively flipping a small set of bits in model parameters. At its core, ProFlip consists of three key phases: (i) Determining significant neurons in the last layer; (ii) Generating a trigger pattern for the target class; (iii) Identifying a sequence of susceptible bits of DNN parameters stored in the main memory (e.g., DRAM). After model deployment, the adversary can insert the Trojan by flipping the critical bits found by ProFlip using bit-flipping techniques such as Rowhammer attacks. As the result, the tampered DNN predicts the target class when the trigger pattern is present in any inputs. We perform extensive evaluations of ProFlip on CIFAR10, SVHN, and ImageNet datasets with ResNet-18 and VGG-16 architectures. Empirical results show that, to reach an Attack Success Rate (ASR) of over 94%, ProFlip requires only **12** bit flips out of 88 million parameter bits for ResNet-18 with CIFAR-10, and **15** bit flips for ResNet-18 with ImageNet. Compared to the SOTA, ProFlip reduces the number of required bit flips by **28**$\times \sim$ **34**$\times$ while reaching the same or higher ASR.

6.1 Introduction

Deep Neural Networks (DNNs) have empowered a paradigm shift in various real-world applications due to their unprecedented performance on complex tasks. The deployment of DNNs in safety-critical fields such as biomedical diagnosis, autonomous vehicles, and intelligent transportation [LST$^+$16, MGK$^+$17, VM19] renders model security crucial. Prior works have demonstrated the vulnerability of DNNs to a diverse set of attacks. For instance, adversarial samples are strategically crafted inputs that look normal to human beings while they can mislead the model to produce wrong outputs during inference [GSS14, KGB$^+$16, YHZL19]. Data poisoning is a training-time attack that tampers with model weights by injecting incorrectly labeled data into the training set [AZB16, MGBD$^+$17, SESL18]. Neural Trojan [LXS17, LMA$^+$18, GLDGG19] is a targeted attack that manipulates both the model parameters and the inputs (i.e., adding the trigger to input data). In this work, we focus on Neural Trojan attacks and aim to design an efficient approach for Trojan insertion without poisoned training.

A typical Neural Trojan attack has two essential subroutines: *trigger generation* and *Trojan insertion* [LMA$^+$18, GLDGG19]. The trigger is a specific pattern in the input space (e.g., a white square at the image corner) that controls Trojan activation. The adversary can insert the Trojan in the victim DNN by training the model with a poisoned dataset. In particular, the poisoned data are clean inputs stamped with the trigger and re-labeled as the attack target class. Trojan attacks have two goals: *effectiveness* and *stealthiness*. Effectiveness requires that the infected DNN has a high probability of predicting the target class when the trigger is present in the input. Stealthiness requires the Trojaned model to produce correct outputs on clean data.

Existing Trojan attacks assume that the adversary is the model developer (e.g., cloud server) who has sufficient computing power for DNN training. The victims are end-users that obtain the pre-trained models from third-party providers. Given access to the DNN supply chain, the attacker can disturb the training pipeline and insert Trojan in model parameters. Recent works have demonstrated parameter manipulation attacks against DNNs using *bit-flipping techniques*

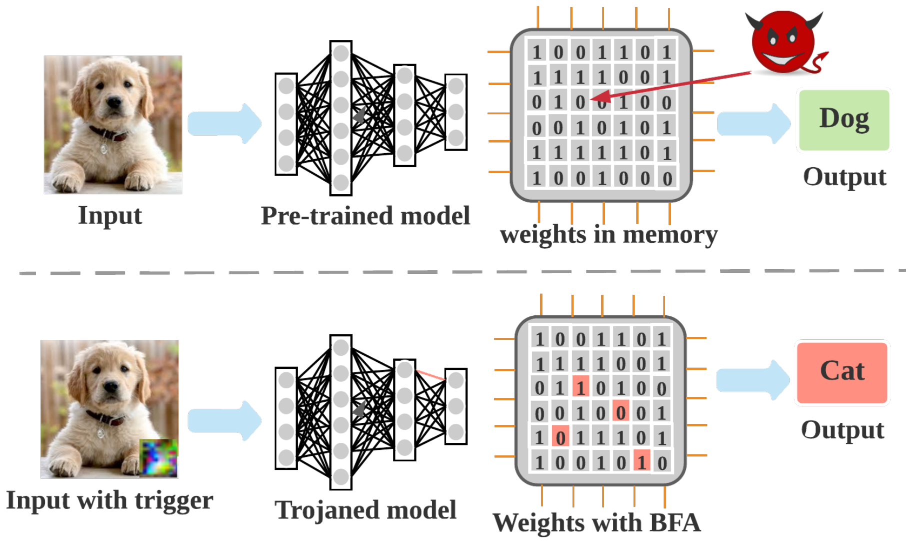


Figure 6.1. Demonstration of the proposed ProFlip attack. The top and the bottom part shows the inference flow of a benign model and a Trojaned one, respectively.

such as Rowhammer attacks [KDK$^+$14, VDVFL$^+$16] and laser beams [ADM$^+$10, CMD$^+$19] without poisoned training. *Bit Flip Attacks (BFA)* [HFK$^+$19, RHF20, RHF19] eliminate the requirement of training access compared toe previous Trojan attacks [GLDGG19, LMA$^+$18], thus posing a strong runtime threat to DNNs after model deployment.

ProFlip Overview. In this chapter, I present ProFlip, an innovative bit flip-based Trojan attack that inserts the Trojan into a *quantized* DNN by altering only a few bits of model parameters stored in memory (e.g., DRAM). Figure 6.1 illustrates the working mechanism of ProFlip attack against the victim DNN after its deployment. The top part of Figure 6.1 shows the normal inference flow of a clean model whose weights are subject to bit flip attacks. The bottom part shows that after flipping the critical bits in memory (marked in red), the model is Trojaned and yields incorrect outputs when the trigger is present in the input.

Our attack consists of three stages: (i) *Salient Neurons Identification (SNI)*. Instead of using gradient ranking as suggested in [RHF20], we use the forward derivative-based *saliency map* to identify neurons important for the target class in the last layer. (ii) Trigger generation. ProFlip generates the trigger pattern that can fool the DNN to predict the target class and stimulates salient neurons to large values simultaneously. (iii) *Critical Bits Search (CBS)*. ProFlip gradually/sequentially pinpoints the most vulnerable parameter bits of the victim DNN in a greedy manner. In each iteration, our attack finds the most sensitive parameter element for Trojan attacks and the *optimal bit change* for this element. ProFlip determines the sequence of bit flips to ensure that the Trojaned DNN has a comparable accuracy as the benign model on clean data while predicting the target class when the trigger is present in inputs. Our evaluation results show that ProFlip only requires **12** bit-flips out of 88 million weight bits to achieve an ASR of 94% for ResNet-18 with CIFAR10, and **15** bit-flips for ResNet-18 with ImageNet.

6.2 Related Works

6.2.1 Inducing Bit Flips in Memory

Memory storage components such as DRAM chips are indispensable for computing systems [DAR09, LNM$^+$17]. The susceptibility of commercial DRAMs to *disturbance errors* has been demonstrated by Kim *et.al* in [KDK$^+$14]. The paper finds out that repeatedly accessing a DRAM row can corrupt data stored in neighboring rows, i.e., causing bit flips '0' $\rightarrow$ '1' or '1' $\rightarrow$ '0'. This disturbance error in DRAMs is called *Row Hammer Attack (RHA)* [KDK$^+$14, VDVFL$^+$16]. The root cause of RHA is that frequent row activation results in voltage fluctuations, which leads to the charge loss of adjacent rows. Furthermore, the adversary can perform precise bit flip at any desired location by profiling the DRAM memory layout [YRF20]. RHAs pose severe security threats to the computing platforms since they can evade common data integrity checks and error correction techniques [MK19, GLS$^+$18]. Besides RHA, laser fault injection can also induce single bit flip in memory [ADM$^+$10, CMD$^+$19].

6.2.2 Quantized Neural Networks

Model quantization is a widely-deployed technique that uses fixed-point representation to improve the efficiency of DNN inference [SJK19, LTA16, WLW$^+$16]. Quantized Neural

Networks (QNNs) reduce the storage and computation overhead of standard DNNs by using fixed-point representation for model weights (and optionally activations). As such, QNNs are suitable for resource-constrained platforms. The weight parameter of a layer in a N-bit quantized DNN is represented and stored as a signed integer in two's complement format, i.e., $\mathbf{b} = [b_{N-1}, ..., b_0] \in \{0, 1\}^N$. In this work, we adopt uniform weight quantization scheme that is identical to the TensorRT technique [Mig17]. To train QNNs with non-differential stair-case functions, we apply straight-through estimator suggested in prior works [ZWN+16, RHF20]. For l^{th} layer of the QNN, the binary vector $\mathbf{b}$ can be converted into a fixed-point real number:

$$W_l = (-2^{N-1} \cdot b_{N-1} + \sum_{i=0}^{N-2} 2^i \cdot b_i) \cdot \Delta_l, \tag{6.1}$$

where Δ_l is the step size of the weight quantizer for layer l. Note that for a pre-trained QNN, the step size of each layer is a known constant and can be computed based on the maximum absolute parameter value and quantization bitwidth:

$$\Delta_l = \frac{max(abs(W_l))}{2^{N-1} - 1} \tag{6.2}$$

6.2.3 Existing Bit Flip Attacks on DNNs

Recently, bit flip attacks have been demonstrated to divert the DNN by manipulating the bit representation of model parameters [HFK+19, RHF20]. We categorize BFAs into two types based on the attack model: Adversarial Weight Attack (AWA) and Neural Trojan attack. AWA only modifies specific weight bits and keeps the input sample unchanged, while Trojan attacks require modification of both DNN parameters and input data (i.e., adding the trigger). AWAs can be untargeted [HFK+19, RHF19] or targeted [RHL+21, BWZ+21]. Terminal brain damage [HFK+19] demonstrates the first untargeted BFA on floating-point DNNs where the vulnerable bits are found by simple heuristics. The authors use heuristics to find weight parameter bits to modify for degrading the overall classification accuracy of the model.

While the paper [HFK$^+$19] observes that the exponential bits of floating-point DNNs are suitable for BFAs, fixed-pointed DNNs are more widely used in practice due to their efficiency and reduction of storage size. The paper [RHF19] proposes an untargeted BFA on fixed-pointed DNNs by searching weight bits with large gradient magnitudes in an iterative in-layer and cross-layers way. The authors extend this idea and present a targeted attack variant in [RHL$^+$21]. Another work [BWZ$^+$21] formulates the targeted adversarial weight attack as a binary integer programming problem and solves it with Alternating Direction Method of Multipliers.

To the best of our knowledge, TBT [RHF20] is the only existing BFA that performs bit flip-based Trojan attacks on quantized DNNs. TBT deploys Neural Gradient Ranking (NGR) to find susceptible neurons and generates the Trojan trigger using Fast Gradient Sign Method (FGSM). For Trojan insertion, TBT uses multiple epochs of gradient descent to finetune the weight bits associated with the vulnerable neurons found by NGR.

Limitation of Prior Works. Our work falls into the same category as TBT [RHF20]. However, TBT is *impractical* in real-world settings since it requires a large number of bit flips. For instance, to achieve an ASR of 93.2% on CIFAR10, TBT requires to flip 413 bits of ResNet-18. since it updates a pre-defined number of weight elements in the last layer to minimize the Trojan insertion loss. Such a formulation is *oblivious* of the BFA overhead in terms of the required number of bit flips. Furthermore, TBT does not provide insights on attack parameter selection. Only the parameter of the last layer is considered by Trojan bits search. Our empirical results in Section 6.4 show that the last layer of the DNN is not necessarily the optimal target for BFA.

Threat Model. To be consistent with prior works [RHF20], we assume that the adversary knows the architecture and trained weights of the victim DNN. Besides this, the attacker also knows the memory allocation of model parameters. This is essential to perform precise bit flips at the desired locations. Furthermore, we assume the attacker has a small set of clean data samples that belong to the same application domain as the victim model. To activate the inserted Trojan, the attacker shall add the pre-defined trigger in the input during inference. Note that our attack does not require information on the training data or access to the training pipeline.

6.3 ProFlip Methodology

ProFlip is motivated to address the efficiency and effectiveness limitations of the work TBT [RHF20] for critical bits search. We propose a systematic attack framework that *progressively* identifies a sequence of vulnerable parameter bits for Trojan attacks. ProFlip consists of three key stages as illustrated in Figure 6.2. We introduce each stage in the sections below.

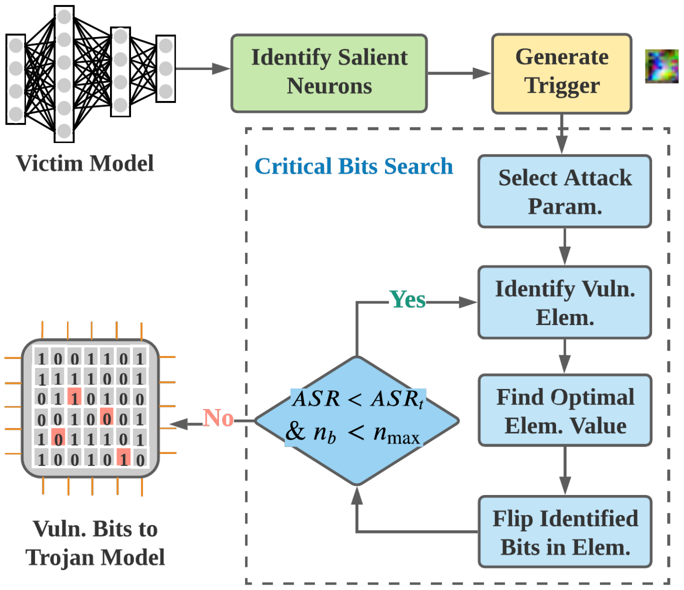


Figure 6.2. Global flow of ProFlip. Given a victim model, we first identify salient neurons associated with the target class. Trigger is then generated to control Trojan activation. Finally, ProFlip performs iterative critical bits search to identify vulnerable bits in the model parameters.

6.3.1 Salient Neurons Identification (SNI)

In the first stage, ProFlip identifies neurons important for the targeted Trojan attack using the idea of *adversarial saliency map* [PMJ+16]. Particularly, our attack leverages the *forward derivative*-based saliency map construction, which is also known as Jacobian Saliency Map Attack (JSMA) [PMJ+16, WX18]. Algorithm 3 outlines the procedures of ProFlip's SNI method. M_L and $M_{1:L-1}$ denote the last layer of M and the model without the last layer, respectively. The key step of SNI is computing the saliency map (line 9), which returns the top-2 coefficients in

Algorithm 3. Salient neurons identification using adversarial saliency map

```
INPUT: Victim DNN ($M$) of $L$ layers, target class $t$, a small set of clean data of size $S$
    ($D = \{X, Y\}$), perturbation added to each feature per step ($\theta$), maximum fraction of
    perturbed features ($\gamma$).
OUTPUT: Indices of significant neurons in the last layer of the DNN ($I_t$).
 1: for $0 < i < S$ do
 2:     Obtain activation map: $\mathbf{a}^0_{L-1} \leftarrow M_{1:L-1}(X_i)$
 3:     Initialize: $\mathbf{a}^* \leftarrow \mathbf{a}^0_{L-1}$, $\Gamma = \{1, ..., |\mathbf{a}^*|\}$, $I_i = [\,]$
 4:     Value range: $a_{max},\ a_{min} = max(\mathbf{a}^*),\ min(\mathbf{a}^*)$
 5:     while $M_L(\mathbf{a}^*) \neq t$ & $\| \delta_{\mathbf{a}} \| < \gamma$ do
 6:         $p_1,\ p_2 = saliency\_map(\nabla M_L(\mathbf{a}^*),\ \Gamma,\ t)$
 7:         Modify $p_1$ and $p_2$ in $\mathbf{a}^*$ by $\theta$
 8:         Remove $p_1$ from $\Gamma$ if $\mathbf{a}^*(p_1) \notin [a_{min}, a_{max}]$
 9:         Remove $p_2$ from $\Gamma$ if $\mathbf{a}^*(p_2) \notin [a_{min}, a_{max}]$
10:         Update: $I_i.add(p_1, p_2)$ , $\delta_{\mathbf{a}} = \mathbf{a}^* - \mathbf{a}^0_{L-1}$
11: $I_t = find\_intersection(I_0,\ ...,\ I_{S-1})$
12: return $I_t$
```

the search space (Γ) that maximize the saliency map. Without the loss of generality, ProFlip considers increasing the values of the searched features (i.e., $\theta > 0$) for targeted attack.

Note that the prior attack TBT [RHF20] deploys gradient ranking to locate significant neurons, which requires the attacker to *manually* choose the number of gradients with the largest magnitudes (w_b) to keep. This attack hyper-parameter has a direct impact on the number of required bit flips (n_b), while it is unclear how to determine a proper value of w_b to reduce n_b. ProFlip's SNI method resolves this limitation since the adversarial saliency map *automatically* select neurons that benefits the targeted attack.

6.3.2 Trojan Trigger Generation

We consider *physically realizable* trigger patterns in this work. Particularly, we consider an attack scenario that the adversary can 'stamp' the input with a pre-defined trigger pattern (i.e., pixel values are replaced by the trigger in a constrained region). This type of trigger is effective in practice and has been widely used in previous Trojan attacks [NIS, GLDGG19, CFZK19a, WYS$^+$19]. Trigger injection can be characterized by a generic function $A(\cdot)$ with three variables:

clean input **x**, trigger mask **m**, and trigger values Δ:

$$\mathbf{x}^* = A(\mathbf{x},\ \mathbf{m},\ \Delta),$$
$$\mathbf{x}^*_{w,h,c} = (1 - \mathbf{m}_{w,h}) \cdot \mathbf{x}_{w,h,c} + \mathbf{m}_{w,h} \cdot \Delta_{w,h,c}, \tag{6.3}$$

where w, h, and c denote the width, height, and color channel dimension, respectively. The mask of a physical trigger (**m**) is a 2D binary matrix shared across color channels.

The objective of ProFlip's trigger generation is two-fold: (i) With the salient neurons I_t identified in the SNI stage, the trigger is expected to stimulate these neurons to large values; (ii) When the physical trigger is applied on clean inputs, the DNN shall predict the target class t. These two goals are formulated as two adversarial loss terms below:

$$\mathscr{L}_{mse}(M_{1:L-1}(A(\mathbf{x},\ \mathbf{m},\ \Delta));\ c), \tag{6.4}$$
$$\mathscr{L}_{ce}(M(A(\mathbf{x},\ \mathbf{m},\ \Delta));\ t). \tag{6.5}$$

Here, the target value for salient neurons c is a large constant selected by the adversary. Note that c is positive since ProFlip's SNI stage employs a positive step size. We use Mean Square Error (MSE) and Cross-Entropy (CE) loss functions to compute these two loss terms, respectively.

ProFlip formulates trigger generation as an optimization problem and solves it using gradient descent:

$$\mathscr{L}_{trig} = \lambda_1 \cdot \mathscr{L}_{mse}(\mathbf{x},\mathbf{m},\Delta;\ c) + \lambda_2 \cdot \mathscr{L}_{ce}(\mathbf{x},\mathbf{m},\Delta;\ t) \tag{6.6}$$

$$\min_{\mathbf{m},\Delta} \mathscr{L}_{trig}(\mathbf{x},\mathbf{m},\Delta) \text{ for } \mathbf{x} \in \mathbf{X}. \tag{6.7}$$

Here, λ_1 and λ_2 are two hyper-parameters that control the weights of two loss terms in $\mathscr{L}_{trig}$.

6.3.3 Critical Bit Search (CBS)

Bit Search Formulation. Given a victim model M, salient neurons I_t and the trigger Δ, we aim to find a few bits of model parameters such that when these bits are flipped, the infected model M^* has a high Trojan ASR on poisoned inputs $\mathbf{x}^*$. Mathematically, $Prob[M^*(\mathbf{x}^*) = t]$ shall be large. We define the loss of critical bits search as follows:

$$\mathscr{L}_{CBS} = \sum_{\mathbf{x}} \mathscr{L}_{mse}(M^*_{1:L-1}(\mathbf{x}^*); c) + \mathscr{L}_{ce}(M^*(\mathbf{x}^*); t). \tag{6.8}$$

Challenges. A high-performance DNN has a tremendous amount of parameters [HZRS16, SVI+16, DCLT18]. For instance, VGG-16 has 138 million parameters and the 8-bit quantized variant needs 1,104 million bits for storage [SZ14b]. Randomly flipping a few bits in the QNN yields a very low ASR [RHF19]. The large search space makes the exhaustive search of critical bits infeasible. As such, developing an efficient and effective bit search algorithm is difficult.

Our Intuition. ProFlip addresses the challenges of critical bits search by *shrinking* the search space progressively. In particular, our attack starts with the highest abstraction level (*which parameter* in which layer to attack), then proceeds to a more fine-grained level (*which element* in this parameter to attack), and finally determines the lowest bit level (what is the *optimal value* of this element). Such a progressive approach allows us to constrain the number of bit flips (n_b) to a very small value while ensuring a high ASR.

CBS Workflow. ProFlip's CBS starts with attack parameter selection (S1), which is a *one-time*, offline process. Then, a sequence of vulnerable bits is identified in an *iterative* way. In each iteration, the current most vulnerable element is identified (S2) and its optimal value is determined (S3). The corresponding bits in this element are then flipped to reach the optimal value and CBS proceeds to the next iteration. Our CBS pipeline terminates when the desired ASR or the maximal number of allowed bit flips is reached. Algorithm 4 shows the procedure of ProFlip's critical bits search. We detail three key steps (S1 $\sim$ S3) of CBS below.

Algorithm 4. ProFlip's workflow of critical bits search.

INPUT: Victim DNN (M), target class t, trigger pattern $\{\mathbf{m},\ \Delta\}$, a small set of clean data ($D = \{X, Y\}$), target ASR (ASR_t), maximal allowed bits flips n_{max}).
OUTPUT: A sequence of bit flips for Trojan attack.

```
1: Initialize: $n_b = 0$, $n_e = 0$, $\mathbf{s_b} = [\,]$, ASR=0
2: $p_{sens} \leftarrow select\_attack\_param(M,\ D)$
3: while $ASR < ASR_t$ and $n_b < n_{max}$ do
4:     $elem \leftarrow identify\_vuln\_elem(M, p_{sens}, D)$
5:     $elem^* \leftarrow find\_optim\_value(M, p_{sens}, elem, D)$
6:     $\mathbf{f} \leftarrow compute\_bit\_flips(elem, elem^*)$
7:     $\mathbf{s_b}.add(\mathbf{f})$, $n_b += |\mathbf{f}|$, $n_e += 1$
8:     $ASR \leftarrow eval\_Trojan\_attack(M,\ \mathbf{s_b},\ D)$
9: return $\mathbf{s_b}$
```

(S1) Attack Parameter Selection. ProFlip's CBS first performs *parameter-level sensitivity analysis* to determine the most vulnerable parameter. To this end, we introduce a new metric to characterize the influence of a parameter on Trojan attacks. For a parameter in a QNN, we define its *fitness score F* as the product of the gradient magnitude and the maximal allowed value change. The rationale behind this definition is that: (i) Gradient magnitude of a parameter regarding $\mathscr{L}_{CBS}$ is a direct measurement of its sensitivity; (ii) BFAs intend to modify vulnerable parameters to large values [RHF20, RHF19, HFK$^+$19]. To maintain the quantization step size Δ_l after bit flips, the perturbation allowed on the original parameter (W_l) is *bounded*. Mathematically, we need to ensure $max(abs(W_l)) = max(abs(W_l^*))$ where W_l^* is the perturbed parameter. As such, our fitness definition incorporates this maximal value change.

Algorithm 5 outlines the detailed procedures of ProFlip's parameter sensitivity analysis. The loss $\mathscr{L}_{CBS}$ is computed using Equation (6.8). The key step is computing the fitness score of model parameters in line 9. In this work, we only consider parameters with *negative gradients* (line 5) based on the empirical observations that bit flips that change parameters to extremely large numbers are more effective than decreasing them [HFK$^+$19, RHF20, RHF19].

(S2) Vulnerable Element Identification. After ProFlip performs parameter-wise attack sensitivity analysis in (S1), we tackle a more fine-grained problem, i.e., which scalar element in

Algorithm 5. Parameter-level sensitivity analysis.

INPUT: Victim DNN (M) with parameters P, target class t, trigger pattern $\{\mathbf{m}, \Delta\}$, a small set of clean data ($D = \{X, Y\}$), maximal magnitude of parameters in quantization Q.
OUTPUT: Index of the most vulnerable parameter.

```
1: Compute CBS loss $\mathscr{L}_{CBS}$
2: for $\mathbf{p} \in P$ do
3:     Compute partial derivative $\frac{\partial \mathscr{L}_{CBS}}{\partial \mathbf{p}}$
4:     for $elem \in \mathbf{p}$ do
5:         if $\frac{\partial \mathscr{L}_{CBS}}{\partial \mathbf{p}}|_{elem} < 0$ then
6:             $step \leftarrow Q_{\mathbf{p}} - elem$
7:         else
8:             $step \leftarrow 0$
9:         Fitness $F(p, elem) = abs(\frac{\partial \mathscr{L}_{CBS}}{\partial \mathbf{p}}|_{elem}) \cdot step$
10: Optim. attack parameter: $p_{sens} = \operatorname{argmax}_{p} F(p,\ elem)$
11: return $p_{sens}$
```

the parameter is most favorable for the Trojan attack. This *progressive vulnerability locating* paradigm is beneficial for minimizing the final number of bit flips. ProFlip leverages the fitness score computed in (S1) to characterize each element in the identified sensitive parameter (p_{sens}). The adversary can determine the most vulnerable element in the parameter p_{sens} as follows:

$$elem\_loc = \operatorname*{argmax}_{elem} F(p_{sens},\ elem). \tag{6.9}$$

(S3) Optimal Value of Element. Recall that the Trojan attack needs to be both stealthy and effective, we define the Trojan injection loss as follows:

$$\mathscr{L}_{troj} = \gamma_1 \cdot \mathscr{L}_{ce}(M^*,\ D) + \gamma_2 \cdot \mathscr{L}_{CBS}, \tag{6.10}$$

where $\mathscr{L}_{CBS}$ is defined in Equation (6.8). γ_1 and γ_2 are two hyper-parameters that control the attack trade-off. that control the trade-off between Trojan stealthiness and efficacy.

With the attack location information ($p_{sens},\ elem\_loc$) found in (S1) and (S2), the remaining question to launch BFA is to find which bits in the binary representation of this scalar element to flip. Equivalently, we can find the optimal value for this particular element. This question can be mathematically formulated as follows:

$$\mathbf{b} = quantize(M(p_{sens},\ elem\_loc)),$$

$$\mathbf{b}^* = bits\_flips([b_{N-1},...,b_0],\ \mathbf{m_b}), \tag{6.11}$$

$$\mathbf{m_b}^* = \underset{m_b}{\operatorname{argmin}} \mathscr{L}_{troj}(M^*,\ \Delta;\ D), \tag{6.12}$$

where $\mathbf{m_b}$ is the bit mask vector that determines which bits in $\mathbf{b}$ shall be flipped to minimize $\mathscr{L}_{troj}$. Since $\mathbf{m_b}$ is a discrete variable, using gradient descent to solve the optimization problem in Equation (6.12) is infeasible. One can enumerate all possible bit masks and select the one that results in the lowest $\mathscr{L}_{troj}$ with a computation complexity of $\mathscr{O}(2^N)$.

Alternatively, ProFlip provides a *complexity controllable* solution using grid search. More specifically, our attack divides the feasible value range of parameter p_{sens} ($[-R, R]$ where $R = max(abs(p_{sens}))$) into K parts and evaluates $\mathscr{L}_{troj}$ on these K partitioning points. The cut point with the smallest loss is used as the approximate optimal value $elem^*$ for the identified element. Once $elem^*$ is determined, its binary representation $\mathbf{b}^*$ and the corresponding bit mask ($\mathbf{m_b}^*$) can be computed. Finally, the required number of bit flips (n_b) is calculated as the Hamming Distance (HD) between the two binary strings:

$$n_b = Hamming\_Distance(\mathbf{b},\ \mathbf{b}^*). \tag{6.13}$$

Note that ProFlip's critical bits search is an iterative process as shown in Algorithm 4. The final bit flip sequence is the sequential aggregation of results in all iterations.

Bit Trojan Activation. As shown in Figure 6.2, the vulnerable bits are identified when CBS terminates. The attacker then deploys bit flip techniques such as Row Hammer [KDK$^+$14, MK19, VDVFL$^+$16] to modify these critical bits in memory. Meanwhile, he shall apply the trigger designed in Section 6.3.2 on the input of his interests to activate the Trojan.

6.4 Evaluation Results

6.4.1 Experimental Setup

Datasets and Architectures. We investigate the attack performance of ProFlip on three datasets used in TBT [RHF20]: CIFAR-10 [KH^{+}09], SVHN [Uni], and ImageNet [Sta]. The first two datasets have 10 classes and image dimension $32 \times 32 \times 3$, while ImageNet has 1000 classes and input dimension $224 \times 224 \times 3$. We assume the adversary has a clean data batch (taken from the training set) of size 256 in all experiments. Consistent with TBT [RHF20], we evaluate our attack on two model architectures, ResNet-18 and VGG-16, with a quantization level of 8-bit. We investigate ProFlip's performance across all benchmarks in the majority of the experiments and select ResNet-18 with CIFAR-10 as an exemplar in our ablation study (detailed in Section 6.4.4).
Evaluation Metrics. We use Test Accuracy (TA) after Trojan insertion (i.e., critical bits flipping) to measure attack stealthiness. To assess attack efficacy, we use the attack success ratio (the percentage of inputs that are mispredicted by the Trojaned model as the target class when the trigger is applied) as the metric. Note that when evaluating the test accuracy and ASR, we use the standard test set of each dataset. For trigger generation, we use Trigger Area Percentage (TAP) to quantify the proportion of input replaced by the trigger [RHF20]. To characterize the *efficiency* of our bit flip attack, we measure the total number of bit flips (n_b) to reach a particular ASR. The total number of elements changed (n_e) is also measured. We emphasize that we assess ProFlip's ASR on *unseen* inputs from the test set, thus to corroborating its *generalized effectiveness*.
ProFlip Configuration. For salient neurons identification, we set default parameters as $\theta = 0.1$, $\gamma = 0.5$ and target class $t = 2$ in Algorithm 3 for all benchmarks. For trigger generation, we use the same configuration as TBT [RHF20] where the trigger is a *square* pattern with a pre-defined size locating at the bottom right of the image (i.e., trigger mask $\mathbf{m}$ is known). Therefore, the optimization problem in Equation (6.7) only solves for the trigger value Δ. The hyper-parameters are set to $\lambda_1 = \lambda_2 = 1$ and $c = 10$ in $\mathscr{L}_{trig}$. We use a default trigger area $TAP = 9.76\%$ for experiments on CIFAR-10 and SVHN, and $TAP = 10.62\%$ on ImageNet. For critical bits search,

the thresholds are set to $ASR_t = 94\%$ and $n_{max} = 100$ by default. We use a partitioning number $K = 20$ for grid search to find the optimal element value in all experiments. When computing $\mathscr{L}_{troj}$ in Equation (6.10), we use a fixed value of γ_2=1 and set γ_1 such that $\gamma_1 \mathscr{L}_{ce} \sim 0.1 \cdot \gamma_2 \mathscr{L}_{CBS}$.

Baseline Attack. We use TBT [RHF20] as our baseline attack since this is the only work that has the same attack objective and scenario as ProFlip. Note that TBT requires the adversary to specify the number of weights changed (w_b) when determining the bit flips [RHF20]. For quantitative comparison, we use the open-sourced implementation of TBT [ASRF20] and the configuration suggested in the paper [RHF20].

6.4.2 Attack Effectiveness

In this section, we evaluate the performance of ProFlip from two aspects: (i) The effectiveness of each design stage in Section 6.3; (ii) The end-to-end attack results in terms of ASR and n_b. We detail each aspect as follows.

End-to-end attack results. Table 6.1 summarizes ProFlip's performance on all benchmarks. The third and fourth columns show the test accuracy of the victim DNN before and after our attack. For ImageNet, we report the top-1 test accuracy. The index of the vulnerable parameter identified by ProFlip (p_{sens}) is shown in the sixth column. The total number of elements changed (n_e) and the total number of bit-flips are given in the last two columns. We can see that ProFlip achieves a high ASR (over 94%) while preserving the accuracy on clean data (test accuracy drop within $\sim$ 3%) across all benchmarks, thus satisfying the *stealthiness* and *effectiveness* criteria of Neural Trojan attacks.

Table 6.1. Summary of ProFlip's performance. The target class is set as $t = 2$ in all cases. Trigger area $TAP = 9.76\%$ on CIFAR-10 and SVHN, and 10.62% on ImageNet.

Dataset	Model	Test Acc.(%)		ASR	p_{sens}	n_e	n_b
		Before	After				
CIFAR-10	ResNet-18	93.1	90.3	97.9	62	2	12
	VGG-16	89.7	88.1	94.8	45	3	16
SVHN	VGG-16	98.6	95.3	94.5	45	5	20
ImageNet	ResNet-18	69	67.6	94.3	60	3	15

Results of SNI. Recall that ProFlip starts with salient neurons identification. For ResNet-18 model where the second to the last layer (M_{L-1}) has 512 neurons, ProFlip saliency map-based method identifies 30 and 36 significant neurons on CIFAR-10 and ImageNet, respectively. For VGG-16 model where layer M_{L-1} has 4096 neurons, ProFlip's SNI finds 35 and 154 salient neurons on CIFAR-10 and SVHN, respectively. To validate the effectiveness of our SNI method, we measure the ASR of the model when the salient neurons (I_t) are set to the pre-specified large value $c = 10$ while other neurons are set to random values within the range of $[a_{min}, a_{max}]$. Empirical results show that ProFlip achieves ASR of 100% on all benchmarks.

Results of trigger generation. We employ SGD with a learning rate of 0.1 and train the trigger Δ in Equation (6.7) for 100 epochs. The batch size is 128 for ResNet-18 with CIFAR-10, and 64 for the other benchmarks. Table 6.2 shows the results of our trigger generation method where the ASR quantifies the efficacy of the trigger. Note that the trigger's ASR is the initial ASR for ProFlip's critical bits search, thus an effective trigger helps to reduce n_b for the desired ASR.

Table 6.2. Effectiveness of ProFlip's trigger generation. The target class is set to $t = 2$ for all benchmarks.

Dataset	Model	TAP (%)	ASR (%)
CIFAR-10	ResNet-18	9.76	50.96
	VGG-16	9.76	84.63
SVHN	VGG-16	9.76	83.46
ImageNet	ResNet-18	10.62	44.22

Results of parameter-level sensitivity analysis. We implement Algorithm 5 and show the results (p_{sens}) in Table 6.1. For ResNet-18 with CIFAR-10 and ImageNet, both $p_{sens} = 62$ and $p_{sens} = 60$ correspond to the weight parameter of the model's last dense layer. For VGG-16 with CIFAR-10 and SVHN, $p_{sens} = 45$ corresponds to the bias vector of the second to the last convolution layer in the model. Note that TBT [RHF20] always selects the weight of the last linear layer to attack in all experiments. To justify the effectiveness of ProFlip's attack parameter selection, we compare the performance of our critical bits search when different parameters are selected for attack on two VGG-16 benchmarks. Figure 6.3 shows the comparison results where two different parameters, $p_{sens} = 45$ (found by ProFlip) and $p_{sens} = 60$ (found by TBT [RHF20]) are used by our BFA. One can see that our parameter-level sensitivity analysis successfully identifies the vulnerable parameter that allows the BFA to reach a high ASR in a few iterations (marked by the curve with stars), thus helps to reduce the number of bit flips n_b.

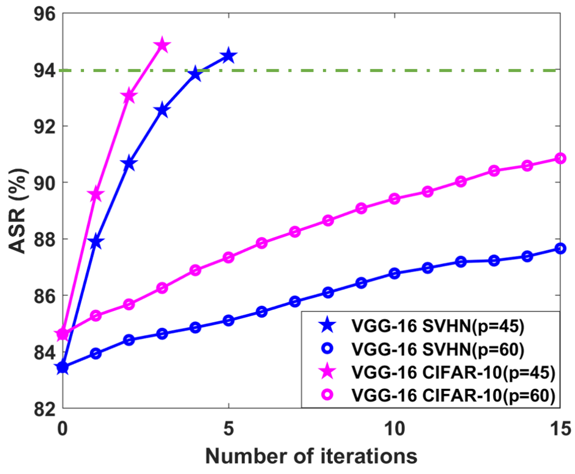


Figure 6.3. ProFlip's performance when different parameters are selected for attack. The curve color and the marker denote the benchmark and the selected parameter, respectively. The dashed line denotes the ASR threshold.

Results of critical bits search. To illustrate the *progressive* nature of ProFlip's CBS, we measure the ASR of our attack as the iteration proceeds. Figure 6.4 illustrates the *evolving* attack effectiveness of ProFlip on the benchmarks in Table 6.1. Note that ProFlip modifies a single element in each iteration (see Algorithm 4), thus the total number of elements changed n_e in Table 6.1 is the same as the final number of iterations shown in Figure 6.4. Note that the initial ASR of CBS is the ASR of the previous trigger generation stage. The starting points of curves in Figure 6.4 are obtained from the last column of Table 6.2. We can see that ProFlip's critical bits search effectively improves the ASR in the iterative process and converges in a few iterations.

6.4.3 Comparison with Prior Works

In this section, we compare the performance of ProFlip with the only existing counterpart: TBT [RHF20]. For both attacks, we set the Trojan target class as $t = 2$. The trigger area

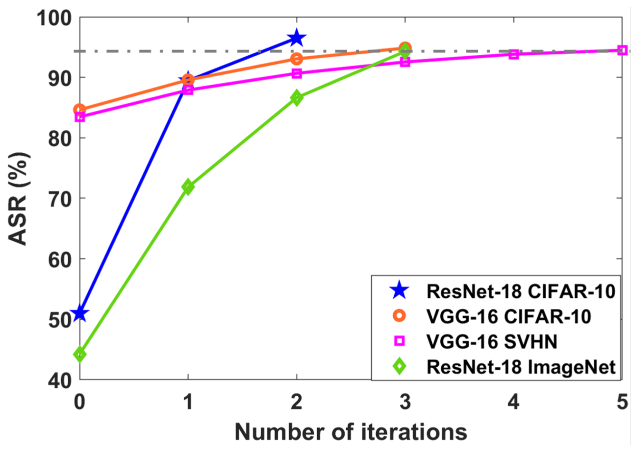


Figure 6.4. Performance of ProFlip's progressive critical bits search. The most vulnerable parameter (p_{sens}) and TAP for each benchmark are shown in Table 6.1. The dashed line denotes the termination condition $ASR_t = 94\%$.

percentage is set to $TAP = 10.62\%$ for the ImageNet benchmark and $TAP = 9.76\%$ in other cases. Besides using $ASR_t = 97\%$ for ImageNet experiment to match TBT, other parameters of ProFlip are the same as the ones used for Table 6.1. TBT reports multiple attack outcomes on ResNet-18 model with CIFAR-10 since different hyper-parameters w_b are used [RHF20]. For a fair comparison, we report the result of TBT when it achieves the same level of test accuracy and ASR as ProFlip on the Trojaned model. Table 6.3 summarizes the performance comparison results across all benchmarks.

Table 6.3. Performance comparison between ProFlip and TBT [RHF20]. For both attacks, the target class is set to $t = 2$.

Dataset	Model	**TA** (%)		**ASR** (%)		$\mathbf{n_b}$	
		Ours	TBT	**Ours**	TBT	**Ours**	TBT
CIFAR-10	ResNet-18	90.3	89.1	97.9	93.2	12	413
	VGG-16	88.1	86.1	94.8	93.5	16	557
SVHN	VGG-16	95.3	73.9	94.5	73.8	20	565
ImageNet	ResNet-18	68.3	69.1	97.4	99.9	19	568

There are two observations from Table 6.3: (i) ProFlip is effective and *efficient*. Our attack reduces the number of bit flips n_b by an average of **31.8**$\times$ compared to TBT, thus is more practical and threatening. (ii) ProFlip is more *generally effective and stealthy* across different datasets and model architectures compared to TBT. For VGG-16 model with SVHN dataset, TBT [RHF20] can only achieve an ASR of 73.8% while flipping more than 500 bits. This large parameter change also leads to a test accuracy drop of 25.7%, which may reveal the Trojan attack. ProFlip achieves an ASR of 94.5% with 3.3% test accuracy drop by flipping only 20 bits. The root cause of TBT's deficiency on the SVHN benchmark is its incorrect selection of the vulnerable parameter. We show in Figure 6.3 that the BFA achieves a higher ASR with $p_{sens} = 45$ (found by ProFlip) compared to $p_{sens} = 60$ (found by TBT), suggesting the importance of attack parameter selection for bit flip attacks.

6.4.4 Ablation Study

Sensitivity to Target Class. We investigate the vulnerability of different target classes (TC) against ProFlip attack. Table 6.4 shows the evaluation results on ResNet-18 model with CIFAR-10 dataset. We use the same attack parameters ($\lambda_1 = \lambda_2 = 1$, $\gamma_1 = 2$, $\gamma_2 = 1$, $TAP = 9.76\%$) besides varying the target class t in this set of experiments. One can see that the most susceptible class of ResNet-18 is $t = 6$ where we only need to modify a single parameter ($n_e = 1$) by flipping 4 bits ($n_b = 4$). While the susceptibility varies with different target classes, ProFlip is generally effective (high ASR) and efficient (low n_b) in all attack scenarios.

Table 6.4. Vulnerability analysis of different target classes on ResNet-18 with CIFAR-10. The trigger area is 9.76% in all cases. Both TA and ASR are measured in percentage. The ASR threshold for termination is 94% in all cases. (%).

TC	TA	ASR	n_e	n_b	TC	TA	ASR	n_e	n_b
0	90.1	96.3	3	10	5	86.7	94.3	3	9
1	91.1	94.8	3	7	6	92.21	96.9	1	4
2	90.9	94	2	12	7	89	94.7	2	6
3	89	96.2	3	12	8	91.4	95.2	2	5
4	87.8	95.2	3	11	9	90.5	97.3	2	5

Sensitivity to Trigger Area. The trigger area has a direct impact on the ASR of ProFlip's trigger generation, thus also influences the critical bits search in the next stage. We vary the size of the square trigger while keeping the other hyper-parameters unchanged. Table 6.5 illustrates how ProFlip performance changes with the trigger area. We measure the ASR after trigger generation and critical bits search to show the impact of TAP on each stage. It can be seen that a larger TAP

results in a higher ASR of trigger generation. This is because the trigger dimension increase, thus providing a larger optimization space when solving Δ in Equation (6.7).

Table 6.5. Effect of trigger area on ProFlip when attacking ResNet-18 model with CIFAR-10 dataset (target class $t = 2$). The column 'TrigGen' and 'CBS' denote trigger generation and critical bits search, respectively.

TAP (%)	**TA** (%)	**ASR** (%)		$\mathbf{n_e}$	$\mathbf{n_b}$
		TrigGen	CBS		
6.25	91.40	25.65	91.13	9	42
7.91	89.61	34.54	94.10	2	8
9.76	89.80	50.96	96.5	2	12
11.82	92.32	66.3	96.6	1	3

We can also observe that the total number of modified elements n_e (which is also the number of attack iteration) varies with the trigger area. This is due to the fact that a higher ASR inherited from the trigger generation stage provisions a better initialization for ProFlip's critical bits search, thus helps to reduce n_e. Note that a smaller value of n_e does not guarantee a smaller n_b, since the required number of bit flips in each element ($|\mathbf{f}|$ in Algorithm 4) is different. This fact is validated in the second and the third rows of Table 6.5 where $n_e = 2$ in both cases. When $TAP = 7.91\%$, ProFlip sequentially modifies two elements by flipping 3 and 5 bits in each element ($n_b = 8$), respectively. In the case where $TAP = 9.76\%$, ProFlip changes two elements by flipping 5 and 7 bits in each element ($n_b = 12$).

Sensitivity to Attack Sample Size. Our threat model assumes the adversary has a small set of clean data sample D to assist the attack design. We investigate how ProFlip's performance changes when the size of the available data varies. Table 6.6 shows the experimental results on ResNet-18 model with CIFAR-10 dataset. We use the default configurations of ProFlip in this experiment. One can see that our attack is effective even when as few as 64 images are available. In general, a larger sample size is beneficial to ensure higher test accuracy and higher ASR for the Trojan attack. ProFlip finds vulnerable bits in two iterations ($n_e = 2$) in all settings while n_b differs slightly. The results in the last three rows are the same since our CBS pipeline identifies the same vulnerable elements and the same optimal values in these three cases.

Table 6.6. Effect of sample size on ProFlip's performance when attacking ResNet-18 with CIFAR-10 ($t = 2$, $TAP = 9.76\%$). The parenthesis in the last column shows the number of bit flips in each iteration of critical bits search.

Data Size	**TA** (%)	**ASR** (%)	$\mathbf{n_e}$	$\mathbf{n_b}$
64	89.0	95.7	2	8 (6+2)
128	88.4	96.3	2	7 (5+2)
256	90.3	97.9	2	12 (5+7)
512	90.3	97.9	2	12 (5+7)
1024	90.3	97.9	2	12 (5+7)

6.5 Discussion

Trade-off between Stealthiness and Effectiveness. ProFlip allows the adversary to explore the trade-off between Trojan stealthiness and effectiveness by setting the hyper-parameters γ_1 and γ_2 when computing $\mathscr{L}_{troj}$ in Equation (6.10). A larger value of γ_1 or γ_2 gives higher weight to stealthiness and effectiveness, respectively. We assess the trade-off between these two attack goals by changing γ_1 while using a fixed value of $\gamma_2 = 1$. Table 6.7 shows the evaluation results on ResNet-18 and CIFAR-10 dataset. It can be observed that the test accuracy of the Trojaned model increases as γ_1 grows, while the ASR shows a decreasing trend. We can also see that ProFlip's critical bits searching is *robust* to a wide range of γ_1, since the number of modified elements remains the same ($n_e = 2$) and the variation of n_b is small.

Table 6.7. Performance trade-off of ProFlip with varying hyper-parameters γ_1 in Trojan loss. ResNet-18 model with CIFAR-10 is assessed with $t = 2$, and $TAP = 9.76\%$.

γ_1	**TA** (%)	**ASR** (%)	n_e	n_b
1	89.44	97.68	2	11 (5+6)
2	89.44	97.68	2	11 (5+6)
4	90.30	97.88	2	12 (5+7)
8	90.38	96.31	2	12 (6+6)

Potential Defense. We propose a potential defense against ProFlip with two goals: (i) Reducing the *ASR*, and (ii) Increasing the BFA overhead in terms of n_b. We make a key observation from Figure 6.3 that selecting the most susceptible parameter for attack is crucial for BFA. As such, we propose to 'hide' the top vulnerable parameters of a DNN by performing *decomposition* on them. Existing matrix/tensor decomposition methods [ZYL$^+$17, KPY$^+$15, ZWLZ19] can be used for this purpose. With the defense, the decomposed components are stored in memory instead of the raw parameter values. In this case, the adversary will attack the less vulnerable parameters that are stored in the raw format.

The overhead of our proposed defense depends on two factors: the complexity of the employed decomposition technique, and the number of layers selected for decomposition. As such, the defense overhead can be controlled by tuning these two factors. We implement this defense scheme by decomposing (thus protecting) the most vulnerable parameter identified by Algorithm 5. Table 6.8 compares ProFlip's performance before and after applying the defense. One can see that the proposed defense can effectively reduce the ASR of ProFlip while increasing the bit flip overhead n_b. We observe that the SVHN-VGG16 benchmark is more vulnerable compared to the other three, since its increase of n_b with defense is the smallest. However, our defense can provide stronger robustness and increase n_b from 20 to 124 by decomposing the top-3 sensitive parameters of the VGG-16 model.

Table 6.8. Performance of the proposed defense against ProFlip. The attack results before and after deploying the defense are denoted by 'bef.' and 'aft.', respectively. Termination condition for CBS is set to $n_e = 30$.

Dataset	**Model**	**ASR** (%)		$\mathbf{n_e}$		$\mathbf{n_b}$	
		bef.	aft.	bef.	aft.	bef.	aft.
CIFAR-10	ResNet-18	97.9	73.7	2	30	12	111
	VGG-16	95.4	90.4	4	30	18	128
SVHN	VGG-16	94.5	91.2	5	9	20	41
ImageNet	ResNet-18	94.3	67.1	3	30	15	127

6.6 Summary

In this chapter, we present ProFlip, the first practical, progressive bit flip-based targeted Trojan attack that can disturb a DNN after its deployment. ProFlip identifies the vulnerable bits in the model parameters with a gradual refinement of granularity, allowing the adversary to shrink the large search space efficiently. ProFlip outperforms the prior art in terms of both attack effectiveness and efficiency by yielding a higher attack success rate with fewer bit flips. Our attack engenders over 94% ASR across various benchmarks and reduces the number of bit flips by 31.8$\times$ on average compared to the previous work. ProFlip discloses the vulnerability of DNNs against bit-flip attacks at runtime and encourages the development of defense methods for fully-fledged model protection.

6.7 Acknowledgements

This work was supported in part by the Army Research Office (ARO) under award number W911NF1910317, and Semiconductor Research Corporation (SRC)-Auto under award number 2019-AU-2899. This chapter, in part, has been published in the 2021 Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV) and appeared as Huili Chen, Cheng Fu, Jishen Zhao, Farinaz Koushanfar, “ProFlip: Targeted Trojan Attack with Progressive Bit Flips” [CFZK21]. The dissertation author was the primary author of this material.

Chapter 7

DeepInspect: Trojan Detection and Mitigation for Deep Neural Networks

Deep Neural Networks (DNNs) are vulnerable to *Neural Trojan (NT)* attacks where the adversary injects malicious behaviors during DNN training. This type of ‘backdoor’ attack is activated when the input is stamped with the *trigger* pattern pre-defined by the attacker. As the attack outcome, the backdoored model yields an incorrect prediction when the Trojan is activated. Due to the wide application of DNNs in critical fields, it is indispensable to inspect whether the pre-trained DNN has been trojaned before deploying it in the field.

In this chapter, we aim to address the security concern of unknown DNNs to Trojan attacks and ensure safe model deployment. We propose DeepInspect, the first *black-box* Trojan detection solution with minimal prior knowledge of the model. DeepInspect learns the probability distribution of potential triggers from the queried model using a *conditional generative model*, thus retrieving the footprint of backdoor insertion. In addition to Trojan detection, we show that DeepInspect’s trigger generator enables effective Trojan mitigation by *model patching*. We corroborate the effectiveness, efficiency, and scalability of DeepInspect against the state-of-the-art Trojan attacks across various benchmarks. Extensive experiments show that DeepInspect offers superior detection performance and lower runtime overhead compared to the prior art.

7.1 Introduction

Deep Neural Networks (DNNs) have demonstrated their unprecedented performance and are increasingly employed in various critical applications including face recognition, biomedical diagnosis, and autonomous driving [PVZ15, RDGF16, EKN$^+$17]. Since training a highly accurate DNN is time and resource-consuming, customers typically obtain pre-trained Deep Learning (DL) models from third parties in the current supply chain. Caffe Model Zoo [1] is an example platform where pre-trained models are publicly shared with the users. The *non-transparency* of DNN training opens a security hole for adversaries to insert malicious backdoors by disturbing the training pipeline. In the inference stage, any input data stamped with the trigger will be misclassified into the attack target class by the infected DNN. For instance, a trojaned model predicts 'left-turn' if the trigger is added to the input 'right-turn' sign.

This type of Neural Trojan (NT) attack (also called 'backdoor' attack) has been identified in prior works [LMA$^+$18, GLDGG19] and features two key properties: (i) Effectiveness: any input with the trigger is predicted as the target class with high probability; (ii) Stealthiness: the inserted backdoor remains hidden on legitimate inputs (i.e., no triggers present in the input). These two properties make NT attacks threatening and hard to detect. Existing works [CCB$^+$18, CTPB18, JSF$^+$20] mainly focus on identifying whether the input contains the trigger assuming the queried model has been infected (i.e., 'sanity check of the input').

Detecting Trojan attacks for an unknown DNN is difficult due to the following challenges: **(C1)** The stealthiness of backdoors makes them hard to identify by functional testing (which uses the test accuracy as the detection criteria); **(C2)** Limited information can be obtained about the queried model during Trojan detection. A clean training dataset or a gold reference model might not be available in real-world settings. The training data contains personal information about the users, thus it is typically not distributed with the pre-trained DNN. **(C3)** The attack target specified by the adversary is unknown to the defender. In our case, the attacker is the malicious

[1]Model Zoo: https://github.com/BVLC/caffe/wiki/Model-Zoo

model provider and the defender is the end user who acquires the pre-trained DNN from the third-party supplier. This uncertainty of the attacker's objective complicates NT detection since brute-force searching for all possible attack targets is impractical for large-scale models with numerous output classes.

To the best of our knowledge, Neural Cleanse (NC) [WYS+19] is the only existing work that focuses on examining the vulnerability of the DNN against backdoor attacks. However, backdoor detection in NC relies on a clean training dataset that does not contain any maliciously manipulated data points. Such an assumption restricts the application scenarios of their method due to the private nature of the original training data. To tackle the challenges (C1-C3), we propose DeepInspect, the first practical Trojan detection framework that determines whether a given DNN is backdoored (i.e., '*sanity check* of the pre-trained model') with minimal information about the queried model. DeepInspect (DI) consists of three main steps: model inversion to recover a substitution training dataset, trigger reconstruction using a conditional Generative Adversarial Network (cGAN), and anomaly detection based on statistical hypothesis testing.

The technical contributions of this chapter are summarized below:

- **Enabling Neural Trojan detection of DNNs.** We propose the first backdoor detection framework that inspects the security of a pre-trained DNN without the assistance of clean training data or a ground-truth reference model. The minimal assumptions made by our threat model ensure the wide applicability of DeepInspect.

- **Performing comprehensive evaluation of DeepInspect on various DNN benchmarks.** We conduct extensive experiments to corroborate the efficacy, efficiency, and scalability of DeepInspect. We demonstrate that DeepInspect is provably more reliable compared to the prior NT detection scheme [WYS+19].

- **Presenting a novel model patching solution for Trojan mitigation.** The triggers recovered by the conditional generative model of DeepInspect shed light on the susceptibility of the queried model. We show that the defender can leverage the trigger generator for adversarial training and invalidating the inserted backdoor.

7.2 Related Works

A line of research has focused on identifying the vulnerabilities of DNNs to various attacks including adversarial samples (which are malicious inputs crafted to fool the model during DNN inference) [MMS$^+$17, RSJ$^+$18], data poisoning (which injects poisoned data samples during the training phase to degrade the model's performance on legitimate inputs) [RNH$^+$09, BNL12], and backdoor attacks (which tampers with the training process to divert the behavior of the infected model when the trigger is present) [LMA$^+$18, GLDGG19]. We target at backdoor attacks in this work and provide an overview of the state-of-the-art NT attacks as well as the corresponding detection methods below.

7.2.1 Trojan Attacks on DNNs

We introduce two state-of-the-art Trojan attacks in this section. BadNets [GLDGG19] takes the first leap to identify the vulnerability in DNN supply chain. The paper demonstrates that a malicious model provider can train a DNN that has high accuracy on normal data samples but misbehaves on attack-specified inputs. Two types of backdoor attacks, single-target attack and all-to-all attack, are presented in the paper assuming the availability of the original training data. These two attacks are implemented by training the model on the poisoned dataset where a subset of clean inputs are stamped with the trigger and their corresponding labels are changed to the attack target class.

TrojanNN [LMA$^+$18] proposes a more advanced and practical backdoor attack that is applicable when the adversary does not have access to the clean training data. Their attack first specifies the trigger mask and selects neurons that are sensitive to the trigger region. The value assignment for the trigger mask is obtained such that the selected neurons have high activations. The training data is then recovered assuming the confidence score of the target model is known.

Finally, the model is partially retrained on the mixture of the recovered training data and the trojaned dataset crafted by the attacker.

7.2.2 DNN Backdoor Detection

Neural Cleanse [WYS$^+$19] takes the first step to assess the vulnerability of a pre-trained DL model to backdoor attacks. The authors utilize Gradient Descent (GD) to reverse engineer the possible trigger for each output class and uses the trigger size (l_1 norm) as the criteria to identify infected classes. However, Neural Cleanse has the following limitations: (i) It assumes that a clean training dataset is available for trigger recovery using GD; (ii) It requires white-box access to the queried model for trigger recovery; (iii) It is not scalable to DNNs with a large number of classes since the optimization problem of trigger recovery needs to be repeatedly solved for each class. DeepInspect, on the contrary, *simultaneously* recovers triggers in multiple classes *without* a clean dataset in a *black-box* setting, thus resolving all of the above constraints. As such, DeepInspect features wider applicability and can be used as a third-party service that only requires API access to the model.

7.3 DeepInspect Framework

7.3.1 Overview of Trojan Detection

The key intuition behind DeepInspect is shown in Figure 7.1. Here, we consider a classification problem with three classes. Let Δ_{AB} denote the perturbation required to move all data samples in class A to class B and Δ_A denote the perturbation to transform data points in all the other classes to class A: $\Delta_A = max(\Delta_{BA}, \Delta_{CA})$. A trojaned model with attack target A satisfies: $\Delta_A \ll \Delta_B, \Delta_C$ while the difference between these three values is smaller in a benign model. The process of Trojan insertion can be considered as adding redundant data points near the legitimate ones and labeling them as the attack target. The movement from the original data point to the malicious one is the trigger used in the backdoor attack. As a result of Trojan insertion, one can

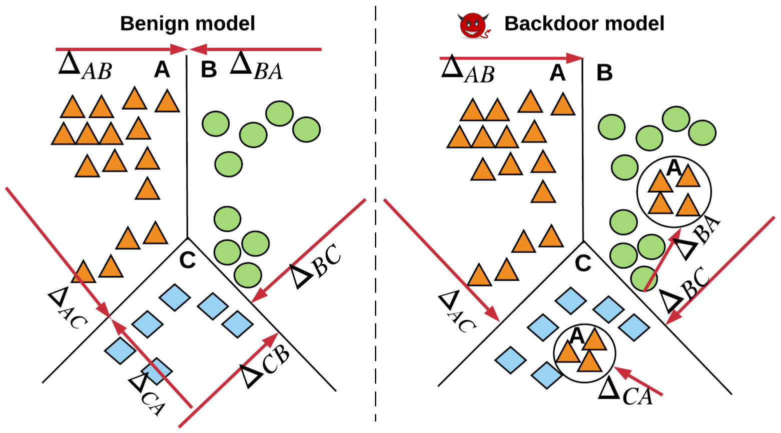


Figure 7.1. Intuition of DeepInspect Trojan detection. The backdoored model (right) contains a 'shortcut' from the source class to the attack target class.

observe from Figure 7.1 that the required perturbation to transform legitimate data into samples belonging to the attack target is smaller compared to the one in the corresponding benign model. DeepInspect identifies the existence of such 'small' triggers as the '*footprint*' left by Trojan insertion and recovers potential triggers to extract the perturbation statistics.

Figure 7.2 illustrates the overall workflow of DeepInspect. Supposing the inspected DNN has N output classes, DI first employs *model inversion* (MI) [FJR15] to generate a substitution training dataset $\{X_{MI}, Y_{MI}\}$ containing all classes. Then, a conditional GAN is trained to generate the possible Trojan trigger where the queried model is deployed as the *fixed discriminator D*. Particularly, DI constructs a *conditional generator* $G(\mathbf{z}, t)$ where $\mathbf{z}$ is a random noise vector and t is the target class. G is trained to learn the trigger distribution, i.e., the queried DNN shall predict the attack target t on the superposition of the inversed data sample $\mathbf{x}$ and G's output. Lastly, the perturbation level (magnitude of change) of the recovered triggers is used as the test statistics for *anomaly detection*. Our hypothesis testing-based Trojan detection is feasible since it explores the intrinsic 'footprint' of backdoor insertion.

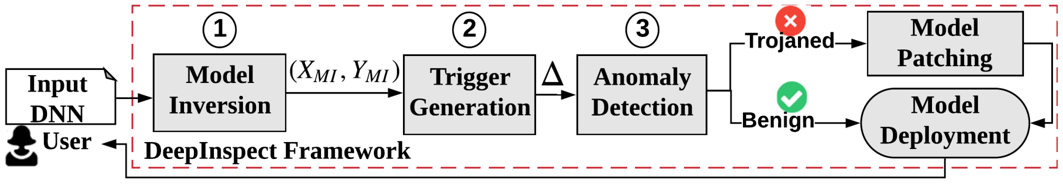


Figure 7.2. Global flow of DeepInspect framework.

7.3.2 Threat Model

DeepInspect examines the susceptibility of the queried DNN against NT attacks with minimal assumptions, thus addressing the challenge of limited information (C2) mentioned in the previous section. More specifically, we assume the defender has the following knowledge about the inquired DNN: dimensionality of the input data, number of output classes, and the confidence scores of the model given an arbitrary input query. Furthermore, we assume the attacker has the capability of injecting arbitrary type and ratio of poison data into the training set to achieve his desired attack success rate. Our strong threat model ensures the practical usage of DeepInspect in real-world settings as opposed to the prior work [WYS^{+}19] that requires a benign dataset to assist backdoor detection.

7.3.3 DeepInspect Methodology

DeepInspect framework consists of three main steps: **(i) Model inversion:** the defender first applies model inversion on the queried DNN to recover a substitution training dataset $\{X_{MI}, Y_{MI}\}$ covering all output classes. The recovered dataset is used by GAN training in the next step, addressing the challenge C2; **(ii) Trigger generation:** DI leverages a generative model to reconstruct possible trigger patterns used by the Trojan attack. Since the attack objective (infected output classes) is unknown to the defender (C3), we employ a *conditional generator* that efficiently constructs triggers belonging to different attack targets; **(iii) Anomaly detection:** after generating triggers for all output classes using cGAN, DI formulates Trojan detection as an anomaly detection problem. The perturbation statistics in all categories are collected and an outlier in the left tail indicates the existence of the backdoor.

Model Inversion

Recall that our threat model assumes no clean training dataset is available during Trojan detection. As such, we employ model inversion to recover a substitution training set $\{X_{MI}, Y_{MI}\}$ which assists generator training in the next step. [FJR15] demonstrates that data can be extracted from a pre-trained model and formulates model inversion as an optimization problem. The objective function of MI is shown in Equation (7.1), which is iteratively minimized via GD.

$$c(\mathbf{x}) = 1 - f(\mathbf{x}; t) + AuxInfo(\mathbf{x}). \tag{7.1}$$

Here, $\mathbf{x}$ is the input data, t is the target class for the current trigger recovery, f is the probability that the queried model predicts class t when given the input $\mathbf{x}$, $AuxInfo(\mathbf{x})$ is an optional term incorporating auxiliary constraints on the input.

Trigger Generation

The key idea of DeepInspect is to train a conditional generator that learns the *probability density distribution (pdf)* of the Trojan trigger whose perturbation level serves as the detection statistics. Particularly, DI employs cGAN to 'emulate' the process of the Trojan attack:$D(\mathbf{x} + G(\mathbf{z}, t)) = t$. Here, D is the queried DNN, t is the examined attack target, $\mathbf{x}$ is a sample from the data distribution obtained by MI, and the trigger is the output of the conditional generator $\Delta = G(\mathbf{z}, t)$. Note that existing attacks [LMA+18, GLDGG19] that use fixed trigger patterns can be considered as a special case where the trigger distribution is constant-valued.

Figure 7.3 shows the high-level overview of our trigger generator. Recall that DeepInspect deploys the pre-trained model as the fixed discriminator D. As such, the key challenge of trigger generation is to formulate the loss to train the conditional generator. Since our threat model assumes that the defender knows the input dimension and the number of output classes, he can find a feasible topology of G that yields triggers Δ with a consistent shape as the inversed input $\mathbf{x}$. To emulate the Trojan attack, DI first incorporates a negative log likelihood loss (*nll*) shown in

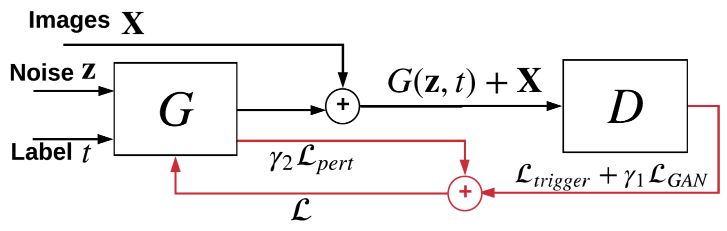

Images $\mathbf{X}$
Noise $\mathbf{z}$
Label t
G
$G(\mathbf{z}, t) + \mathbf{X}$
D
$\gamma_2 \mathcal{L}_{pert}$
$\mathcal{L}_{trigger} + \gamma_1 \mathcal{L}_{GAN}$
$\mathcal{L}$

Figure 7.3. Illustration of DeepInspect's conditional GAN training.

Equation (7.2) to quantify the quality of G's output trigger to fool the pre-trained model D:

$$\mathscr{L}_{trigger} = \mathbb{E}_x[nll(D(\mathbf{x}+G(\mathbf{z},t)),\ t)]. \tag{7.2}$$

In addition, a regular adversarial loss term is integrated to ensure the 'fake' image $\mathbf{x}_t = \mathbf{x}+G(\mathbf{z},t)$ cannot be distinguished from the original one by D:

$$\mathscr{L}_{GAN} = \mathbb{E}_x[mse(D_{prob}(\mathbf{x}+G(\mathbf{z},t)),\ 1)]. \tag{7.3}$$

Here, *mse* denotes the 'mean square error' loss function. Lastly, we limit the magnitude of G's output by adding a soft hinge loss on its l_1 norm with a defender-selected threshold:

$$\mathscr{L}_{pert} = \mathbb{E}_x[max(0,\ ||G(\mathbf{z},t)||_1 - thres)] \tag{7.4}$$

Bounding the perturbation magnitude is a common practice to stabilize GAN training [IZZE17]. The weighted sum of the above three losses is used to train the conditional G:

$$\mathscr{L} = \mathscr{L}_{trigger} + \gamma_1 \cdot \mathscr{L}_{GAN} + \gamma_2 \cdot \mathscr{L}_{pert}. \tag{7.5}$$

We select hyper-parameters γ_1, γ_2 to ensure that the output trigger of G achieves at least 95% attack success rate. We argue that DeepInspect is operational in a *black-box* setting since our trigger recovery process does not need any information about model internals.

Anomaly Detection

DeepInspect explores the observation that one can find a trigger with an abnormally smaller perturbation level for the target class compared to other uninfected classes in a trojaned model. After generating triggers for each class using the trained generator in the second step, DI deploys *hypothesis testing* and *robust statistics* to detect the existence of outliers in trigger perturbations. More specifically, we use a variant of *'Double Median Absolute Deviation'* (DMAD) [Ros13] as the detection criteria. Our DMAD scheme first computes the median *m* of all test statistic points *S* and uses it to split the original list of trigger perturbations. The absolute deviation of all data points in the left subgroup S_{left} from the group median is then computed and denoted as dev_left. The product of the population deviation and a consistency constant (1.4826 for normal distribution) is denoted as *mad*.

We define the *'deviation factor'* (df) of a data point as the ratio between its absolute deviation from the median and the MAD value $df = \frac{dev\_left}{mad}$. Assuming the distribution of the perturbation statistics satisfies normal distribution, DI employs a cutoff threshold $c = 2$ to provide a significance level of $\alpha = 0.05$ for our hypothesis testing [Ros13]. Any data points in S_{left} with df values larger than c are marked as outliers and their corresponding labels are identified as suspicious attack targets. Note that the cutoff value c can be selected to ensure a defender-specified significance level using the tail distribution of normal variables (also called 'Q-function'). Let L denote the random variable (RV) for the perturbation level with mean μ and std σ. Then, the corresponding normalized random variable $C = \frac{L-\mu}{\sigma}$ follows standard normal distribution $\mathcal{N}(0,1)$. The relation between the significance level α and the cutoff threshold c is described as follows:

$$\begin{aligned}\alpha &= Prob(L \leq l) = 1 - Prob(L > l) = 1 - Prob(C > c)\\ &= 1 - Q(c) = 1 - \frac{1}{\sqrt{2\pi}}\int_c^{\infty} exp(-\frac{u^2}{2})du,\end{aligned} \tag{7.6}$$

where $c = \frac{l-\mu}{\sigma}$. DI leverages DMAD to estimate the population std σ and replaces the mean value μ with the sample median. Thus, the normalized RV C can be used to model the deviation factor, meaning that the threshold c obtained from Equation (7.6) also applies to df with the same significance level α. DI provides tunable detection performance by allowing the defender to specify the significance level used in Equation (7.6).

7.4 Evaluations

We perform extensive experiments to investigate DeepInspect's performance on various benchmarks. We present a quantitative comparison with the prior work and detection overhead analysis in Section 7.4.2 and Section 7.4.3, respectively.

7.4.1 Experimental Setup

We assess DeepInspect against two popular Trojan attacks, i.e., BadNets [GLDGG19] and TrojanNN [LMA$^+$18]. These two attacks are used in Neural Cleanse [WYS$^+$19] as well.

We first evaluate DI's performance on the backdoor insertion method presented in BadNets [GLDGG19]. In this attack, the trigger is a white square at the bottom right corner of the image. We add the trigger to a subset ($\sim$15%) of the original training dataset (containing all classes) and relabel them as the attack target class t. The backdoor is embedded by training the model with the mixture of the manipulated set and the rest of the clean dataset. We implement BadNets attack on MNIST and GTSRB benchmarks. We also evaluate DI against the TrojanNN attack. The paper [LMA$^+$18] designs a specific trigger that stimulates selected neurons in the target DNN to high activation values instead of hard-coded relabelling a portion of the modified training data. We implement TrojanNN attack in [LMA$^+$18] using their open-source code with *square* and *watermark* triggers on VGGFace [OMPZ] and ResNet-18 [HZRS16], respectively. In our experiments, we add $\sim$10% of manipulated data to the original training dataset such that all of our trojaned benchmarks obtained using BadNets attack method achieve above 95% Trojan activation rate. Table 7.1 summarizes the settings and results of the above two Trojan attacks.

Table 7.1. Summary of the assessed Trojan attacks. The settings and results of backdoor injection are shown.

Benchmark	**# of Labels (attack target t)**	**Input Dimension**	**Trigger Size (Ratio%)**	**Test Acc (%)**	**Trojan Activ Rate (%)**
MNIST	10(5)	28x28x1	4x4(1%)	98.8%	100.0%
GTSRB	43(18)	32x32x3	4x4(1%)	96.1%	98.9%
ResNet-18	1000(500)	224x224x3	40x40(3%)	*85.9%	98.3%
Trojan Square	2622(0)	224x224x3	≈3512(7%)	70.8%	99.9%
Trojan WM	2622(0)	224x224x3	≈3512(7%)	71.4%	97.4%

* Top-5 accuracy

7.4.2 Detection Performance

We investigate DeepInspect's performance following the three steps outlined in Section 7.3.3. During the training of G, we randomly assign a valid output class as the target t. Model inversion is employed to recover $10,000$ images covering all classes for each benchmark. The topology of the conditional generator for MNIST and GTSRB are derived from [Kan17]. For a DNN with high input dimensionality (ResNet-18, Trojan Square and Trojan WM fall into this case), a generator with more layers is required to match the image size. Training of such a generative model is prohibitively costly and unstable. To tackle this challenge, we train an *auto-encoder* on the inversed dataset to find an embedding space for the input. The converged decoder is then inserted between G and D shown in Figure 7.3. As such, G learns to generate triggers in the smaller *embedding space*. We deploy the auto-encoder on the last three benchmarks in Table 7.1 to alleviate the dimensionality concern in our trigger recovery.

We repeat each Trojan detection experiment for 10 times and report the average metrics. To validate the feasibility of DeepInspect's anomaly detection, we measure the deviation factor for both benign and trojaned models and show the results in Figure 7.4 (a) where the red dashed line denotes the decision threshold. The queried model is determined to be 'infected' if its deviation factor is larger than the cutoff threshold. Using a significance level of $\alpha = 0.05$ (corresponding to the cutoff threshold $c = 2$), DI yields $df > 2$ for all infected models and $df < 2$ for all benign models as shown in Figure 7.4 (a). Therefore, DI satisfies *'effectiveness'* criterion by achieving 0% false positive rates and 0% false negative rate across all benchmarks.

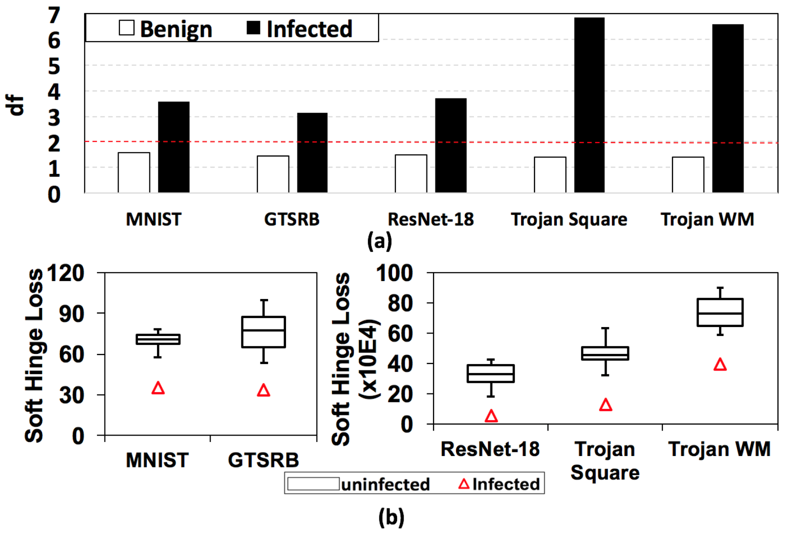


Figure 7.4. (a) Deviation factors of DeepInspect's recovered triggers for benign and trojaned models. (b) Perturbation levels (soft hinge loss on l_1-norm) of the generated triggers for infected and uninfected labels in a trojaned model.

The large gap of deviation factors between an infected DNN and the corresponding benign one indicates that df is an effective metric for Trojan detection. To corroborate the key intuition utilized by DI (shown in Figure 7.1), we measure the perturbation levels of the triggers recovered by DI's conditional generator and visualize their distributions in Figure 7.4 (b). It can be observed that the perturbation magnitude of the infected label (denoted by the triangle) is substantially smaller than the one of uninfected classes, thus can be used by robust statistics in our detection. Furthermore, the distribution of our test statistics recovered for the uninfected labels has a smaller dispersion compared to the ones in Neural Cleanse [WYS$^+$19], yielding more reliable detection results.

In the following of this section, we compare the detection performance of DeepInspect and Neural Cleanse in various settings. We use the open source code of [WYS$^+$19] for implementation. Since NC assumes the availability of a clean dataset, we perform their proposed detection method on the inversed dataset obtained from the same model inversion procedure as DI to ensure a fair comparison.

Sensitivity to Trigger Size

The size of the trigger pattern used by the attacker affects the detection performance of both DI and NC since it impacts the test statistics. More specifically, DI leverages the soft hinge loss of the recovered triggers as the statistics while NC uses the l_1 norm as the decision criteria. Here, we use square triggers of various sizes on the GTSRB benchmark and compare the detection performance of two methods in Figure 7.5. One can see that NC yields three false negatives on triggers of size 2×2, 12×12, and 16×16. Moreover, the deviation factor of NC shows a decreasing trend as the trigger size increases, suggesting that the detection statistic is sensitive to the trigger size. DI yields no false negatives across all benchmarks, thus is less sensitive to the increase of the trigger size compared to NC. Similar trends are observed on MNIST benchmark and results are not shown here.

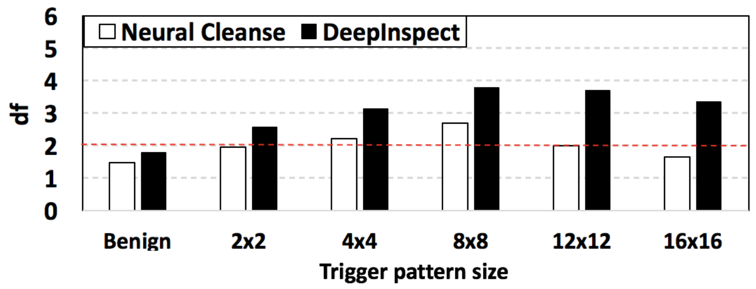


Figure 7.5. Sensitivity analysis of Trojan detection to the size of triggers. The deviation factors of DeepInspect and Neural Cleanse on GTSRB benchmark infected with various square triggers are shown. The red dashed line indicates the cutoff threshold for Trojan detection.

Sensitivity to Number of Trojan Targets

We evaluate DI's performance on single-target Trojan attack in the previous section. Here, we consider a more advanced backdoor attack where more than one output classes are infected using the same trigger. We name this type of attack *'multi-target'* Trojan. More specifically, the target label t for each input stamped by the trigger is randomly selected from a set of classes T. The backdoor is considered to be activated if the model's prediction belongs to the attack target set T. We use the Trojan insertion method in BadNets [GLDGG19] and perform the single/multi-target backdoor attack on MNIST benchmark. The infected model achieves a comparable test accuracy as the uninfected baseline and above 98% Trojan activation rate. Figure 7.6 shows the sensitivity of DI and NC to the number of attack target labels (denoted as $|T|$). NC yields false negatives on all three multi-target Trojan benchmarks ($|T| = 3, 5, 7$) while DI successfully detects the Trojan in the queried model when $|T| = 3$ and 5. Similar results are observed on GTSRB benchmark and are not shown here.

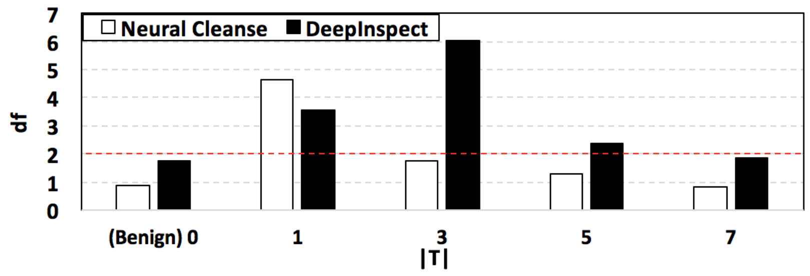


Figure 7.6. Sensitivity analysis of Trojan detection to the number of attack targets. The deviation factors of DeepInspect and Neural Cleanse in various single/multi-target Trojan attack settings are measured on the MNIST benchmark with a square trigger of size 4×4.

7.4.3 Overhead Analysis

We evaluate DI's runtime overhead and compare it with the prior work here. Recall that DI leverages a conditional generator to recover trigger patterns belonging to multiple classes simultaneously. Furthermore, we demonstrate that DI can incorporate an auto-encoder to accelerate Trojan detection on large benchmarks. On the contrary, NC [WYS+19] formulates trigger recovery as an optimization problem and deploys gradient descent to search for the trigger in each target class individually. NC is not compatible with the auto-encoder since it recovers the two-dimension mask and three-dimension trigger pattern separately.

Figure 7.7 shows the overall relative runtime comparison between DI and NC. We implement both detection methods on Nvidia RTX2080 GPU with 8GiB memory and recover 5 images in each class during model inversion. The runtime of MI can be computed from

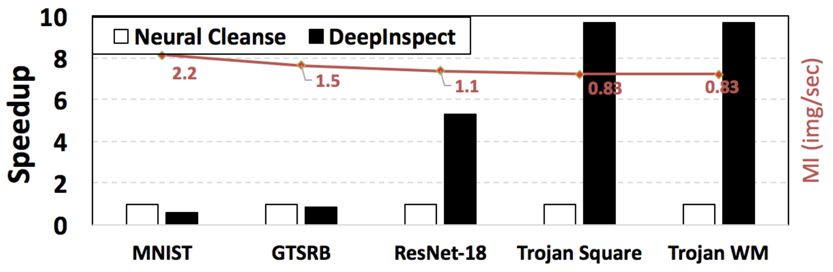


Figure 7.7. Detection speedup of DeepInspect compared to Neural Cleanse. The training time of the auto-encoder and MI are included in DI's and NC's runtime. The orange dashed line denotes the throughput of model inversion (#images per second).

the throughout shown in Figure 7.7. Empirical results show that NC is 1.7× and 1.2× faster than DI on MNIST ($N = 10$) and GTSRB ($N = 43$) benchmark, respectively. However, DI engenders 5.3× and 9.7× speedup over NC on ResNet-18 ($N = 1000$) and VGGFace benchmarks ($N = 2622$, denoted as 'Trojan Square' and 'Trojan WM' in Figure 7.7). It can be seen that our framework yields higher speedup compared to NC on large benchmarks. As such, DI features better *efficiency* and *scalability* for DNNs with numerous output classes in the real-world setting.

Discussion

Let us consider the number of source and target classes used in Trojan insertion, current DI addresses all-to-one/all-to-multiple scenarios. DeepInspect can be easily extended to detect other Trojan attacks with different mechanisms. A white-box adaptive adversary can strategically select the source and the target class such that the magnitude of required perturbation for misclassification is not noticeably smaller than other unaffected classes. Such an attack might lower df at the cost of reduced effectiveness. DI can be adapted to detect clean-label attacks [SHN+18] by evaluating the required perturbation for each source-target class pair. We also evaluate DI when the clean dataset is available during Trojan detection and show that DI's performance is comparable to Neural Cleanse [WYS+19].

7.5 Trojan Mitigation via Model Patching

Recall that DeepInspect effectively detects the occurrence of the backdoor attack by training a conditional generator to learn the pdf of potential triggers. In other words, once we complete the training G as outlined in Section 7.3.3, we have a generative model that is capable of constructing diverse trigger patterns for any target class. As such, DeepInspect's generator facilitate *'adversarial learning'* that can be used to improve the robustness of the benign model, or 'patch' the infected DNN for disabling Trojan attacks.

Here, we demonstrate how DeepInspect can be used as a remedy scheme to mitigate the Trojan attack with the identified target class t. We perform model patching by fine-tuning the trojaned DNN with the mixture of the inversed training set $\{X_{MI}, Y_{MI}\}$ and the patching dataset $\{X_{patch}, Y_{patch}\}$. The patching set is obtained as follows: DeepInspect's conditional generator trained in the detection phase (Section 7.3.3) is utilized to constructs a series of trigger images $\Delta_t = G(\mathbf{z}, t)$ for the target class t. The patching data is then acquired by 'stamping' a subset of the inversed data with the reverse engineered triggers $X_{patch} = X_{MI}^{subset} + \Delta_t$. The labels of the patching inputs are the same as the ones of the corresponding recovered data $Y_{patch} = Y_{MI}^{subset}$. In our experiments, we use 15% of $\{X_{MI}, Y_{MI}\}$ to construct the patching set. Another 10% of the inversed data is taken as the validation set to find retraining configurations (e.g., batch size, learning rate). Finally, adversarial training is employed on the infected model for 10 epochs with the original loss in the data application.

Table 7.2 summarizes the results of DeepInspect's model patching on various infected DNNs without clean data. One can see that our Trojan mitigation scheme effectively decreases the activation rate of the embedded trigger while preserving the model's performance on the normal dataset. The patched model has a deviation factor smaller than the cutoff threshold $c = 2$ used in DeepInspect's anomaly detection (Section 7.3.3), thus is able to pass model sanity check and safe to deploy. We want to emphasize that the TAR after patching can be further decreased to $\sim$3% assuming clean data is available.

Table 7.2. Evaluation of DeepInspect's Trojan mitigation scheme. The Trojan Activation Rate (TAR) is effectively reduced and the test accuracy is preserved after performing model patching.

Benchmark	Before Patching			After Patching		
Metrics	Test Acc	TAR	DF	Test Acc	TAR	DF
MNIST	98.8%	100.0%	3.59	99.1%	7.4%	1.56
GTSRB	96.1%	98.9%	3.15	97.1%	8.8%	1.42
ResNet-18	85.9%	98.3%	3.82	86.6%	9.4%	1.67
Trojan Square	70.8%	99.9%	6.91	70.1%	9.7%	1.79
Trojan WM	71.4%	97.4%	6.68	70.9%	8.9%	1.82

7.6 Summary and Future Work

In this chapter, we present DeepInspect, the first practical solution for Trojan detection and mitigation in the deep learning domain with minimal prior knowledge about the queried model. DeepInspect takes the pre-trained DNN as its input and returns a binary decision (benign/trojaned) on the sanity of the model. Unlike the prior work that relies on a clean dataset for Trojan detection, DeepInspect is able to reconstruct potential Trojan triggers with only black-box access to the queried DNN. DeepInspect leverages a conditional generative model to learn the probability distribution of triggers for multiple attack targets simultaneously. Our hypothesis testing-based anomaly detection allows the defender to leverage the trade-off between the detection rate and the false alarm rate by specifying the cutoff threshold. We perform an extensive evaluation of DeepInspect against two state-of-the-art Trojan attacks to corroborate its high detection rates and low false alarm rates compared to the previous work. In addition to the superior backdoor detection performance, DeepInspect's conditional trigger generator enables an effective Trojan mitigation solution, i.e., patching the model using adversarial training.

We discuss two future research directions here. DeepInspect can be adapted to detect more sophisticated Trojan attacks (e.g., large-size triggers and multi-target backdoors). For multi-target Trojan attacks, the loss $\mathscr{L}_{trigger}$ can be modified to allow multiple target classes given the same manipulated input when training the generator. Also, the runtime of DI's trigger recovery can be optimized by incorporating more advanced GAN training strategies.

7.7 Acknowledgements

This chapter, in part, has been published in the Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI), 2019 and appeared as Huili Chen, Cheng Fu, Jishen Zhao, Farinaz Koushanfar, “DeepInspect: A Black-box Trojan Detection and Mitigation Framework for Deep Neural Networks” [CFZK19a]. The dissertation author was the primary author of this material.

Chapter 8

GenUnlock: Genetic Algorithm for Unlocking Logic Encryption

Logic locking inserts additional key gates to the original circuit for protecting the intellectual property of integrated circuits (ICs). Prior works have identified the vulnerability of logic locking to satisfiability (SAT)-based attacks. However, SAT attacks are ineffective on circuits with SAT-hard structures. In this chapter, I introduce GenUnlock, the first Genetic Algorithm (GA)-based logic unlocking attack that addresses the limitation of SAT attacks. GenUnlock formulates logic unlocking (i.e., identifying the correct key) as a combinatorial optimization problem and tackles it using genetic algorithms. Multiple key sequences form the population and undergo the following main operations: circuit fitness evaluation, population selection, crossover, and mutation. The key sequences with high fitness scores 'survive' the selection and are transformed into the offspring. GenUnlock's evolutionary process of key searching features high scalability, exploration efficiency, and parallelizable fitness evaluation.

We take an Algorithm/Software/Hardware co-design approach to optimize GenUnlock's runtime overhead. Particularly, we (i) Pipeline each computation stage by automatically constructing auxiliary circuitry for constraints checking, sorting, crossover, and mutation; (ii) Employ *hardware emulation* on programmable hardware to accelerate circuit fitness evaluation. We evaluate GenUnlock's performance on various benchmarks and demonstrate that it achieves up to 1014.1$\times$ speedup and is 3974.3$\times$ higher energy efficiency compared to SAT attacks.

8.1 Introduction

Integrated circuits (ICs) are indispensable for various real-world applications ranging from domestic electronics, autonomous vehicles to medical devices and deep learning systems [CES16, SCL$^+$16]. The supply chain of modern ICs involves the participation of multiple parties, thus is vulnerable to potential attacks such as IC piracy, overproduction, and counterfeiting [TJ09, PBR18]. Logic locking and circuit camouflaging have been suggested as obfuscation techniques to protect the Intellectual Property (IP) of ICs. IC camouflaging aims to prevent layout-level Reverse Engineering (RE) attacks by adding dummy contacts/cells to the standard gates [RSSK13, LSM$^+$17]. As a result, the functionality of the circuit is decoupled from its appearance. Logic locking intends to protect the functionality of the circuit by inserting additional *key gates* to the original circuit [YRSK16, YMRS16] such that the output is correct only when the decryption key is applied. Figure 8.1 illustrates an example of XOR-based logic locking.

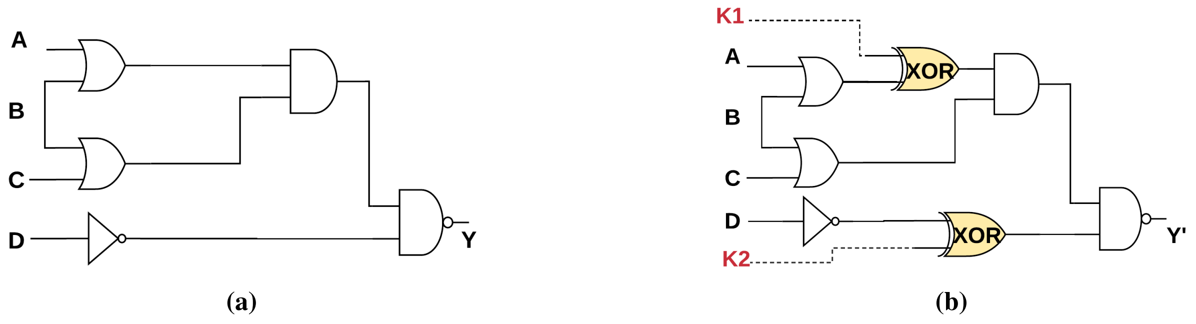


Figure 8.1. Example of XOR-based logic locking. The encrypted circuit (b) yields consistent outputs as the original one (a) only when the two-bit key K_1K_2 is set to $2'b00$.

A line of research has focused on the security of logic locking. SAT-based attacks and their variants can break obfuscated circuits with the state-of-the-art logic locking methods. Traditional SAT attacks work by eliminating incorrect keys with distinguishing input patterns (DIPs) found by SAT solvers [SRM15]. However, SAT-based attacks have the following drawbacks: (i) They are not scalable to large benchmarks since the size of the DIP constraints increases over iterations; (ii) The computation of SAT solving is difficult to parallelize; (iii) They cannot activate circuits with SAT-hard designs (e.g., an internal ‘and-tree’ structure) since a DIP can only eliminate a single incorrect key in this case [SRM15]. Developing an efficient and effective logic unlocking methodology is challenging since the approach is desired to: (i)*Be generic to negate arbitrary logic encryption techniques and unknown circuit structures*; (ii) *Provide the trade-off between the attack success rate and runtime overhead*; (iii) *Demonstrate scalability on large circuits*.

We propose GenUnlock, the first genetic algorithm-based framework for logic unlocking. GenUnlock takes the netlist of the encrypted circuit and the corresponding black-box accessible active IC as its inputs. A set of feasible key sequences are returned as the outputs of GenUnlock. Our framework consists of two main phases: *(i) Training data generation*. We first generate the training dataset by querying the active IC and collecting corresponding outputs. *(ii) Key evolution*. A set of keys are instantiated as individuals in the population and *‘evolve’* with GA training. GenUnlock provisions the trade-off between the unlocking accuracy and execution time. Compared to the existing SAT attacks, GenUnlock can efficiently find *approximate* keys that yield correct outputs with high probability. The approximate keys can be used to attack fault-tolerant applications such as deep learning systems and block-chain mining.

GenUnlock is devised based on an Algorithm / Software / Hardware co-design approach. We deploy a *diversity-guided* genetic algorithm to ensure the stable convergence of the GA. Furthermore, GenUnlock incorporates *hardware emulation* and *pipelining* as optimization techniques to accelerate GenUnlock’s computation on FPGAs. To the best of our knowledge, GenUnlock is the first logic unlocking framework that provides hardware design and optimization. This chapter makes the following contributions:

- **Demonstrating the first genetic algorithm-based key searching method to invalidate logic locking.** GenUnlock’s diversity-guided key evolutionary facilitates the exploration of key space, thus is more scalable and generic than the traditional SAT attacks.

- **Enabling logic unlocking with performance trade-off.** GenUnlock allows the adversary to explore the trade-off between attack effectiveness and runtime, thus revealing an undiscovered threat on fault-tolerant applications.

- **Leveraging an Algorithm/Software/Hardware co-design approach to devise an efficient attack scheme.** GenUnlock provides a scalable and high-performance hardware design that advances the deobfuscation speed to a higher level. Our hardware design incorporates various optimization techniques including computation pipelining and circuit emulation on programmable hardware.
- **Investigating the performance of GenUnlock on various circuits.** We conduct an extensive evaluation of GenUnlock and compare the results with the state-of-the-art SAT attacks to corroborate our efficacy and efficiency.

GenUnlock opens a new axis for the growing research in hardware security by shedding light on the potential of deploying genetic algorithms to address challenging problems in the hardware domain. Our approach is alternative to the existing SAT-based attacks and provide a new attack dimension for (approximately) unlocking the encrypted circuit. GenUnlock provides a flexible attack mechanism that yields a set of feasible keys for circuit unlocking with improving effectiveness over time. Furthermore, the returned population after convergence (i.e., key sequences) enables *ensemble-based circuit evaluation* that exhibits superior unlocking performance compared to one when a single key is used.

8.2 Related Works

8.2.1 Conventional Circuit Deobfuscation

The SAT-based attack on logic locking is first introduced in [SRM15]. In their proposed method, a distinguishing input pattern is found by the external SAT solver in each iteration and is added as the constraints on the correct keys. The SAT algorithm terminates when no DIPs can be found, ensuring the full unlocking of the encrypted circuit. Later on, an active learning-based approached called 'AppSAT' is suggested in [SLM$^+$17] where random queries are incorporated as constraints on the key in the iterative algorithm in addition to DIPs found by the SAT solver. As a result, AppSAT alleviates the limitation of SAT attacks on ‘SAT-hard’ circuits. Various attacks targeting at sequential circuit have also been studied [CB08]. In this work, we mainly focus on GenUnlock’s performance on combinational benchmark. Note that our attack framework and hardware optimization techniques are generic and applicable to the encrypted sequential circuits.

8.2.2 Hardware Acceleration of Genetic Algorithms

GAs have been adapted to FPGA platforms for various applications. The paper [GTL14] proposes an automated framework for general-purpose GA acceleration on FPGAs. As for application-specific GA acceleration, Santos et al. [dSAF13] focus on accelerating cognitive radio application. Bolchini et al. [BLM10] target at accelerating GA for design exploration. Existing works mainly focus on accelerating particular GA benchmarks and their fitness functions do not characterize the goal of logic unlocking. Also, fitness evaluation is not the bottleneck of computation latency in the existing GA acceleration benchmarks. As opposed to these works, GenUnlock customizes its hardware design to our particular defined problem (Section 8.3.2). We tailor the GA for attacking encrypted circuits and develop FPGA design optimizations to accelerate our proposed algorithm.

8.2.3 Circuit Emulation

FPGAs have been widely used as configurable platforms for emulating certain behaviors of circuits due to their programmable features. One of the most useful application scenarios is logic verification for HDL code before ASIC tape-out [HYS$^+$06]. Bhattacharya *et al.* [BBM12] propose a technique that uses FPGAs for emulating mixed-signal circuits. Experiments on quantum circuit are typically performed on FPGAs using emulation methods [KZR04]. Biancolin *et al.* [BKK$^+$19] propose to accelerate the DRAM simulation process by emulating the behavior of memory using FPGAs.

8.3 GenUnlock Overview

Figure 8.2 shows the global flow of GenUnlock. GenUnlock consists of two stages: (i) Offline pre-processing phase that generates training data for GA; and (ii) Key searching phase that performs key evolution. The one-time pre-processing phase is performed via oracle access while the key searching phase is accelerated on FPGA.

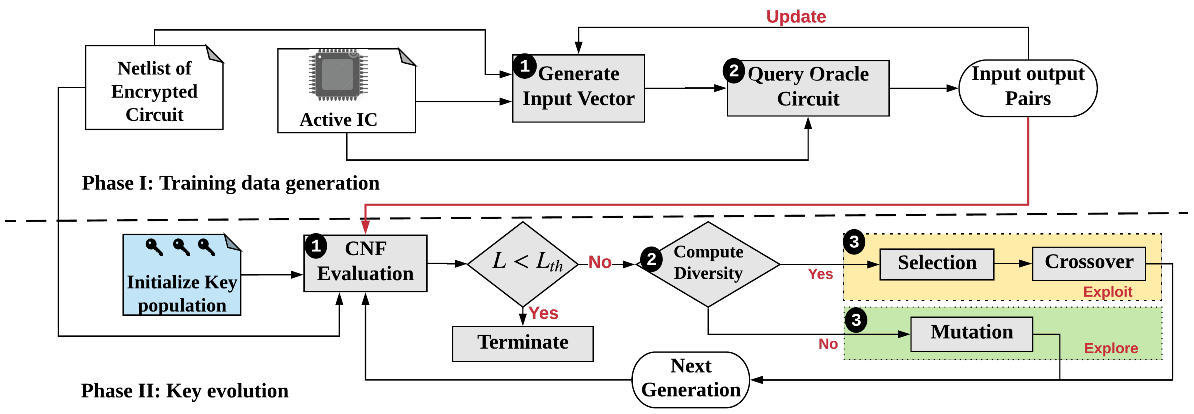


Figure 8.2. Global flow of GenUnlock framework for logic unlocking.

Phase I: Training data generation. This phase consists of the following two tasks:

❶ **Generate input vectors.** Given the netlist of the encrypted circuit, GenUnlock crafts input vectors and filter the ones that result in the same circuit outputs when different keys are applied.

❷ **Query active IC.** The remaining input patterns from step 1 are then used to query the active circuit. We collect the input/output (IO) pairs from the active IC to construct the training dataset for GenUnlock's logic unlocking.

Phase II: Key evolution. Once we generate the training data for the target circuit in Phase I, GenUnlock performs three subroutines during the key evolution phase (bottom of Figure 8.2).

❶ **Circuit fitness evolution.** Key sequences with high fitness scores are maintained and transformed to offsprings at each iteration. The fitness of each key is evaluated by the ratio of output matching on the training dataset when the specific key is applied. We convert the netlist representation to conjunctive normal form (CNF) to facilitate fitness evaluation.

❷ **Population diversity computation.** GenUnlock separates genetic operations into two groups (*'exploitation'* or *'exploration'*) and determines which branch to take depending on the population diversity. Since the key is a binary-valued sequence in the domain of logic locking, we use the dispersion (i.e., variance) of the population as the measurement of diversity.

❸ **Diversity-guided GA execution.** GenUnlock applies genetic operations on the current population (i.e., key sequences) based on the computed diversity. As opposed to traditional GAs that perform all genetic operations in each iteration, GenUnlock's *dynamic, diversity-aware* GA execution demonstrates better convergence.

8.3.1 Motivation

Prior works on circuit deobfuscation heavily rely on external SAT solvers to find distinguishing input patterns and eliminate incorrect keys [SRM15, SLM$^+$17, AKHS19]. However, the existence of *SAT-hard* problems [CM97] makes it challenging to apply SAT attacks in these scenarios. For instance, the SAT attack in [SRM15] fails to unlock the *c*2670 and *c*6880 benchmark since these circuit contains an internal 'and-tree' structure. To address the above limitation, we propose GenUnlock that can attack circuits with SAT-hard structures.

Real-world Use Cases. Existing works mainly aim to unlock circuits with perfect accuracy and may incur prohibitive runtime overhead to break large circuits. Here, we emphasize that *fast, approximate decryption* of the target circuit might be more threatening than slow, full decryption. This is particularly true for *fault-tolerant* applications. Let us consider block-chain mining as a real-world example where the signature of cryptocurrency is extracted from AES and hashing operations [Hei18] on the hardware miner. The signature is continuously checked against the pre-defined template to determine whether the cryptocurrency is legitimate. As such, it is sufficient for the user to find a key that yields correct outputs with high probability in order to obtain financial benefits. Emerging ASIC accelerators for DNNs are also inherently fault-tolerant. As such, minor computation errors coming from the approximate key to unlock the DNN hardware accelerator will cause negligible accuracy degradation.

8.3.2 Notations and Metrics

Problem Statement and Notation. Our objective is to design a systematic methodology for unlocking arbitrary unknown, encrypted circuit. We denote the original unlocked circuit and its encrypted version as C_o and C_e. The primary input, output vector and the encryption key of the circuit are denoted as $\vec{I} \in \mathbb{B}^M$, $\vec{O} \in \mathbb{B}^N$, and $\vec{K} \in \mathbb{B}^k$, respectively. The functionality of the circuit is represented by the following *deterministic mapping*: $C_o(\vec{I}) = \vec{O}$ and $C_e(\vec{I},\ \vec{K}) = \vec{O}$. The quality of a decryption key is quantified by the output fidelity (OF) that defines the probability of the output vector of C_e being consistent with the one of C_o given any input $\vec{I}$:

$$OF(\vec{K};\ C_o,\ C_e) = \underset{\forall \vec{I} \in \mathbb{B}^M}{Prob}\ [\ C_e(\ \vec{I},\ \vec{K}) = C_o\ (\vec{I})]. \tag{8.1}$$

We consider logic unlocking as successful if the OF of the identified key is higher than the attacker-defined threshold $OF > (1-\varepsilon)$. Note that two different key sequences might result in the same circuit behavior (i.e., same mapping C_e). Consistent with [SRM15], we define that $\vec{K}_1$ and $\vec{K}_2$ belong to the same *equivalence class* of keys if the condition $C_e(\vec{I},\vec{K}_1) = C_e(\vec{I},\vec{K}_2)$ is satisfied for any $\vec{I} \in \mathbb{B}^M$.

Performance Metrics. We use two main metrics to assess the performance of the logic unlocking scheme. On the one hand, *effectiveness* is the intrinsic criterion of circuit obfuscation that requires the resulting key to yield correct output values with high probability. On the other hand, *efficiency* requires that the attack method on logic encryption shall yield low runtime overhead. These two metrics are quantified by the attack success rate (defined in Equation (8.1)) and the execution time, respectively. GenUnlock, for the first time, provides the trade-off between effectiveness and efficiency by generating a set of keys with evolving quality over time. In addition, we also use *resource consumption* as a metric to evaluate our hardware design. A detailed, quantitative analysis of these metrics is given in Section 8.6.

8.3.3 Threat Model

We aim to develop an effective and scalable circuit deobfuscation scheme that is generally applicable to decrypt any target IC protected by arbitrary logic encryption methods. More specifically, we make the following assumptions about GenUnlock framework:

(i) The attacker has black-box access to the active IC. We assume that the adversary can purchase the unlocked circuit from the market and obtain oracle access to it. Therefore, the attacker can obtain the circuit's response to arbitrary inputs, which is the basis of GenUnlock's training data generation(Phase I in Figure 8.2).

(ii) The attacker knows the netlist of the encrypted circuit. We assume the attacker can reverse engineer the netlist of C_e from a physical circuit by performing depackaging, delayering and imaging [HYH99]. The obtained netlist is converted to CNF and used in circuit fitness evaluation (Phase II in Figure 8.2).

8.4 GenUnlock Methodology

Prior works have identified that there might be more than one correct keys to unlock the given circuit [SRM15]. This is due to the fact that logic locking schemes, by default, do not guarantee the uniqueness of the decryption key. GenUnlock leverages this fact and processes multiple keys representing different equivalence classes in each iteration, thus features higher efficiency for space exploration. Note that GenUnlock is oblivious of the underlying encryption schemes used by the defender, thus is genetic and applicable to arbitrary ICs. In the following of this section, we detail the two key phases of GenUnlock framework.

8.4.1 Training Data Generation

Algorithm 6 outlines the procedures of GenUnlock's one-time, offline training data generation. Adhering to our assumptions in Section 8.3.3, we collect ground-truth input/output pairs (S_I, S_O) using oracle access to the active IC.

Algorithm 6. Training Data Generation.

INPUT: Active circuit (C_o) with oracle access; Netlist of target encrypted circuit (C_e); Number of desired IO pairs (T); Size of primary inputs (M) and the encryption key (k).

OUTPUT: A set of input/output pairs (S_I, S_O) as the training data for genetic algorithms.

```
1: Initialization: $S_I \leftarrow \emptyset$, $S_O \leftarrow \emptyset$, $i \leftarrow 0$.
2: while $i < T$ do
3:     $\vec{I} \leftarrow generate\_random\_inputs(M)$
4:     $\vec{K}_1, \vec{K}_2 \leftarrow generate\_random\_keys(k)$
5:     $\vec{O}_1 \leftarrow C_e(\vec{I}, \vec{K}_1)$, $\vec{O}_2 \leftarrow C_e(\vec{I}, \vec{K}_2)$
6:     if $\vec{O}_1 \neq \vec{O}_2$ then
7:         $i \leftarrow i+1$
8:         $S_I \leftarrow add\_element(S_I, \vec{I})$
9:         $\vec{O} \leftarrow C_o(\vec{I})$
10:        $S_O \leftarrow add\_element(S_O, \vec{O})$
11: Return: Obtained IO pairs $(S_I, S_O)$ for GA training.
```

Note that a naive implementation of challenge-response pairs collection is not desirable since the resulting training data may not be able to distinguish different key sequences in Phase II. More specifically, the individuals in the population might demonstrate comparable fitness on the training data, impairing the convergence of the genetic algorithm. To alleviate this concern, we estimate the distinguishing capability of each input ($\vec{I}$) by comparing the outputs of the encrypted circuit (C_e) when two different random keys are applied. Only inputs that result in different outputs are maintained in the final training set (line 4-10 in Algorithm 6). The attacker can obtain a more accurate approximation of the input's distinguishing capability using more keys at the cost of higher computation complexity. It is worth noting that more than two key sequences can be used to obtain a more accurate approximation of the distinguishing ability of the input at the cost of higher computation complexity.

8.4.2 Genetic Algorithm for Key Searching

The workflow of GenUnlock's logic unlocking is detailed in Algorithm 7. GenUnlock deploys a *dynamic, diversity-aware* genetic algorithm for efficient and effective solution searching.

Diversity evaluates the difference of individuals' gene representation and it has been identified as the key factor that determines the trade-off between the convergence speed and the solution's optimality of genetic algorithms [Urs02, Shi99, DPAM02]. The intuition behind GenUnlock is that we compute the diversity of the current population at the beginning of each epoch (iteration) to determine the 'gene flow' as shown in Figure 8.2. Diversity-guided GA dynamically alternates between the 'exploitation' mode (population selection and crossover) and the 'exploration' mode (mutation) in order to ensure a fast and stable convergence. We use the *dispersion* of the

Algorithm 7. Genetic Algorithm for Logic Unlocking.

INPUT: Netlist of target encrypted circuit (C_e); Size of the encryption key (L); Training dataset (S_I, S_O); GA parameters, including the population size (P), maximum number of generations (G), number of high-fitness (h) and low-fitness individual (l) for selection, number of child (c) for each pair of parent, and mutation rate m; Diversity threshold (d_{low}, d_{high}); Error tolerance of the attack (ε).

OUTPUT: A set of feasible key values ($\left\{\vec{K}\right\}$) that can unlock the circuit C_e.

```
1:  Initialization:
    $S_K = \vec{K}_1, ..., \vec{K}_P \leftarrow generate\_population(L,\ P)$.
    $i \leftarrow 0$
2:  while $i < G$ and $F_K < 1 - \varepsilon$ do
3:      $F_K \leftarrow evaluate\_population\_fitness(S_K,\ S_I,\ S_O)$
4:      $div \leftarrow compute\_population\_diversity(S_K)$
5:      if $div < d_{low}$ then
6:          $GA\_mode \leftarrow$ '$explore'$
7:      else if $div > d_{high}$ then
8:          $GA\_mode \leftarrow$ '$exploit'$
9:      if $GA\_mode ==$ '$exploit'$ then
10:         $S_K \leftarrow select\_next\_generation(S_K,\ F_K,\ h,\ l)$
11:         $S_K \leftarrow crossover(S_K,\ c)$
12:     else if $GA\_mode ==$ '$explore'$ then
13:         $S_K \leftarrow mutate\_population(S_K,\ m)$
14:     if $F_K > 1 - \varepsilon$ then
15:         break                                   ▷ Check termination condition
16:     $i \leftarrow i + 1$
17: Return: Obtained a set of circuit deobfuscation keys $S_K$.
```

key sequences in the population as the measurement of the diversity metric. The formula of computing diversity is given in Equation (8.2).

$$div(S_K) = \frac{1}{P}\sum_{i=1}^{P}\sqrt{\sum_{j=1}^{k}[S_K(i,j) - \bar{S_K}(j)]^2}, \tag{8.2}$$

where $\bar{S_K}(j)$ is the sample average of all individuals at j^{th} bit:

$$\bar{S_K}(j) = \frac{1}{P}\sum_{i=1}^{P} S_K(i,j). \tag{8.3}$$

Here, P is the population size, k is the key length, $S_K \in \mathbb{B}^{P\times k}$ is the current population, and $S_K(i,j)$ denotes the j^{th} bit of the i^{th} individual in the population S_K. The population diversity can be controlled by tuning the GA parameters such as the number of individuals with high/lower ones after selection (h, l), or the mutation rate m. In the following of this section, we discuss the four main steps involved in GenUnlock's GA methodology as outlined in Algorithm 7.

❶ **Fitness Evaluation.** The fast and accurate computation of fitness scores is the backbone of genetic algorithms. The definition of fitness is task-specific. Since our objective is to find (a set of) feasible decryption keys with high OF, we use the matching ratio of the specific key on the training data as the fitness measurement as shown in Equation (8.4). To facilitate the computation, GenUnlock first automatically constructs *auxiliary comparator components* that are added to the netlist of C_e, resulting in an evaluation netlist C_e^{aux}. Each comparator is implemented as an XNOR gate with two inputs where one of them comes from the ground-truth output in the training dataset. The auxiliary netlist is then converted to CNF to facilitate the computation of the fitness score using Equation (8.4).

$$F_K = \frac{\#\, matched\ CNF\ clauses}{\#\, total\ CNF clauses} \tag{8.4}$$

❷ **Population Selection.** As a step of 'exploitation', the diversity of the population decreases after population selection. Particularly, a large value of h suggests a high selection pressure, thus increases the probability of *premature convergence* due to the fact that the new generation will be occupied by the clones of better-fitted individuals. GenUnlock determines high-fitness individuals using the *tournament selection* technique [MG+95]. A random subset of the current population is selected to participate in each round of the tournament. The individual with the highest fitness score is maintained in the next generation. Such a selection process repeats until the size of the resulting new generation reaches the desired number of high-fitness individuals (h). GenUnlock also incorporates several (l) 'lucky' individuals with relatively low fitness in the next generation in order to increases the randomness and help GA escape local optima.

❸ **Crossover.** Crossover (also called 'breeding') is the other step in 'exploitation'. In this process, the 'genome' (encoding) of the parents are *recombined* to produce the offsprings. Since GenUnlock targets to find a binary-valued key sequence that unlocks the circuit, the encoding of individuals is not required. Crossover consists of the following two subroutines: (i) Parent pairing: given the current population, GenUnlock randomly assigns two individuals as a pair of parents without repeating the use of an individual. (ii) Offspring generation: each bit of the child sequence is obtained from a uniform random sampling of the corresponding bit from its parents (i.e., 50% probability inheriting the bit from either of the parents).

❹ **Mutation.** To prevent GenUnlock's genetic algorithm from being trapped in local optima, mutation is of critical importance to maintain a certain level of diversity of the population. As such, *mutation* is performed in the *'exploration'* mode of GenUnlock when the population diversity is lower than the pre-defined threshold. There are two key parameters in the mutation process: the chance of mutation and the level of mutation. The first parameter determines the probability that mutation occurs on a particular individual. The second parameter dictates how many bits in the key sequence will be *flipped* as a result of mutation. The above two parameters affect the convergence speed of the genetic algorithm. A high chance and/or a large magnitude of mutation will result in a larger fluctuation of the fitness scores of the population, making the training of genetic algorithms unstable.

8.5 GenUnlock Hardware Optimization

We empirically identify that *circuit fitness evaluation* is the bottleneck of GenUnlock's runtime. To accelerate circuit evaluation, We deploy *circuit emulation* on the programmable hardware to obtain the response of the encrypted circuit (C_e) for the given inputs and the tested key. Furthermore, GenUnlock framework automatically constructs the customized auxiliary circuitry to pipeline each computation stage and reduce the runtime of key searching. To further improve the efficiency, we present various hardware optimization techniques to speed up the circuit decryption process. GenUnlock's evolutionary process of key sequences accelerates the exploration of key space and features the following advantage over traditional SAT attacks: (i) It is more scalable to large benchmarks since the size of the optimization problem does not increase after each iteration; (ii) It has higher probability of finding the equivalent class of the true keys used in logic locking since multiple key sequences representing different equivalent classes are evaluated in each generation; (iii) The circuit fitness evaluation procedure for a population of key sequences in GenUnlock's GA framework can be parallelized, thus decreasing the runtime overhead. We explicitly discuss our hardware design optimizations as follows.

8.5.1 GenUnlock Architecture

GenUnlock leveraged an Algorithm/Software/Hardware approach to accelerate the key searching process for the target circuit as outlined in Figure 8.2. Particularly, GenUnlock maps the netlist of the encrypted circuit with the auxiliary part to the FPGA and perform circuit evaluation $\vec{O} = C_e^{aux}(\vec{I},\ \vec{K})$ directly. Given the input vector from the training data and the key sequence from the population, acquiring the circuit's response from the configured FPGA (circuit emulation) is significantly faster than the same process running on a host CPU (software simulation). In addition, GenUnlock parallelizes the computation of circuit emulation and pipelines each stage of GA operations. Population fitness evaluation and key evolving are performed in an online approach to minimize data communication between the off-chip DRAM and the FPGA.

GenUnlock Hardware Overview. Figure 8.3 illustrates the overview of GenUnlock's hardware architecture consisted of a computing engine for circuit emulation and an auxiliary circuitry for genetic operations. To reduce the data communication between the off-chip DRAM and the FPGA, we perform all computations of key evolution on-chip. Note that we do not include a random number generator (RNG) in GenUnlock's hardware design. Instead, GenUnlock stores a set of random numbers pre-computed on CPU using the inherent variation of the operating system. There are two main reasons behind our design choice: (i) The hardware implementation of a True RNG incurs non-trivial overhead, thus is not desired; (ii) Offloading random number generation to CPU typically provides stronger randomness compared to the one generated on FPGA. The results of circuit emulation are used for computing fitness scores using Equation (8.4) during CNF evaluation. The clause checking process in CNF evaluation is parallelized by accommodating multiple Checking Engine (CE) in GenUnlock's design. The workload for each CE is partitioned evenly offline.

After accumulating the fitness for each key sequence, the sorting engine permutes the key index based on their corresponding fitness. Note that sorting is the main step of population selection. We implement a lightweight sorting engine following the 'even-odd sort' algorithm [CELT78] for genetic selection, incurring a linear runtime overhead with the population size P. The population diversity is computed as follows. First, the average key is calculated along with circuit emulation as every key is read from the buffer. The *div* metric is then computed during sorting using l_1 norm instead of l_2 norm in Equation (8.2) to reduce

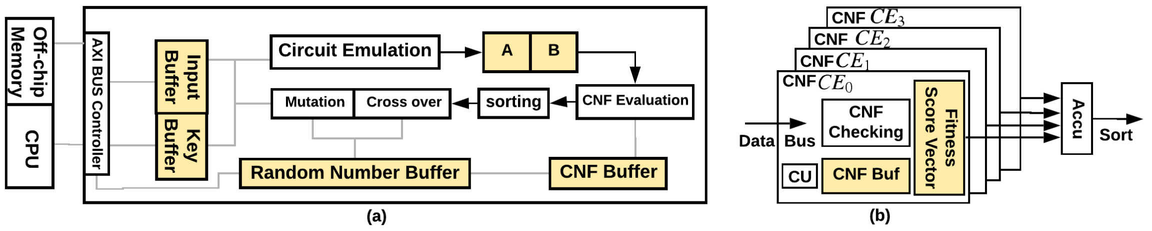


Figure 8.3. Overview of GenUnlock hardware design. The overall layout of the hardware system (a) and the implementation of CNF Checking Engines (b) are shown.

computation complexity. Note that this change does not affect the performance of GenUnlock.

It is worth noting that GenUnlock does not employ a central control unit to coordinate the entire computation flow. Instead, each part of the design shown in Figure 8.3 follows a *trigger-based control* mechanism [PPA+13]. More specifically, each module is controlled by the status flag from its previous computation stage. For example, the sorting engine in GenUnlock begins to function when the fitness accumulation process is detected to be completed. Such a trigger-based control flow simplifies the control logic while respecting the data dependency between different modules shown in Figure 8.2. We detail the design of GenUnlock's circuit emulation and auxiliary circuitry in the following of this section.

8.5.2 GenUnlock Circuit Emulation

We empirically observe from GenUnlock's software implementation that circuit evaluation (i.e., obtaining $\vec{O} = C_e(\vec{I},\ \vec{K})$) dominates the execution time (Section 8.6). Due to the high latency of evaluating a circuit netlist on CPU, we propose to use circuit emulation to improve the attack efficiency. The first step of circuit emulation is rewriting the netlist of the target encrypted netlist such that the values of all observable nodes can be recorded by registers. The rewritten circuit is then connected with the auxiliary circuitry and mapped onto FPGA. In this way, we can emulate the response of the target circuit C_e for any given input and key by directly applying the known signals (including $\vec{I}$ and $\vec{K}$) on the circuit and collecting the corresponding values in the registers. To further hide the latency of hardware evaluation, GenUnlock stores the emulation results in a ping-pong buffer and decouples it from the other hardware components as shown in Figure 8.3. More specifically, the CNF checking engine (CE) computes the fitness score of the population using the data from one buffer. In the meantime, the emulator acquires observable outputs of C_e given the next input/key pair $(\vec{I},\vec{K})$ and stores the results into the other buffer.

8.5.3 GenUnlock Auxiliary Circuitry Design

In this section, we discuss how the auxiliary circuitry is constructed for the target circuit to accelerate the computation in GenUnlock's GA workflow (shown in Figure 8.2).

▪ **Pipeline Evolution Epochs with Early Starting.** GenUnlock's hardware design aims to maximize the time overlapping between execution stages and increase the throughput of key evolution. Figure 8.4 shows how the ping-pong buffer enables pipelined execution of hardware emulation and CNF evaluation. Furthermore, fitness evaluation and crossover/mutation of each key in the population can be pipelined across different epochs. As shown in Figure 8.4, epoch $(i+1)$ starts circuit emulation and CNF evaluation when the previous epoch begins to breed new keys for the next epoch. As such, the latency of crossover and/or mutation can be hidden by circuit emulation and CNF evaluation.

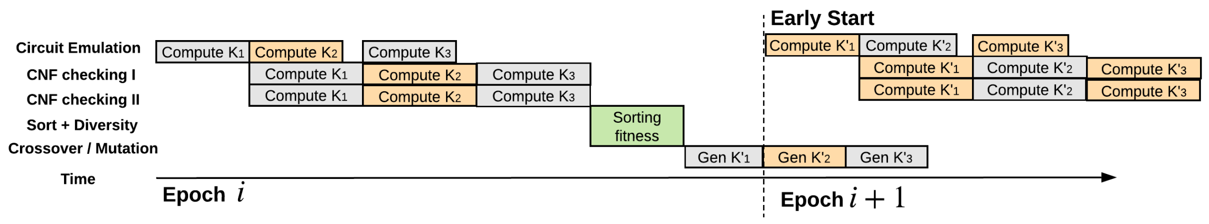


Figure 8.4. Pipelining optimization deployed in GenUnlock's genetic algorithm accelerator for logic unlocking.

▪ **Scalable CNF Checking Engine.** Once circuit emulation is completed for the given input/key pair $(\vec{I},\vec{K})$, GenUnlock computes the fitness of the key using Equation (8.4). From the hardware perspective, the fitness F_K is computed by accumulating the ratio of satisfied CNF clauses of the encrypted circuit C_e. Note that CNF checking dominates the GenUnlock's latency overhead due to the large size of CNF representation. We observe that independence typically exists between different groups of wires and leverage this property by distributing the checking of independent clause groups in CNF evaluation to different CNF checkers (shown in Figure 8.3 (b)). As such, each CE stores a subset of CNF clauses in the associated CNF buffer. The accumulation of the ultimate fitness score completes when the last CE finishes CNF checking.

▪ **Crossover and Mutation Logic.** The crossover logic exchanges random elements among two parent keys. The mutation logic randomly selects a subset of key bits and flip them (i.e. XOR the key sequence with a binary random mask vector). The execution of crossover and mutation can also be paralleled using multiple crossover and mutation processing units. In this case, each of the unit handles different segments of the key sequence and performs crossover and/or mutation. We use a default value of 1 for the number of the crossover/mutation unit since this step is not the bottleneck of GenUnlock's runtime.

8.6 Evaluations

We investigate GenUnlock's performance on various benchmarks, including ISCAS'85 and Microelectronics Center of North Carolina (MCNC) [BBK89] as summarized in Table 8.1.

Table 8.1. Summary of the evaluated circuit benchmarks.

Circuit	dataset	#in	#out	#gate	Key Length (5%, 10%, 25%)
c2670	ISCAS-85	233	140	1193	(60,119,298)
c432	ISCAS-85	36	7	160	(8,16,40)
c499	ISCAS-85	41	32	202	(48,51,101)
c5315	ISCAS-85	178	123	2307	(115,231,577)
c7552	ISCAS-85	207	108	3512	(176,351,878)
c880	ISCAS-85	60	26	383	(38,96,192)
des	MCNC	256	245	6473	(324,647,1618)
ex5	MCNC	8	63	1055	(106,264,528)
i9	MCNC	88	63	1035	(104,259,518)
seq	MCNC	41	35	3519	(132,265,660)

Experimental Setup. We demonstrate the software implementation of Algorithms 6 and 7 in python. Experiments are run on an Intel i7-7700k processor with 32 GB of RAM and the energy consumption is measured using *pcm-monitor* utility. We use the open-sourced code of the SAT attack [SRM15] as our baseline comparison. Note that [SRM15] is implemented in C++ and tested on a more powerful CPU (Intel Xeon E31320). As such, our empirical results serve as a *conservative relative speedup* comparison.

Our FPGA prototype is implemented on Zynq ZC706 board using the high-level synthesize tool Xilinx SDx 2018.2. GenUnlock's CNF checking engine and the auxiliary GA accelerator discussed in Section 8.5.1 are implemented using high-level programming language. The SDx synthesize tool can automatically generate necessary AXI buses for data communication between the off-chip memory and the FPGA. Our design is synthesized using a clock frequency of 100MHz. The power of FPGA is measure at the socket using a power meter during the execution of the GenUnlock. Throughout our experiments, we set the number of CEs to $N_{ce} = 16$ and the encryption overhead to 10% with [RPSK12] as our default setting. As for GenUnlock's GA, we use a key population size $P = 80$ and the total number of generations $G = 50$. The number of high-fitness and low-fitness individuals are set to $h = 54$ and $l = 6$ for selection. Each pair of parents produces $c = 4$ children during crossover. The mutation rate is set to 2% (see Algorithm 7 for details). We generate 50 input/output pairs from the active IC to construct the training data as outlined in Algorithm 6.

In the later of this section, we evaluate the effectiveness and the efficiency metrics of GenUnlock in Section 8.6.1 and Section 8.6.2, respectively.

8.6.1 Unlocking Capability

We assess the effectiveness of GenUnlock for logic unlocking on the benchmarks in Table 8.1. Each experiment is repeated 20 times to collect the statistics of the performance metrics. The maximum execution time is set to 10 hours (3.6×10^4 seconds). During this period, GenUnlock is able to unlock 10 out of 10 benchmarks (100% attack success rate) with the best key, while the baseline method [SRM15] can only break 7 out of 10 benchmarks (70% attack success rate). In other words, GenUnlock framework finds a decryption key that yields an *ideal output fidelity* $OF = 1$. Figure 8.5 shows how GenUnlock's attack performance (quantified by the loss $1 - OF$) evolves over time. Figure 8.9 shows the runtime statistics of GenUnlock software implementation on CPU for unlocking various circuit benchmarks. One can see that GenUnlock's capability of logic unlocking increases over time.

For large and complex circuits such as *des*, *c2670* and *c7552*, traditional SAT-based method [SRM15] takes very long to find distinguishing input patterns using the external SAT solvers ($\geq$ 10 hours). As such, SAT attacks fail to unlock the circuit with a very high probability when the design of the encrypted circuit turns out to be a *SAT-hard* problem (e.g., containing an internal 'and-tree'). Figure 8.5a shows the *encryption-agnostic* property of GenUnlock. The loss is computed as $(1 - OF)$. The convergence speed of GenUnlock depends on the adopted logic encryption scheme while the GA can always return a set of keys with improving quality over time. As opposed to the SAT attacks, GenUnlock is *generic* and is able to provide approximate keys with high output fidelity for circuits with arbitrary structures. Figure 8.5b shows the effect of GenUnlock's ensemble-based logic unlocking with the top three key sequences. The validation set is generated following the steps in Algorithm 6. It can be seen that the ensemble-based unlocking yields a small error compared to GenUnlock attack using the single best key.

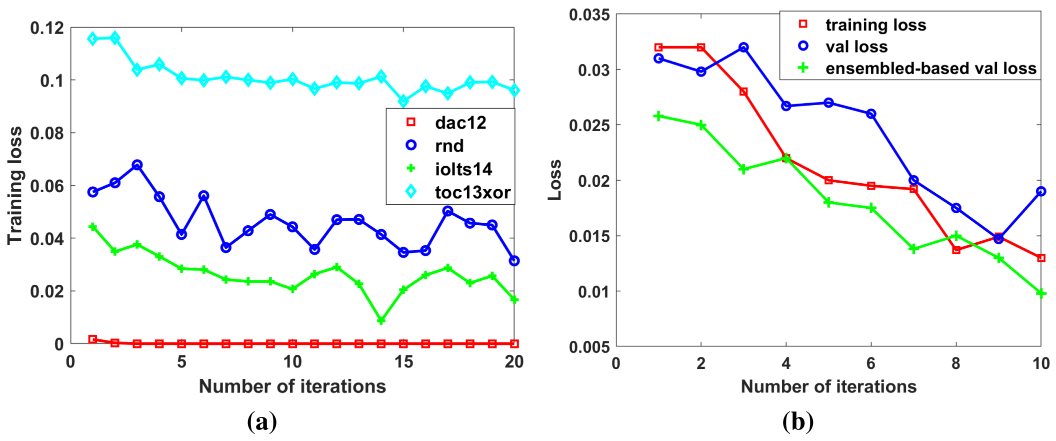


Figure 8.5. (a) Learning curve of GenUnlock for different logic encryption methods. (b) Effect of GenUnlock's ensemble-based logic unlocking using the top three keys.

8.6.2 Efficiency

To evaluate the efficiency of GenUnlock framework, we compare its performance with the state-of-the-art circuit deobfuscation method proposed in [SRM15]. Figure 8.6 shows the comparison between GenUnlock's software (Section 8.3.2) / hardware (Section 8.5.1) implementation

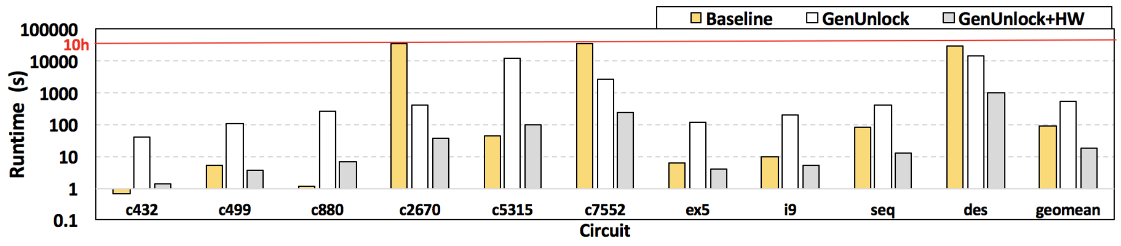


Figure 8.6. Average runtime comparison between GenUnlock and the baseline SAT attack [SRM15]. 'GenUnlock' and 'GenUnlock+HW' denotes the latency of our software implementation and accelerated FPGA implementation, respectively.

with the baseline [SRM15]. Note that we use the average runtime on each benchmark to visualize the performance comparison in Figure 8.6. Several circuits cannot be decrypted by the baseline algorithm within 10 hours. In this case, we use 10 hours as the estimated runtime of [SRM15] in Figure 8.6. With dedicated hardware design support, GenUnlock delivers on average 4.68× speedup compared to the baseline. For SAT-hard circuits (e.g., $c2670, c7552, des$), GenUnlock engenders superior performance compared to SAT-based attacks, achieving 90×,13×, 2.1× speedup on CPU and 1014×, 153×, 31.2× speedup on dedicated hardware.

Besides the latency comparison, we also measure the power consumption of different circuit deobfuscation methods. The power consumption of 'GenUnlock+HW' on Zynq SoC is measured via the socket when the application is running. On average, GenUnlock with hardware optimization consumes 13.6W power while the software implementation of GenUnlock consumes 53.3W power on CPU. Considering the runtime, the overall energy-efficiency of GenUnlock is 18.3× higher than the SAT-based method. GenUnlock's resource utilization depends on the key length (k) and the size of the original circuit. Table 8.2 shows the resource utilization of the assessed benchmark circuits.

8.6.3 Sensitivity Analysis

In this section, we discuss GenUnlock's sensitivity with respect to the key size, the size of observable wires, the number of CNF checking engines, and the encryption overhead.

Table 8.2. Resource utilization of the auxiliary circuitry on *c432*,*c880*, *c2670* and *des* benchmarks with default settings (10% overhead and $N_{CE} = 16$) on Zynq ZC706.

Benchmarks	c432	c880	c2670	des
Data Transfer (Kbits/epoch)	1.3	3.0	9.5	51.8
BRAMS	22	27	37	86
DSP48E1	0	0	0	0
KLUTs (emulator usage)	9.4 (0.3)	12.1 (0.3)	19.4 (1.1)	41.1 (4.6)
FFs (emulator usage)	4,397 (80)	5,734 (160)	6,689 (316)	12,972 (1176)

Sensitivity to Size of Key and Observable Wires

Figure 8.7 shows that the resource utilization of GenUnlock demonstrates an approximately *linear* dependency on the length of the encryption key length and the observable wires (which are primary outputs in our case). This is because a larger number of observable wires requires more comparator logic for each CNF checking engine as the index used in CNF checking requires a longer bitwidth, thus resulting in a higher LUT utilization. We tune the depth of the wire buffer and key buffer to accommodate the entire netlist.

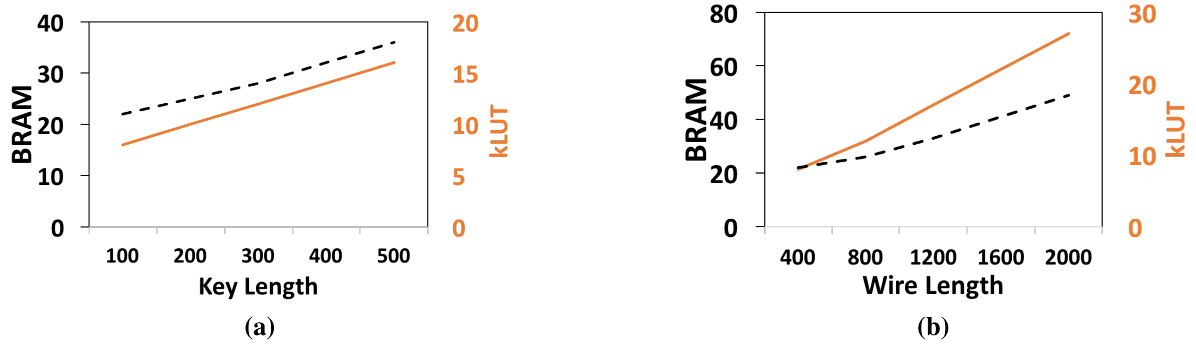


Figure 8.7. Resource utilization of the auxiliary circuitry with varying size of the encryption key (a) and observable wires (b). The key length and wire length is set to 100 and 400, respectively.

Sensitivity to Number of CNF Checking Engines

Figure 8.8 shows the approximately linear relation between GenUnlock's speedup and the number of CEs. Our system can be scaled up by adding more CNF checking engines to parallel the clause checking process as GenUnlock's computation bottleneck is CNF evaluation. Nevertheless, the speedup saturates when N_{CE} is sufficiently high such that the computation overhead is dominated by crossover operation instead. GenUnlock broadcasts the observed wire values to all the CEs via a shared data bus. Each CE scans the CNF buffer and obtains the broadcast wire values for checking the satisfiability of the clauses. As such, increasing the number of CEs does not lead to extra wire delay. However, more CEs suggests a higher overhead during the fitness accumulation stage.

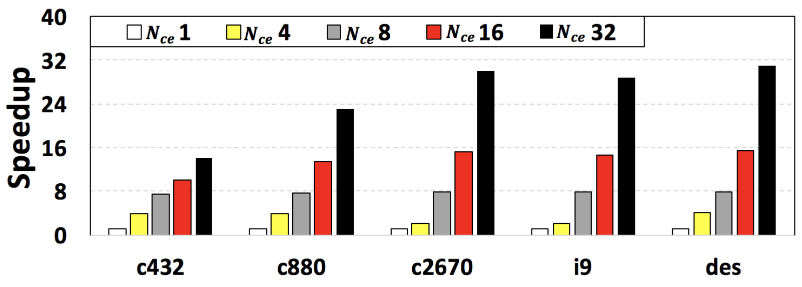


Figure 8.8. Scalability of GenUnlock to the number of CNF CEs. The speedup is near-linear with N_{CE} on large circuits where CNF checking is the computation bottleneck.

Sensitivity to Obfuscation Overhead

Encryption overhead is defined as the ratio of the additional key gates to the total number of gates in the original circuit. Larger encryption overhead suggests that a longer key sequence is used to encrypt the circuit. Figure 8.9 shows the execution time averaged across all assessed benchmarks with varying obfuscation overhead. One can see that GenUnlock's execution time does not grow exponentially with the increase of the obfuscation overhead, suggesting the scalability of GenUnlock framework to large circuits.

8.7 Summary

In this chapter, we introduce GenUnlock, the first genetic algorithm-based framework for logic unlocking. GenUnlock leverages an *Algorithm/Software/Hardware co-design* approach and

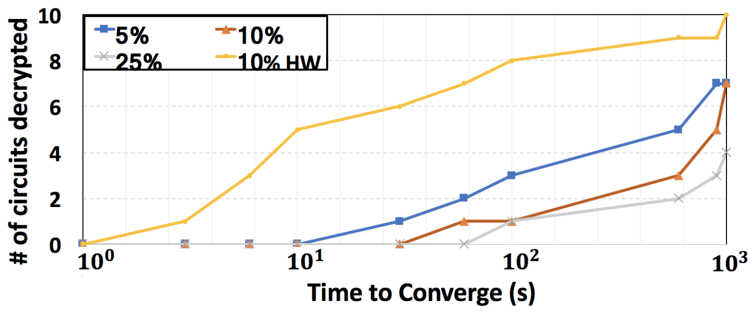


Figure 8.9. Execution time of GenUnlock averaged across all benchmarks. Circuits are encrypted using the logic locking technique in [RPSK12] with different obfuscation overhead.

engenders superior performance improvement compared to traditional SAT-based attacks. More specifically, GenUnlock is *encryption-agnostic* and is generic to arbitrary circuit designs. Our framework yields a set of feasible keys that unlock the obfuscated circuit with an attacker-defined output fidelity. Furthermore, GenUnlock, for the first time, provides the trade-off between runtime overhead and output fidelity of the resulting keys.

In real-world settings, GenUnlock poses a threat to the rising amount of fault-tolerant applications such as block-chain mining and deep neural networks. We devise various hardware optimization techniques including circuit emulation and pipelining to further accelerate the computation of GenUnlock. We perform comprehensive experiments to corroborate the efficiency and effectiveness of GenUnlock across different circuit benchmarks. In future work, we will consider using the learning-based algorithms to guide the mutation and crossover operations.

8.8 Acknowledgements

This chapter, in part, has been published in 2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD) and appeared as Huili Chen, Cheng Fu, Jishen Zhao, Farinaz Koushanfar, “GenUnlock: An Automated Genetic Algorithm Framework for Unlocking Logic Encryption” [CFZK19b]. The dissertation author was the primary author of this material.

Chapter 9

AdaTest: Reinforcement Learning for Hardware Trojan Detection

This chapter presents AdaTest, a novel *adaptive test pattern generation* framework for efficient and reliable Hardware Trojan (HT) detection. HT is a backdoor attack that tampers with the design of victim integrated circuits (ICs), resulting in private information leakage or circuit malfunction. AdaTest improves the existing HT detection techniques in terms of scalability and accuracy in detecting smaller Trojans in the presence of noise and variations. To achieve high trigger coverage, AdaTest leverages Reinforcement Learning (RL) to produce a diverse set of test inputs. Particularly, we *progressively* generate test vectors with high 'reward' values in an *iterative* manner. In each iteration, the test set is evaluated and adaptively expanded as needed. Furthermore, AdaTest integrates *adaptive sampling* to prioritize test samples that provide more information for HT detection, thus reducing the number of samples while improving the samples' quality for faster exploration.

We develop AdaTest with a *Software/Hardware co-design* principle and provide an optimized on-chip architecture solution. AdaTest's architecture minimizes the hardware overhead in two ways: (i) Deploying circuit emulation on programmable hardware to accelerate reward evaluation of the test input; (ii) Pipelining each computation stage in AdaTest by automatically constructing auxiliary circuit for test input generation, reward evaluation, and adaptive sampling. Our architecture design is modular and is *scalable* to large circuits. We evaluate AdaTest's performance on various HT benchmarks and compare it with two prior works that use logic testing for HT detection. Experimental results show that AdaTest engenders up to two orders of test generation speedup and two orders of test set size reduction compared to the prior works while achieving the same level or higher Trojan detection rate.

9.1 Introduction

Integrated circuits (ICs) are indispensable components for a diverse set of real-world applications including healthcare systems, smart home devices, industrial equipment, and Machine Learning (ML) accelerators [CES16, CLC$^+$09]. The vulnerability of digital circuits may result in severe outcomes due to their deployment in security-critical tasks. The design and manufacturing process of contemporary ICs are typically outsourced to (untrusted) third parties. Such a supply chain structure results in hardware security concerns, such as sensitive information leakage, performance degradation, and copyright infringement [TW11, CB14]. Malicious hardware modifications, a.k.a., *Hardware Trojan (HT)* attack [TK10, BHBN14] may occur at each stage of the IC supply chain.

There are two main components in a HT attack: Trojan trigger and payload. The HT *trigger* is a control signal that determines when the malicious activity of the HT shall be activated. The Trojan *payload* is the actual effect of circuit malfunctioning which depends on the purpose of the adversary, e.g., stealing private information or producing incorrect outputs [TK10]. The attacker intends to design a stealthy HT that remains dormant during functional testing and evades possible detection techniques. As such, the HT trigger is typically derived from the rather rare activation conditions that are easier to hide for the intruder.

To alleviate the concerns about malicious hardware modifications, a line of research has focused on developing effective HT detection methods. Existing HT detection techniques can be categorized into two classes based on the underlying mechanisms: *(i) Side-Channel Analysis* (SCA), and, *(ii) Logic Testing*. SCA-based HT detection explores the fact that the presence of the HT on the victim circuit will change its *physical parameters* (e.g., time, power, and electromagnetic radiation), thus can be revealed by side-channel information [LHM14, LKG$^+$09]. Such a mechanism determines that SCA-based approaches can detect *non-functional* HTs, while they may have high false alarm rates when detecting small HTs due to the operational and physical silicon variation, as well as measurement noise. Logic testing-based techniques intend to activate the stealthy Trojan trigger by generating diverse test patterns [CWP$^+$09, NFH18, SCN$^+$15]. The main challenge of logic testing-based HT detection is to increase the *trigger coverage* with a small number of test patterns.

To enable practical trustworthy IC, efforts are also made on developing hardware to detect potential HTs on-chip. For instance, [ZT11] proposes to use a Ring Oscillator Network (RON) to monitor the power signature of the circuit and verify if the silicon chip is Trojan-free. The suggested RON structure suppresses measurement noise and also compensates for process variations. Statistic analysis is then used to detect the Trojan's contribution to the circuit's transient power. A system-on-chip (SoC) level HT detection method using Reconfigurable Assertion Checkers (RACs) is presented in [AG19]. This paper uses Property Specification Language (PSL) for assertion-based verification. The paper [MRIP17] demonstrates how to detect HTs on Pipelined Multiprocessor SoCs via continuous monitoring and testing of the hardware behaviors.

In this chapter, we aim to simultaneously address three challenges of logic testing-based HT detection: effectiveness, efficiency, and scalability. To this end, we propose AdaTest, the first automated **adaptive, reinforcement learning-based test pattern generation (TPG)** framework for HT detection with *hardware accelerator design*. Figure 9.1 demonstrates the high-level usage of AdaTest to inspect if any hardware Trojans are inserted in the CUT. AdaTest takes the netlist of the circuit under test (CUT) and user-defined parameters as its inputs. A set of test vectors with high reward values are returned as the output of AdaTest.

AdaTest framework consists of two main phases: **(i) Circuit profiling**. Given the circuit netlist, we first characterize each node in the CUT from two perspectives: the *transition*

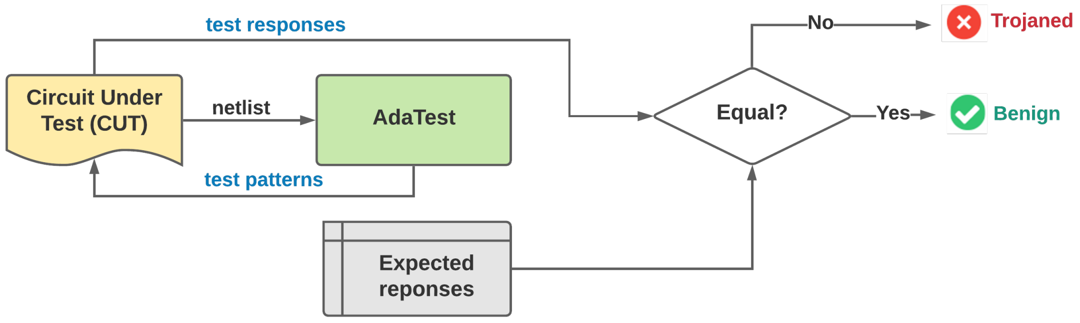


Figure 9.1. High-level usage of AdaTest for hardware-assisted security assurance against Trojan attacks.

probability, and the *SCOAP testability* measures. These two properties are used to identify rare nodes and quantify the fitness of each node, respectively. **(ii) Adaptive test pattern generation.** AdaTest proposes an innovative reward function for test vectors using the following information: the number of times that each rare node is triggered, the SCOAP testability measure of the rare nodes, and the graph-level distance of the circuit (represented as directed acyclic graph) when applying this test input and the historical ones. In each iteration, AdaTest gradually expands the test set by generating candidate test inputs and selecting the ones that have high reward values. AdaTest provisions a flexible *trade-off* between trigger coverage and test generation time. To enable a hardware-assisted solution, we further design an optimized architecture for AdaTest's implementation to reduce the hardware overhead. More specifically, AdaTest architecture pipelines the computation in online TPG and deploys circuit emulation to accelerate the reward evaluation process.

Our technical contributions are summarized as follows:

- **Presenting the first adaptive, reinforcement learning-based test pattern generation framework for HT detection.** AdaTest's dynamic, progressive test generation approach facilitates efficient exploration of the circuit input space, thus is more scalable compared to the existing logic testing-based detection methods.

- **Enabling HT detection with performance trade-off.** AdaTest provisions the defender with the trade-off between trigger coverage and test generation time, thus enabling unprecedented detection flexibility.

- **Leveraging an Algorithm/Software/Hardware co-design approach to devise an efficient detection scheme.** To further accelerate HT detection, AdaTest incorporates various optimization techniques, including computation pipelining and circuit emulation on programmable hardware.

- **Investigating the performance of AdaTest on various circuits.** We perform an extensive assessment of AdaTest and compare the results with the state-of-the-art logic testing-based detection techniques. Empirical results corroborate the superior effectiveness, efficiency, and scalability of AdaTest.

AdaTest opens a new axis for the growing research in hardware security by exploring the idea of reinforcement learning (RL) and adaptive test pattern generation. The adaptive nature of AdaTest ensures that the quality (measured by our reward function) of our dynamic test set always improves over iterations as new test inputs are added to the test set. Furthermore, AdaTest is *generic* and can be easily extended for other hardware security problems, such as logic verification, efficient ATPG, functional testing, and built-in self-test. For example, the concept of RL and adaptive test pattern generation presented in AdaTest can be used in an efficient ATPG application where the RL reward function is designed to reflect the goal of the ATPG (such as fault coverage of considered fault models).

Organization. Section 9.2 introduces preliminary knowledge and related works on Hardware Trojan and its detection, as well as reinforcement learning. Section 9.3 discusses the challenges of HT detection and the overall workflow of AdaTest framework. Section 9.4 presents our test pattern generation algorithm that combines RL and adaptive sampling for fast exploitation. Section 9.5 demonstrates our domain-specific architecture design of AdaTest. Section 9.6 provides a comprehensive performance evaluation of AdaTest on various circuits and comparison with prior works on logic testing-based HT detection. Section 9.7 concludes the paper.

9.2 Related Works

Previous HT detection techniques can be categorized into two broad types: destructive and non-destructive methods. Destructive detection schemes perform de-packaging and de-layering on the manufactured IC to reverse engineer its design layout, thus is prohibitively expensive [EMGT15]. HT detection has been an important research direction and attracted tremendous attention. Non-destructive HT detection includes two types: run-time monitoring and test-time detection. Run-time approaches monitor the IC throughout its entire operational life-cycle with the goal of detecting Trojans that pass other detection methods, providing the 'last-line of defense'. There are two classes of test-time HT detection techniques. We detail each type as follows:

(i) Side-channel Analysis. SCA-based Trojan detection methods explore the influence of the inserted HT on a particular measurable physical property, such as the supply current, power consumption, or path delay. These physical traces can be considered as the *'fingerprint'* of the circuit and allow the defender to detect both *parametric* and *functional* Trojans [LVHM15, LHM14]. Parametric Trojans modify the wires and/or logic in the original circuit while functional Trojans add/delete transistors or gates in the original chip [WSTP08, KRR12, MKG$^+$15]. However, SCA-based HT detection has two limitations: (i) It cannot detect a small HT that causes a negligible impact on the physical side-channel; (ii) The extracted circuit fingerprint is susceptible to manufacturing variation and measurement noise, thus it might incur high false alarm rates.

(ii) Logic Testing. Compared to the side-channel-based approaches, logic testing methods can only detect *functional* Trojans. However, they yield reliable results under process variation and measurement noise. The main challenge of developing a practical and effective logic testing technique for HT detection is the inordinately large space of possible Trojan designs that the adversary can explore. Since the HT trigger is derived from a very rare condition that is unknown to the defender, attempting to stimulate the stealthy Trojan with a limited number of test inputs is difficult. Existing logic testing methods generate test patterns using simple heuristics, and thus cannot ensure high trigger coverage on complex circuits. Also, such heuristic-driven test generation approaches are inefficient (long test generation time) and unscalable to large benchmarks [CWP+09, BHBN14, TK10].

Besides SCA and logic testing, other HT detection techniques have also been explored. For instance, FANCI [WSS13] presents a Boolean functional analysis method to identify suspicious wires that are nearly unused in the circuit. For this purpose, FANCI introduces a concept called 'control value' to characterize the influence of a specific wire on other wires. The wires with small control values are flagged as suspicious. However, the wire-wise control value computation in FANCI is unscalable on large circuits. VeriTrust [ZYW+15] suggests a verification method to detect HT trigger inputs by examining the verification corners. Therefore, VeriTrust is agnostic to the HT implementation styles.

Prior works on logic testing have explored various heuristics to improve trigger coverage while reducing the test generation time. Conceptually similar to the *'N-detection test'* in stuck-at Automatic Test Pattern Generation (ATPG), MERO [CWP+09] leverages random test vectors and mutates them until each rare node in the circuit is individually triggered at least *N* times. Such a simple detection heuristic results in an unsatisfying trigger coverage, particularly Trojans that are hard-to-activate.

To overcome the limitation of MERO, the paper [SCN+15] proposes to use Genetic Algorithms (GA) and Boolean Satisfiability (SAT) to produce test inputs that excite regular rare nodes and internal *hard-to-trigger* nodes, respectively. As the end result, [SCN+15] achieves a higher trigger coverage compared to MERO, while it is inefficient due to the long test generation time. TRIAGE [NFH18] further improves GA-based test generation by devising a more appropriate 'fitness' function that incorporates the controllability and observability factors of rare nodes. However, the GA nature of TRIAGE limits its efficiency for test input space exploration and the resulting test set might be unnecessarily large. TGRL [PM21] suggests training a machine learning model for test patterns generation that combines rare signal stimulation as well as controllability/observability analysis. Although TGRL claims to explore reinforcement learning, its test pattern generation pipeline does not involve sequential decision-making in standard RL techniques. Instead, TGRL learns an ML model via stochastic gradient descent for ATPG.

We aim to develop an adaptive test pattern generation framework for *logic testing* with a high Trojan coverage and a small test set size. Therefore, AdaTest belongs to the test-time detection category introduced in Section 9.2. We choose reinforcement learning over other ML techniques (e.g. neural networks) since the reward-oriented and progressive nature of RL makes it appealing to our goal. Furthermore, to reduce the complexity of RL, AdaTest integrates adaptive sampling to prioritize test patterns that provide more useful information for HT detection.

9.2.1 Hardware-assisted Security Solutions

Previous works have developed various techniques to ensure system-level security, such as hardware architecture for on-chip Trojan detection and built-in self-test (BIST). The paper [ZZT+15] proposes a hardware design of low overhead for HT detection by inserting 2-to-1 MUXs as test points. The number of MUXs is minimized leveraging the fact that the logic gates have a large impact on the transition probability of the gates in their fan-out cone. BISA [XT13] is a built-in self-authentication technique that fills the unused space in a circuit with functional cells to extract the digital signature with logic testing. As such, Trojan insertion can be prevented since it will yield a different signature.

A hardware implementation for a deterministic test generator is presented in LFSROM [DCV93]. The paper suggests a design that uses a cyclic shift register, OR gate network, and a MUX controlled by ripple counters. As the result, LFSROM achieves reduced hardware cost for deterministic test pattern generation compared to the conventional ROM-based design. [MP14] provides an FPGA prototype of low-power test pattern generation-based BIST architecture. The low-power TPG is achieved using Linear Feedback Shift register (LFSR), m-bit generator, gray code generator, NOR-gate network, and XOR array. The proposed BIST hardware reduces the dynamic power consumption of testing by reducing the switching activity among the test patterns without affecting the fault coverage.

9.3 AdaTest Overview

In this section, we first discuss the limitations of prior works on Hardware Trojan detection and our motivation (Section 9.3.1), then introduce our assumptions and threat model for AdaTest framework (Section 9.3.2). We demonstrate the overall workflow of AdaTest test pattern generation technique in Section 9.3.3. AdaTest is a hardware-friendly framework and we present our architecture design in Section 9.5.

9.3.1 Motivation and Challenges

Prior works have advanced logic testing-based Trojan detection using various methods [CWP$^+$09, SCN$^+$15, NFH18]. We discuss the limitations of these detection schemes below.

MERO. Inspired by the traditional ‘N-detect’ test used in stuck-at ATPG, MERO [CWP$^+$09] generates random test vectors to activate each rare node (identified as nodes with transition probability smaller than the threshold θ) to the corresponding rare value at least N times. MERO has three main disadvantages: (i) Triggering all rare nodes for N times might be very time-consuming or even impractical; (ii) It yields low trigger coverage for hard-to-trigger Trojans; (iii) It only explores a small number of test vectors in the entire possible space due to its bit mutation and test vector selection policy.

ATPG based on GA+SAT. The paper [SCN$^+$15] combines genetic algorithms and SAT in test pattern generation for HT detection. While it improves the trigger coverage compared to MERO, [SCN$^+$15] has two constraints: slow test set generation and large memory footprint.

TRIAGE. This paper [NFH18] integrates the benefits of MERO [CWP$^+$09] and [SCN$^+$15]. TRIAGE leverages the SCOAP testability parameters and advises the fitness function of GA for HT detection. However, the evolutionary nature of GA determines that TRIAGE might be ‘trapped’ in the vicinity of a local optimum, thus exploring only a small portion of the whole test inputs searching space.

We present AdaTest as a holistic solution to address the limitations of the previous works.

To this end, we identify three main challenges of developing an efficient and effective logic testing-based HT detection technique as follows:

(C1) High trigger coverage. The test vector set shall yield a high trigger coverage rate to ensure that the probability of activating the stealthy Trojan is large. This property is critical for the *effectiveness* criterion of HT detection.

(C2) Efficient test generation. The runtime overhead of test pattern generation shall be reasonable while attaining a high trigger coverage. For hardware-assisted security, this implies that a test set with a smaller size is preferred. This requirement assures the efficiency and practicality of the HT detection method, particularly on large circuits.

(C3) Scalable to large benchmarks. The runtime consumed by the test pattern generation technique shall not scale exponentially with the size of the examined circuit.

AdaTest tackles the above challenges $(C1) \sim (C3)$ using an *adaptive, RL-based* input space exploration approach. Furthermore, we provide architecture design for AdaTest-based TPG in Section 9.5 to enable hardware-assisted security. We empirically corroborate the superior performance of AdaTest compared to the above counterparts in Section 9.6.

9.3.2 Threat Model

As shown in Figure 2.16, A hardware Trojan consists of two parts: trigger and payload. Figure 2.16 shows an example of HT design. AdaTest is applicable to both combinational and sequential circuits. One can unroll sequential circuits into combinational ones and apply AdaTest for test pattern generation. Without the loss of generality, we assume that the adversary uses a logic-AND gate as the Trojan trigger that takes a subset of rare nodes as its inputs. An XOR gate is used to flip the value of the payload node when the trigger is activated (i.e., each of the trigger nodes has a logical value '1').

We make the following assumptions about AdaTest framework:

(i) The defender knows the netlist of the circuit under test. We assume the party that executes logic testing has the netlist description of the circuit to be examined. This netlist can be obtained by performing de-packaging, de-layering, and imaging [TJ09, LWS12, MZJ16, FSK$^+$17] on the physical circuit. While hardware obfuscation techniques such as camouflaging [YMSR16, LSM$^+$17, SSTF19, SPJ19] and logic encryption [YRSK15, YS17, XS18, TKL$^+$20] could make the trigger design of the Trojan harder to identify, we consider the scenario where the circuit under test is not encrypted in our threat model since this setting is also used in previous Trojan detection papers [CWP$^+$09, SHS$^+$17, YYC$^+$20, PM21].

(ii) The defender can observe the 'indication signal' when the Trojan is activated. We assume the defender can observe certain *manifestations* of the hidden Trojan when it is activated. In particular, we assume the defender knows the correct response of the CUT to a given test input and observes the primary outputs of the CUT for comparison. Note that AdaTest is compatible with techniques that increase manifestation signals (e.g., test point insertion).

9.3.3 Global Flow

Figure 9.2 illustrates the global flow of AdaTest. AdaTest framework consists of two stages: (i) Circuit profiling phase (offline) that computes the transition probabilities and SCOAP testability parameters of the netlist; (ii) Adaptive RL-based test set generation phase (online) that progressively identifies test vectors with high reward values.

Phase I: Circuit Profiling. This stage includes the following:

(1) Compute Transition Probabilities. Given the netlist of the circuit under test, AdaTest first computes the *transition probability* of each internal node in the netlist. In particular, we use the method in [STP11] and assume that each primary input has an equal probability of taking a logical value of 0 and 1. We make this assumption about the primary input values since previous Trojan detection papers [STP11, BHBN14, XFJ$^+$16, LLZ16] use the same assumption when computing the transition probability. Mathematically, the transition probability of a node is computed as $P_{trans} = p(1-p)$ where $p = Prob(node = 1)$. P_{trans} of each node is then compared with a pre-defined threshold θ to identify the *rare nodes*. Identifying rare nodes is important for HT detection since the defender does not know the exact set of trigger nodes used by the

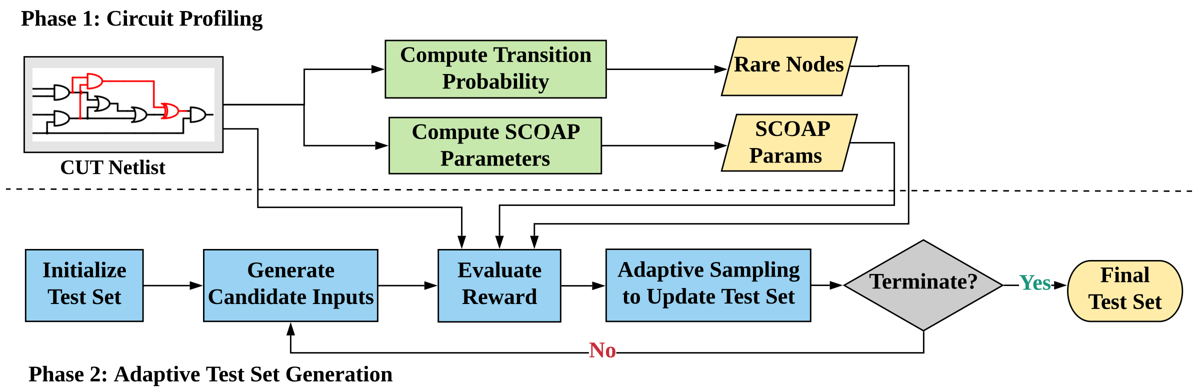

Phase 1: Circuit Profiling
CUT Netlist
Compute Transition Probability
Compute SCOAP Parameters
Rare Nodes
SCOAP Params
Initialize Test Set
Generate Candidate Inputs
Evaluate Reward
Adaptive Sampling to Update Test Set
Terminate?
Yes
No
Final Test Set
Phase 2: Adaptive Test Set Generation

Figure 9.2. Global flow of AdaTest framework for Hardware Trojan detection.

attacker. As such, the activation status of rare nodes provides guidance to generate test inputs that are likely to trigger the stealthy Trojan.

(2) Compute SCOAP Testability Parameters. Controllability and observability are important testability characteristics of a digital circuit. More specifically, *'controllability'* describes the ability to establish a specific node to 0 or 1 by setting the primary inputs. *'Observability'* defines the capability of determining the value of a node by controlling the circuit's inputs and observing the outputs. The *testability* parameters are useful for Trojan detection since they allow AdaTest to distinguish the quality of different rare nodes.

Phase II: Adaptive RL-based test pattern generation. After the CUT is profiled offline in Phase 1, AdaTest performs adaptive test input generation as shown in the bottom of Figure 9.2. We outline each step as follows:

(1) Initialize Test Set. AdaTest first generates an initial test vector set that is used in the later steps. A naive way to do so is random initialization, which may not be optimal for HT detection. To improve the trigger coverage in the later runs, AdaTest employs SAT to find a number of test inputs that activate a subset of rare nodes. We call this method *'smart initialization'* and empirically corroborate its effectiveness in Section 9.6.1.

(2) Generate Candidate Test Inputs. In each iteration of AdaTest's adaptive test vector generation, we first produce a sufficient number of candidate test input patterns that might improve the detection performance when added to the current test set. AdaTest deploys random test generation for this purpose.

(3) Evaluate Reward Function. AdaTest applies the candidate test inputs on the examined circuit and collects the observations, i.e., the netlist status represented as a directed acyclic graph (DAG). We incorporate transition probabilities and SCOAP testability parameters from Phase 1 as well as a novel DAG-level diversity measure to define our reward function.

(4) Adaptive Sampling to Update Test Set. Inspired by the selection step in genetic algorithms, we design an adaptive sampling module that picks 'high-quality' test patterns for fast and efficient input space exploration. In particular, after computing the reward value of each test input in the candidate test vectors, AdaTest selects the ones with the highest scores and append them to the current test set.

At the end of each iteration, AdaTest checks the termination condition and decides whether or not the progressive test generation process shall continue.

Performance Metrics. We use *effectiveness* and *efficiency* as two main metrics to assess the performance of a Trojan detection scheme. In particular, we measure effectiveness from two aspects: trigger coverage and Trojan coverage (i.e. detection rate). The efficiency property is measured by the test set generation time and test set size. AdaTest, for the first time, provides the trade-off between effectiveness and efficiency by adaptively generating a set of test patterns with evolving quality over time. We provide quantitative analysis of the above metrics in Section 9.6.

9.4 AdaTest Algorithm Design

The key to ensuring a high probability of Trojan detection using logic testing is to generate a test set that can trigger the circuit to diverse states, in particular, the rare nodes in the circuit. To this end, AdaTest leverages three important characteristics of the circuit: the transition probabilities, the SCOAP testability measures, and the DAG-level diversity. In particular, AdaTest employs an *RL-driven* test pattern generation approach that uses the above three properties to progressively generate test inputs. Inspired by the selection stage in genetic algorithms, we integrate an adaptive sampling module that progressively expands the current test set (used as historical information) with high-quality test patterns. This **response-adaptive** design is beneficial for statistical search of the HT trigger in the circuit input space, thus improves the efficiency of AdaTest's RL-based pipeline. We detail the two main phases of AdaTest shown in Figure 9.2 in the following of this section.

9.4.1 Circuit Profiling

Algorithm 8 outlines the steps of the circuit profiling phase in AdaTest. This stage obtains two informative properties of the circuit: the transition probabilities and testability measures. In particular, we use *random testing* and *logic simulation* to estimate the transition probability P_{trans} of each node in the netlist C_n. To further investigate the rewards of different rare nodes, AdaTest also computes the SCOAP parameters of the nodes using the technique in [GT80]. This step can be considered as an offline, pre-processing step before executing the genetic algorithm-based circuit deobfuscation scheme.

Algorithm 8. Circuit Profiling.

```
INPUT: Netlist of the circuit under test ($C_n$); Number of random tests ($H$); Threshold on
    transition probability ($\theta$) for rare nodes.
OUTPUT: The set of rare nodes ($R$); Computed testability parameters $TP$ =
    ($CC0, CC1, CO$).
 1: Initialize rare node set: $R \leftarrow \emptyset$
 2: Generate random inputs: $I \leftarrow RandGen(C_n,\ H)$.
 3: Perform logic simulation: $O \leftarrow LogicSim(C_n,\ I)$.
 4: for node in $C_n$ do
 5:     Compute frequency: $p = CountOnes(O, node)/H$
 6:     Estimate transition probability: $P_{trans} = p(1-p)$
 7:     if $P_{trans} < \theta$ then
 8:         $R \leftarrow R \cup node$
 9: Obtain SCOAP parameters:
        $(CC0,\ CC1,\ CO) \leftarrow ComputeSCOAP(C_n)$
10: Return: Obtained rare node set $R$, SCOAP testability parameters $TP = (CC0, CC1, CO)$.
```

AdaTest's circuit profiling stage characterizes the *static reward* properties of the circuit in terms of the transition probabilities of rare nodes and testability measures. We call these two properties *'static'* since they are *independent* of the circuit input for a given circuit netlist. As such, our profiling phase can be performed *offline*. The above two properties are indispensable for the reward computation step in Phase 2 of AdaTest since: (i) Transition probabilities and rare nodes shed light on the potential trigger nodes exploited by the malicious adversary. The defender knows that a subset of rare nodes are used to design the stealthy Trojan while he has no knowledge about the exact trigger set. As such, rewarding the activation of rare nodes encourages the test vectors to stimulate the possible HT. Note that the Trojan activation condition is equivalent to knowledge of the exact trigger set and both are assumed to be unknown to the defender. (ii) Testability parameters provide more fine-grained information about the quality of individual rare nodes in the context of HT detection. One can compare the fitness of two test inputs by counting and comparing the number of activated rare nodes corresponding to each test vector. However, such a naive counting mechanism neglects the intrinsic difference between the quality of individual rare nodes. In principle, a rare node with higher controllability and observability shall be assigned with higher reward values. As such, AdaTest integrates the SCOAP testability measures to quantify the reward of each activated rare node.

9.4.2 Adaptive RL-based Test Pattern Generation

AdaTest deploys a *progressive, reinforcement learning-driven* algorithm for efficient and effective test input space exploration with the goal of HT detection. Section 2.3.3 introduces the basics of RL. We discuss how we map the Trojan detection problem to the RL paradigm below.

AdaTest's RL Formulation of Trojan Detection:

▪ **State.** The objective of AdaTest is to adaptively generate test patterns with high effectiveness for Trojan detection in an iterative manner. As such, AdaTest defines a *state* as the *current test set* in the present iteration.

▪ **Action Space.** Recall that an action transforms the agent into a new state, which is the new test set according to our definition of the state above. Therefore, a feasible *action* for AdaTest is to *identify a set of new test input vectors* in each iteration that improves the quality of HT detection when added to the current test set.

▪ **Environment.** For HT detection, the *netlist of the circuit* (C_n) can be considered as the *environment* that converts the current state and the action, and returns the reward value.

▪ **Observations.** The agent makes the observation of the environment before reward computation. For Trojan detection problems, we model the *DAG formed by the values of all nodes* in the netlist given a specific input vector as an *observation* of the circuit state.

▪ **Reward.** The definition of the reward function directly reflects the objective of the problem that one aims to solve. As such, for the task of logic testing-based HT detection, AdaTest designs a *composite reward* function to encourage the generation/exploration of test inputs that facilitate the excitation of the potential HT.

The mathematical definition of AdaTest's *dynamic* reward function is given below:

$$Reward(T_i|\, S_i) = \lambda_1 \cdot V_{rare}(T_i,\, R) + \lambda_2 \cdot V_{scoap}(T_i,\, R,\, TP) + \lambda_3 \cdot V_{DAG}(T_i|\, S_i). \tag{9.1}$$

Here, S_i and T_i are the current test set (i.e., the state) and the newly generated test inputs in i^{th} iteration, respectively. R and TP are the set of rare nodes and the SCOAP testability parameters identified in Phase 1 (*static attributes*). The hyper-parameters λ_1, λ_2, λ_3 determine the relative weighting of the three reward terms. The reward function $Reward(T_i|\, S_i)$ characterizes the fitness of the specific test inputs T_i while considering the current test set S_i. Evaluating the reward value of T_i *in the context of the historical test patterns* (S_i) makes AdaTest's RL framework *adaptive* and intelligent.

We detail how each term in AdaTest's reward function is designed below. Inspired by the 'N-detect' test, the first reward term in Equation (9.1) aims to activate each rare node in the circuit for at least N times. To this end, we define the **rare node reward** R_{rare} as follows:

$$V_{rare}(T_i,\ R) = -\sum_{r \in R} abs(N - Ctr_i(r)), \tag{9.2}$$

where $Ctr_i(r)$ is the number of times that the rare node r is activated to its corresponding rare value up to the i^{th} iteration.

The second reward term in Equation (9.1) leverages the SCOAP parameter $TP = (CC0, CC1, CO)$ from Phase 1 to encourage the stimulation of rare nodes with high controllability and observability. Given the current test set S_i, we can obtain the set of activated rare nodes Rtr_i (which is a subset of R). The **SCOAP testability reward** V_{scoap} is then computed:

$$V_{scoap}(T_i,\ R,\ TP) = \sum_{r \in Rtr_i} CC(r) + CO(r). \tag{9.3}$$

Here, $CC(r)$ and $CO(r)$ denote the controllability and observability of the rare node r when set to its rare value. More specifically, $CC(r)$ shall be converted to $CC0(r)$ or $CC1(r)$ depending on the rare value of the node r.

Besides leveraging the static attributes identified in Phase 1 to define the rare node reward R_{rare} and the SCOAP testability reward R_{scoap}, AdaTest further explores the **graph-level diversity** extracted from the circuit netlist. In particular, AdaTest identifies the dynamic fitness property, i.e., the DAG-level diversity that is jointly determined by the circuit netlist and the test vector set. Such a DAG-level distance serves as a *dynamic* fitness measure since it is *input-aware*. Recall that AdaTest leverages an RL paradigm and considers the value assignments of all nodes when given the netlist C_n and a specific test input as the observation. We use the *graph representation* of the circuit to abstract the observed netlist status. To facilitate the computation, AdaTest flattens the DAG to an *ordered sequence* based on the circuit level information. The distance between the two transformed DAG sequences is used as the DAG-level diversity measure.

To summarize, we define the **DAG diversity reward** as follows:

$$V_{DAG}(Ti|\ S_i;C_n) = HammDist(DAG(T_i;\ C_n),\ DAG(S_i;\ C_n)). \tag{9.4}$$

Here, $DAG(T_i;C_n)$ denotes the flattened ordered sequence of the DAG obtained when applying the test inputs T_i to the circuit C_n. The diversity measurement function $HammDist$ computes the normalized pairwise distance of the flattened DAGs using the Hamming distance metric. Since the DAG sequence of the circuit is binary-valued (0 or 1), AdaTest employs XOR function as an efficient implementation of the $HammDist$ function. It's worth noting that this graph reward V_{DAG} is aware of historical test inputs (S_i), thus providing guidance to select new inputs that stimulate different internal nodes structure in the context of current test inputs S_i.

■ **Policy.** The policy component of a RL algorithm suggests actions to achieve a high reward given the current state. Recall that AdaTest defines the state and the action space as the current set of test vectors and the expansion with the new test patterns, respectively. Therefore, the policy module of AdaTest selects the most suitable test pattern candidates and add them to the result test set (line 5&6 in Algorithm 9).

Algorithm 9 outlines the procedure of our adaptive test set generation framework. We emphasize that **AdaTest does not require explicit training** on the training set, which is typically required by machine learning models (e.g., gradient descent-based training). The RL nature enables AdaTest to search for distinguishing test inputs with the guidance of the composite reward. This makes our detection method fundamentally different from TGRL [PM21] that still trains an ML model for test pattern generation.

We discuss how AdaTest leverages the RL paradigm formulated above to achieve logic testing-based HT detection in the following of this section.

❶ **Smart Initialization.** Recall that the intuition of logic testing-based Trojan detection is to encourage the generation of test inputs that activate diverse combinations of rare nodes to their corresponding rare values. Random test vectors might be unlikely to yield a high trigger

Algorithm 9. Adaptive Reinforcement Learning based Test Input Pattern Generation.

```
INPUT: Netlist of circuit under test ($C_n$); Rare node set $R$; SCOAP testability parameters
    $TP = (CC0, CC1, CO)$; Size of candidate test inputs per iteration ($M$); Size of selected
    test inputs per iteration ($L$); Maximal number of iterations ($I_{max}$); Percentage threshold
    of rare nodes ($p$); Target activation times ($N$).
OUTPUT: A set of test patterns $S$ for Trojan detection of the target circuit $C_n$.
 1: Initialization:
        $S_0 = \left\{\vec{S}_0^1, ..., \vec{S}_0^L\right\} \leftarrow SmartInitialize(L)$.
        Iteration counter: $i \leftarrow 0$
 2: while $i < I_{max}$ and HT is not activated do
 3:     $T_i \leftarrow GenerateTestCandidates(M;\ C_n)$
 4:     $Reward(T_i|\ S_i) \leftarrow EvaluateReward(T_i,\ S_i;\ C_n)$
 5:     $T_i^{top} \leftarrow SelectTopCandidates(T_i,\ Reward,\ L)$
 6:     Update test set: $S_{i+1} \leftarrow S_i \cup T_i^{top}$        ▷ Adaptive sampling to expand test set
 7:     $A_i \leftarrow CountRareNodeActivation(S_i;\ C_n)$
 8:     if $p\%$ elements in $A_i \geq N$ & $A_i.min() \geq 1$ then        ▷ Check termination condition
 9:         break
10:     $i \leftarrow i + 1$
11: Return: Obtained a test set ($S_i$) for logic testing-based HT detection of the circuit $C_n$.
```

coverage, especially on large circuits. To explore the above intuition, AdaTest leverages SAT to generate the initial test set (line 1 in Algorithm 9) such that it is able to activate diverse rare nodes specified by the defender. We empirically validate the advantage of our smart initialization as opposed to the random variant in Section 9.6.1. It is worth noticing that while the defender can identify rare nodes in the circuit by thresholding the transition probabilities, it might be infeasible to find an input that stimulates all rare nodes to their rare values. Therefore, AdaTest tries to generate test patterns that stimulate different combinations of rare nodes for Trojan detection.

❷ **Generate Candidate Test Patterns.** AdaTest progressively identifies test inputs that are suitable for HT detection using an iterative approach. To this end, AdaTest first generates a sufficient number of candidate test vectors at the beginning of each iteration (line 3 in Algorithm 9). These candidates are responsible for exploring the test input space and aim to find solutions with high rewards. In our experiments, we adopt an adaptive sampling method to generate candidate test patterns at each iteration. In particular, the sampling weights for the test vectors in the initial set S_0 are uniformly assigned at iteration 0. In other words, at iteration 0, we perform a uniform sampling to generate candidate test patterns. Then the sampling weights of test vectors at iteration $i+1$ will be updated based on the normalized reward values evaluated at iteration i. Test vectors with higher reward values will result in higher sampling weights, which in turn increases the probability of the test vectors being included in the generated set S. The adaptive sampling method allows us to optimize test pattern generation by favoring test patterns with higher reward values thus enhancing convergence in our test pattern generation.

❸ **Evaluate Reward Function.** The definition of reward is task-specific. Since our objective is to generate test patterns that stimulate the circuit (particularly the rare nodes) to different states for Trojan detection, AdaTest designs an innovative composite reward function as shown in Equation (9.1). In each iteration, the reward values of the candidate test inputs are evaluated (line 4 of Algorithm 9). Our compound reward function captures informative features that are beneficial for HT detection from three aspects: the number of times that each rare node is activated (V_{rare}), the SCOAP testability measures that quantify the fitness of different rare nodes (V_{scoap}), and the graph-level diversity between the current test inputs and historical ones (V_{DAG}).

❹ **Adaptive Sampling to Update Test Set.** Recall that in AdaTest's RL paradigm, the current test set S_i represents the 'state' variable. After obtaining the reward values of individual candidate test input in T_i from Step 3, AdaTest updates the state by selecting a subset of T_i that has the highest reward values and adding them to the current test set S_i. This step is conceptually similar to the selection stage in genetic algorithms. With the domain-specific definition of reward, AdaTest adaptively samples high-quality test patterns from the randomly generated candidate test inputs, therefore facilitating fast exploration of the circuit input space for HT detection.

❺ **Check Termination Condition.** AdaTest's adaptive test set generation terminates if any of the following three conditions is satisfied: (i) $p\%$ of all rare nodes are activated for at least N times and all rare nodes are activated at lease once (line 8 in Algorithm 9); (ii) The maximal number of iteration I_{max} is reached (line 2 in Algorithm 9); (iii) The current test set S_i activates the hidden Trojan, i.e., all involved trigger nodes are activated to their corresponding rare values by S_i (line 2 in Algorithm 9). Note that we include termination condition (iii) since our threat model assumes that the defender can observe the manifestation of an activated Trojan.

Discussion. As summarized in Algorithm 9, our reinforcement learning approach does not require model training. Instead, we progressively generate the set of test vectors using adaptive sampling given the particular circuit with the goal of maximizing the RL rewards for Trojan detection. From this perspective, our RL-based detection tool generates a specific test set for the circuit under test. However, AdaTest is generic in the sense that it is agnostic to the circuit structure and can be applied to various types of circuits. In other words, applying AdaTest to a different circuit does not require any model training since we do not incorporate neural networks in our RL detection pipeline shown in Algorithm 9.

9.5 AdaTest Architecture Design

Beyond the novel test generation algorithm discussed in Section 9.4, we design a Domain-specific systems-on-chip (DSSoC) architecture of AdaTest for its practical deployment. The bottleneck of AdaTest implementation is the computation of the test input's reward $Reward(T_i|S_i)$ according to Equation (9.1). Given the rare node-set R and SCOAP testability measures of the circuit TP from offline circuit profiling (Algorithm 8), the online reward evaluation of a new test input T_i involves three terms as shown in Equation (9.1): identifying the rare nodes stimulated by T_i (for V_{rare}), obtaining the SCOAP values corresponding to each active rare node (for V_{scoap}), and computing the DAG-level graph distance (for V_{DAG}). Note that the third component requires us to obtain the DAG with nodes value assignment when applying the test input on the circuit $DAG(T_i;\, C_n)$. This information is also sufficient to compute the first two reward terms. Therefore, the main task for AdaTest's on-chip implementation is to obtain the value-assigned DAG for a new test input on the circuit ($DAG(T_i;\, C_n)$).

To accelerate circuit evaluation, AdaTest deploys *circuit emulation* on the programmable hardware to obtain the response $DAG(T_i;\, C_n)$. Furthermore, AdaTest constructs the customized auxiliary circuitry automatically to pipeline each computation stage and reduce the runtime overhead. We design an optimized DSSoC architecture of AdaTest for efficient implementation of our adaptive TPG method outlined in Algorithm 9.

9.5.1 Architecture Overview

The overall hardware architecture of AdaTest's online test patterns generation is shown in Figure 9.3 (a). AdaTest leverages Algorithm/Software/Hardware co-design approach to accelerate the test inputs searching process shown in Figure 9.2 (Phase 2). More specifically, AdaTest maps the netlist of the circuit under test (C_n) with the auxiliary part to the FPGA and performs circuit evaluation to obtain the circuit's response ($DAG(T_i;\, C_n)$) to the test input T_i. We make this design decision to develop the hardware accelerator for AdaTest since acquiring the circuit's response from a configured FPGA (circuit emulation) is significantly faster than the same process running on a host CPU (software simulation). In addition, AdaTest parallelizes the computation of circuit emulation and pipelines at each step of the RL process. AdaTest performs reward computation of the candidate test inputs and adaptive sampling in an online fashion to minimize data communication between the off-chip memory and the FPGA.

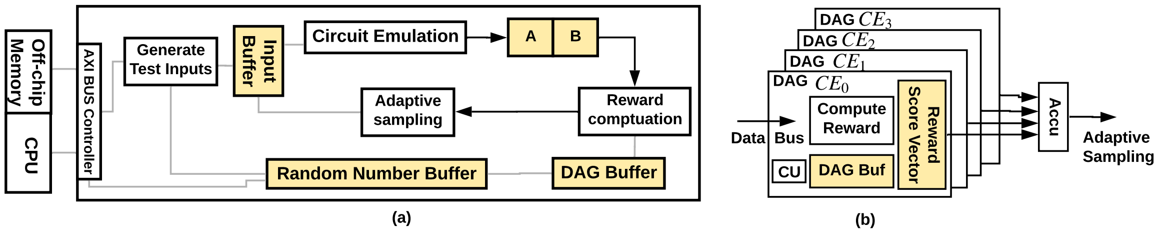


Figure 9.3. Overview of AdaTest architecture design. The overall layout of the hardware system (a) and the implementation of Reward Computation Engines (b) are shown.

Note that we do not include a random number generator (RNG) in our architecture design. Instead, AdaTest stores a set of random numbers pre-computed on CPU using the inherent variation of the operating system. This design choice has two benefits: (i) The hardware overhead of a True RNG is non-trivial and not desired; (ii) Random numbers generated from the

CPU typically feature stronger randomness compared to the one generated on FPGA. The results of circuit emulation are used for computing the reward values of test inputs using Equation (9.1) during reward evaluation. The rare node evaluation and DAG distance computation process in reward evaluation are parallelized by accommodating multiple Computing Engine (CE) in AdaTest's design. We also evenly partition the workload of each CE evenly offline.

After accumulating the reward for each candidate test input, our *adaptive sampling* selects the ones with the highest rewards. This selection process is equivalent to *sorting*. Therefore, AdaTest includes a sorting engine that permutes the key index based on their corresponding rewards. We implement a lightweight sorting engine based on the 'even-odd sort' algorithm [CELT78] for adaptive sampling, incurring a linear runtime overhead with the candidate test set size M.

It is worth noticing that AdaTest does not deploy a central control unit to coordinate the computation flow. Instead, each design component in Figure 9.3 (a) follows a *trigger-based control* mechanism [PPA+13]. Particularly, each module is controlled by the status flag from its previous computation stage. For example, the adaptive sampling module (i.e., the sorting engine) in AdaTest begins to operate when the accumulation of the reward value is detected as completed. Our trigger-based control flow simplifies the control logic while satisfying the data dependency between different components in Figure 9.2. We detail the design of AdaTest's circuit emulation and auxiliary circuitry as follows.

9.5.2 AdaTest Circuit Emulation

We empirically observe from AdaTest's software implementation that circuit evaluation (i.e., obtaining $DAG(T_i;C_n)$) dominates the execution time. Motivated to address the high latency issue of evaluating a circuit netlist on CPU, we propose to use *circuit emulation* to improve AdaTest's efficiency. The first step of circuit emulation is to rewrite the netlist of the circuit under test (C_n) such that the values of internal nodes can be recorded by registers. The rewritten circuit is then connected with the auxiliary circuitry and mapped onto FPGA. In this way, we can emulate the response of the target circuit C_n for any test input by directly applying it to the circuit and collecting the corresponding values in the registers. The collected signal values are used to compute the three reward terms in Equation (9.1).

Furthermore, AdaTest optimizes the latency of hardware evaluation by storing the emulation results in a ping-pong buffer (consisting of two buffers denoted with A and B) and decoupling it from other hardware components as shown in Figure 9.3 (a). More specifically, the reward computing engine (CE) calculates the reward of the candidate test input using the data from buffer A. In the meantime, the emulator acquires the states of C_n given the next input T_i and stores the results into buffer B.

9.5.3 AdaTest Reward Computing Engine

Pipeline with Early Starting. Our architecture design aims to maximize the overlapping time between each execution stage of AdaTest to increase the throughput of TPG. As shown in Figure 9.4, the ping-pong buffer enables pipelined execution of hardware emulation and reward evaluation. Furthermore, reward evaluation and adaptive sampling can be pipelined across different iterations. We can see from Figure 9.4 that epoch $(i+1)$ can start circuit emulation and reward evaluation when the previous epoch begins to generate new test inputs for the next epoch. As such, the latency of candidate test input generation can be hidden by circuit emulation and reward evaluation.

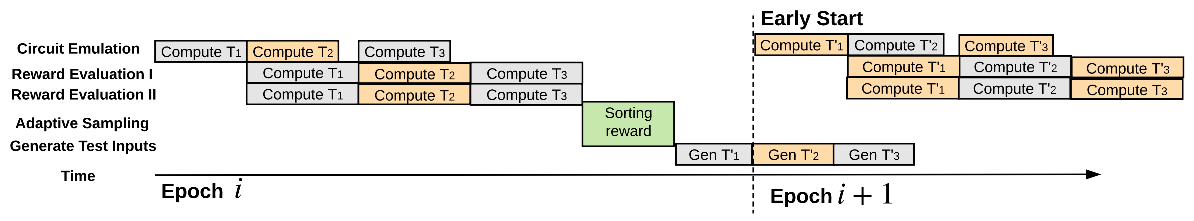


Figure 9.4. AdaTest's hardware accelerator employs pipelining optimization to generate test patterns online for HT detection.

Scalable Reward Computing Engine. Once circuit emulation finishes for the current input T_i, AdaTest begins to calculate the reward of this test input using Equation (9.1). From the hardware perspective, the reward term V_{rare} and V_{scoap} is computed by accumulating the number of activated rare nodes and the corresponding SCOAP values from the circuit C_n, and the reward V_{DAG} is computed by accumulating the Hamming Distance (i.e., XOR) between the values in the current DAG ($DAG(T_i;C_n)$) and the historical ones ($DAG(S_i;C_n)$). Independence between different groups of wire signals typically exists in circuits. AdaTest leverages this property by distributing the computation involving independent groups of nodes to different reward computing engines as shown in Figure 9.3 (b). As such, each CE stores a subset of DAG nodes' values in the associated DAG buffer. The accumulation of the ultimate reward score completes when the last CE finishes reward computing.

9.6 Evaluations

We investigate AdaTest's performance for Hardware Trojan detection on various benchmarks, including ISCAS'85 [HYH99], MCNC [Man12], and ISCAS'89 [BBK06]. The statistics of the evaluated benchmarks are summarized in Table 9.1. To apply AdaTest on sequential circuits in the ISCAS'89 benchmark, we unroll the circuit for two-time frames and convert it to a combinational one [AH04, Yua15]. Note that the unrolling process duplicates the combinational logic blocks, thus increasing the effective circuit size. The transition probability threshold (P_{trans})

Table 9.1. Summary of the evaluated circuit benchmarks.

Circuit	dataset	#in	#out	#gate	# of rare nodes ($P_{trans} < P_T$)
c432	ISCAS-85	36	7	160	14
c499	ISCAS-85	41	32	202	48
c880	ISCAS-85	60	26	383	74
c3540	ISCAS-85	50	22	1669	218
c5315	ISCAS-85	178	123	2307	169
c6288	ISCAS-85	32	32	2416	245
c7552	ISCAS-85	207	108	3512	266
des	MCNC	256	245	6473	2316
ex5	MCNC	8	63	1055	432
i9	MCNC	88	63	1035	85
seq	MCNC	41	35	3519	1356
s5378	ISCAS-89	35	49	2958	258
s9234	ISCAS-89	19	22	5825	398

for rare nodes is set to $P_T = 0.1$ for ISCAS'85 and MCNC benchmarks. As for two ISCAS'89 circuits, we use $P_{trans} = 0.0005$ so that the number of rare nodes is at the same level as the previous two benchmarks. The identification results are shown in the last column of Table 9.1.

To compare AdaTest's performance with other logic testing-based Trojan detection methods, we use trigger coverage and Trojan coverage as the metrics to quantify detection effectiveness. To profile the detection efficiency, we use the number of test vectors and detection runtime as the metrics. We empirically show that AdaTest achieves a higher Trojan detection rate with shorter runtime overhead compared to the counterparts in the rest of this section.

Experimental Setup. Adhering to our threat model defined in Section 9.3.2, we first design the HT and insert it to each benchmark listed in Table 9.1. We use a logic-AND gate as the Trojan trigger and select three rare nodes with rare value 1 as the inputs. To fully characterize the performance of AdaTest, we devise various HTs for each circuit (i.e., using different combinations of rare nodes as the trigger) and repeat the insertion for 50 times. Our Trojaned benchmarks include 'hard-to-trigger' HTs with activation probabilities around 10^{-7} (e.g., $c3540$). To compare the performance of AdaTest with prior works, we re-implement MERO [CWP+09] and TRIAGE [NFH18] based on the methodology described in the paper using Python. Our experiments are performed on an Intel Xeon E5-2650 v4 processor with 14.5 GiB of RAM.

■ **MERO Configuration.** We use the same parameter selection strategy in MERO for re-implementation. Particularly, we set the size of random patterns to 2,500. The hyper-parameter of MERO is N (desired number of times that each rare node shall be activated). A large value of N achieves a higher detection rate while resulting in a larger test set. We use $N = 1,000$ in our experiments since this value is suggested in MERO [CWP+09] .

■ **TRIAGE Configuration.** We use a population size of 100 and select 20 test inputs with the highest fitness score in each generation. The probability of crossover and mutation is set to 0.9 and 0.05, respectively. The termination condition in TRIAGE [NFH18] is used to evolve the test patterns.

■ **AdaTest Configuration.** In AdaTest's circuit profiling stage, we use the Testability

Measurement Tool [Sam14] to compute the SCOAP parameters. The SAT-based smart initialization step of AdaTest's Phase 2 is performed using the pycosat library [Sch17]. Our framework is developed in Python language and does not require extensive hyper-parameter tuning. To ensure the three reward terms in Equation (9.1) have comparable values within the range of $[0,10]$, we set the hyper-parameters to $\lambda_1 = 0.05$, $\lambda_2 = 0.0001$, $\lambda_3 = 0.00025$. The candidate test size and the step size in Algorithm 9 are set to $M = 200$ and $L = 80$ for all benchmarks, respectively. We use the percentage threshold $p = 95\%$ to identify rare nodes and set the target activation times to $N = 20$. The maximal iteration time is set to $I_{max} = 500$.

According to the performance metrics in Section 9.3.3, we use the *trigger coverage* (percentage of trigger nodes identified by the test set) and the *Trojan coverage* (i.e., detection rate) to quantify the effectiveness of HT detection. Meanwhile, we measure the *test set generation time* and test set size of each technique for efficiency comparison. To obtain an accurate and comprehensive performance measurement, we design 50 different HTs for each benchmark in Table 9.1 while fixing the number of trigger nodes to 3. Each set of devised HTs is inserted into the circuit independently. We run AdaTest detection on each Trojaned circuit for 20 times. The trigger and Trojan coverage for each benchmark are computed as the average value over $50 \times 20 = 1000$ runs.

9.6.1 Detection Effectiveness

We assess the detection performance of AdaTest, MERO, and TRIAGE using the aforementioned experimental setup. Figure 9.5 compares the Trojan coverage of the three HT detection techniques on different benchmarks. One can see that our framework achieves uniformly higher detection rates across various circuits. The superior HT detection performance of AdaTest is derived from our definition of *adaptive*, *context-aware* reward functions in Equation (9.1).

We use two metrics to quantitatively compare the effectiveness of different HT detection techniques: **trigger coverage rate** and **Trojan detection rate**. Note that AdaTest determine a Hardware Trojan is present in the circuit if the set of test patterns generated using Algorithm 9

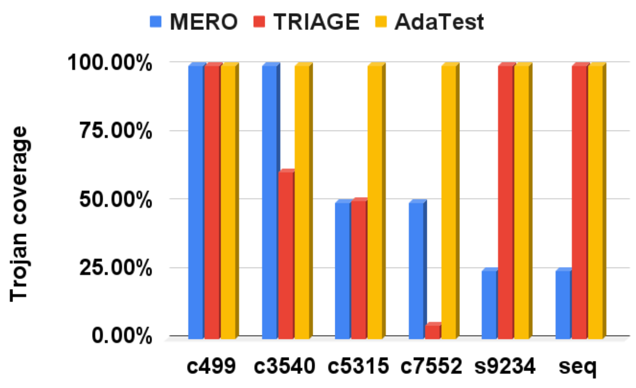


Figure 9.5. Trojan detection rates of AdaTest and prior works on various benchmarks.

result in Trojan activation when the test inputs are applied to the circuit. Therefore, our detection method does not have any false positives and we focus on evaluating the detection rates (which corresponds to the false-negative rate).

Table 9.2 summarizes the HT detection results of three different methods on the benchmarks in Table 9.1. The trigger coverage and Trojan coverage results are shown in the last two columns. It can be seen that AdaTest achieves the highest Trojan coverage while requiring the shortest test generation time across most of the benchmarks. More specifically, AdaTest achieves an average of 15.61% and 29.25%. Trojan coverage improvement over MERO [CWP$^+$09] and TRIAGE [NFH18], respectively. The superior HT detection performance of our logic testing-based approach is derived from the diverse test patterns found by AdaTest adaptive RL-driven input space exploration technique (see Section 9.4.2). We not only encourage the activation of rare nodes and differentiate their qualities using SCOAP testability parameters but also explicitly characterize the graph-level distance of the CUT status under different test stimuli.

We measure the dynamic rare node coverage versus the number of executed iterations to validate the *time-evolving* property of AdaTest framework. Figure 9.6 shows the coverage results of AdaTest with random initialization and SAT-based smart initialization on the $c3540$ benchmark. We can make two observations from Figure 9.6: (i) AdaTest consistently improves the rare node coverage over time (with either initialization method); (ii) SAT-based smart

Table 9.2. Performance comparison summary of different Trojan detection techniques.

circuit	Method	# test vectors	Runtime (s)	Trigger coverage	Trojan coverage
c499	MERO	1660	136.49	100.00%	100.00%
	TRIAGE	250000	25.91	100.00%	100.00%
	AdaTest	**1010**	13.60	100.00%	100.00%
c880	MERO	1332	352.54	100.00%	100.00%
	TRIAGE	250000	1.75	82.29%	18.00%
	AdaTest	**429**	0.43	100.00%	97.50%
c3540	MERO	1920	1577.36	100.00%	100.00%
	TRIAGE	250000	25.85	100.00%	61.00%
	AdaTest	**905**	22.61	100.00%	**100.00%**
c5315	MERO	9265	1660	100.00%	50.00%
	TRIAGE	250000	37.14	100.00%	50.50%
	AdaTest	**1300**	19.76	100.00%	**100.00%**
c6288	MERO	1906	1867.57	100.00%	100.00%
	TRIAGE	250000	44.11	100.00%	91.50%
	AdaTest	**900**	47.06	100.00%	**99.50%**
c7552	MERO	1916	18650.5	100.00%	50.00%
	TRIAGE	250000	20.93	93.88%	5.00%
	AdaTest	**1600**	39.79	98.08%	**100.00%**
s5378	MERO	1103	30960.11	100.00%	100.00%
	TRIAGE	300	0.45	100.00%	100.00%
	AdaTest	100	11.58	100.00%	**100.00%**
s9234	MERO	11	29737.84	100.00%	25.00%
	TRIAGE	500	35.625	100.00%	100.00%
	AdaTest	140	124.99	100.00%	**100.00%**
des	MERO	1120	34943.41	100.00%	100.00%
	TRIAGE	2500	0.84	100.00%	100.00%
	AdaTest	**156.8**	15.11	92.88%	**100.00%**
ex5	MERO	904	115.22	100.00%	100.00%
	TRIAGE	2500	0.13	99.13%	100.00%
	AdaTest	**500**	12.35	93.81%	**100.00%**
i9	MERO	268	808.56	100.00%	100.00%
	TRIAGE	2500	0.09	100.00%	100.00%
	AdaTest	600	12.15	94.58%	**100.00%**
seq	MERO	1776	3773.3	100.00%	66.67%
	TRIAGE	250000	22.11	95.44%	2.00%
	AdaTest	3700	20.72	94.58%	**82.00%**

initialization improves the convergence speed of AdaTest, thus reducing our test set generation time. The first observation corroborates the efficacy of our RL-based *progressive* test pattern generation method. The second observation reveals the importance of proper initialization for fast convergence of RL exploration. Note that a shorter convergence time (i.e., a smaller number of iterations in Algorithm 9) indicates s smaller test set returned by AdaTest, which is beneficial to reduce the test generation time for higher detection efficiency.

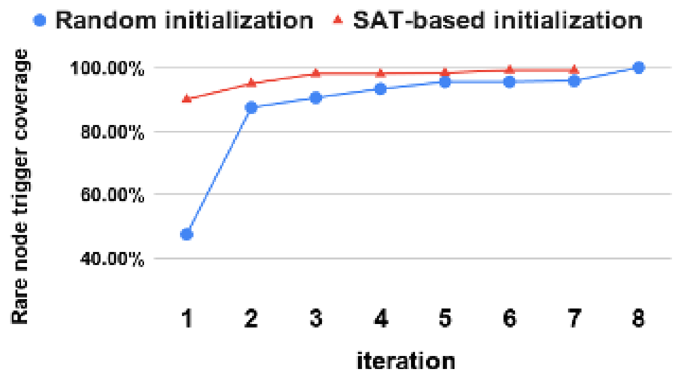


Figure 9.6. The rare node coverage of AdaTest versus the number of executed iterations on c3540 benchmark.

9.6.2 Detection Efficiency

We characterize the efficiency of AdaTest for logic testing based HT detection using two metrics: the test set size (*space efficiency*), and the test set generation time (*runtime efficiency*). The quantitative efficiency measurements of three HT detection methods are shown in the third and fourth columns of Table 9.2. It can be computed that AdaTest engenders an average of 2.04$\times$ and 155.04$\times$ reduction of the test set size compared to MERO and TRIAGE across all benchmarks, respectively. The reduction of test set size has two benefits: (i) A smaller test set features a lower memory footprint; (ii) For on-chip test pattern generation, a smaller test set suggests a shorter test generation time.

Figure 9.7 compares the required test generation time of AdaTest, MERO, and TRIAGE to achieve the coverage results on various benchmarks in Table 9.2. Note that we use *log-scale* for the vertical axis since the range of runtime is diverse across different circuits. We can observe that AdaTest is the most efficient HT detection method among the three and it also achieves high Trojan coverage (last column of Table 9.2). More specifically, AdaTest engenders an average of 366.26$\times$ and 0.63$\times$ test generation speedup compared to MERO [CWP$^+$09] and TRIAGE [NFH18], respectively. Note that although the runtime of TRIAGE is smaller, its Trojan detection rate is 30% lower than AdaTest.

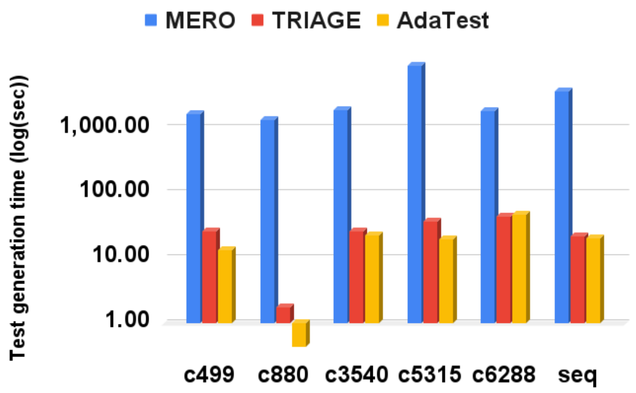


Figure 9.7. Test set generation time comparison between AdaTest and prior works. The runtime shown by the y-axis is represented in the log scale.

9.6.3 AdaTest Architecture Evaluation

The resource utilization of AdaTest depends on the input length and the circuit size. We report the resource utilization results of the evaluated benchmarks in Table 9.3. Figure 9.8 shows that AdaTest architecture achieves approximately linear speedup w.r.t. to the number of CEs. Our hardware design can be scaled up by adding more reward computing engines to parallel the circuit emulation process as AdaTest's computation bottleneck is reward evaluation of the test patterns. Nevertheless, the speedup saturates when N_{CE} is sufficiently high. AdaTest broadcasts the wire values of the circuit response (given a test input) to all CEs via a shared data bus. Each CE scans the DAG buffer and obtains the broadcast wire values to compute the corresponding reward. Therefore, increasing the number of CEs does not lead to extra wire delay. However, more CEs suggest a higher overhead during reward accumulation.

Table 9.3. Resource utilization of the auxiliary circuitry on *c432*,*c880*, *c2670* and *des* benchmarks with default settings ($N_{CE} = 16$) on Zynq ZC706.

Benchmarks	c432	c880	c2670	des
BRAMS	26	36	65	237
DSP48E1	0	0	0	0
KLUTs (emulator usage)	14.9 (0.5)	25.5 (0.6)	61.1 (3.5)	267.9 (26.1)
FFs (emulator usage)	4,440 (80)	5,743 (160)	6,717 (317)	12,943 (1190)

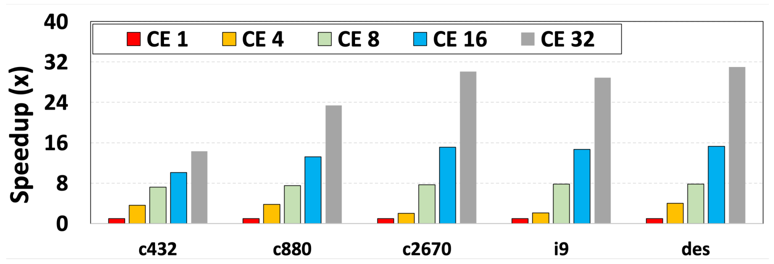


Figure 9.8. AdaTest's scalability to the number of DAG reward computing engines. The speedup is near-linear with N_{CE} on large circuits where reward evaluation is the computation bottleneck.

9.7 Summary

In this chapter, we present a holistic solution to Hardware Trojan detection using adaptive, reinforcement learning-based test pattern generation. To formulate logic testing-based HT detection as an RL problem, we design an innovative reward function to characterize the quality of a test pattern from both static and dynamic aspects. AdaTest progressively expands the test set by identifying test input vectors with high reward values in an iterative approach. AdaTest integrates adaptive sampling to identify and encourage high-reward test patterns, thus accelerating our RL-based input space exploration. Furthermore, AdaTest provides a flexible trade-off between the Trojan coverage and the test set generation time by allowing the defender to customize the termination condition for our adaptive test pattern generation.

We devise AdaTest using a Software/Hardware co-design approach. Particularly, we develop a domain-specific system-on-chip architecture for efficient hardware implementation of AdaTest. Our architecture optimizes reward evaluation via circuit emulation and pipelines the computation of AdaTest. We perform extensive evaluations of AdaTest on various benchmarks and compare its performance with two counterparts, MERO and TRIAGE. Empirical results corroborate that AdaTest achieves superior effectiveness, efficiency, and scalability for HT detection compared to prior works. AdaTest is a *generic* test pattern generation framework, we plan to investigate its performance on other hardware security problems such as logic verification and built-in self-test in our future work.

9.8 Acknowledgements

This work was supported in part by National Science Foundation (NSF) Trust-Hub under award number CNS-2016737, and NSF TILOS under award number CCF-2112665. This chapter, in part, has been published in 2022 ACM Transactions on Embedded Computing Systems (TECS) and appeared as Huili Chen, Xinqiao Zhang, Ke Huang, Farinaz Koushanfar, “AdaTest: Reinforcement Learning and Adaptive Sampling for On-chip Hardware Trojan Detection” [CZHK22]. The dissertation author was the primary author of this material.

Chapter 10

Summary and Future Research Directions

Effective and efficient security assurance techniques are critical for enabling reliable and trustworthy machine learning deployment. Deep learning models are increasingly deployed in various real-world applications ranging from computer vision and natural language processing to biomedical diagnosis and financial analysis. Such a wide deployment raises concerns for both DL model designers and end users. On the one hand, well-trained DL models are valuable assets of the model owner and need to be protected against intellectual property infringement attacks. On the other hand, the end users might be concerned about the safety of the pre-trained model obtained from the third-party model provider since the training process is opaque to the users. Besides the training-time threats, DL models are also vulnerable to inference-time attacks such as adversarial samples and fault injection attacks. This dissertation addresses the IP protection and security assurance challenges of deep learning by provisioning end-to-end, holistic solutions that concurrently optimize the effectiveness and efficiency of the defense while adhering to the resource/energy constraints from the underlying hardware. The research contributions of this dissertation open future directions including:

- **Devising platform-aware intellectual property protection frameworks for deep learning models.** This research topic focuses on designing digital watermarking/fingerprinting techniques suitable for the deep learning domain. I believe that the development of such IP protection frameworks needs to take the model intrinsic and the deployment scenario (e.g., hardware constraints) into account simultaneously.

- **Evaluating the robustness of machine learning systems from a wide spectrum.** This dissertation has investigated the vulnerability of deep learning systems to backdoor attacks and bit flipping-based fault injection attacks. To ensure safe deployment, I believe it is essential for both researchers and practitioners to explore diverse potential attack vectors against DL models. Unveiling the susceptibility of DNNs is also the first step to enhancing the robustness of the DL systems for reliable decision-making.

- **Detecting potential attacks on model inputs or model internals.** The synopsis of my works has revealed training-time as well as run-time threats against DL models and provided holistic defense solutions. My solutions enable effective and efficient detection/mitigation of potential backdoor insertion and fault injection attacks. The systematic defense frameworks in this dissertation are compatible with existing/future defense methods against orthogonal attacks such as adversarial samples and membership inference attacks. I believe that developing a coherent DL defense framework requires strategical adaptation and re-assembly of contemporary defense techniques.

- **Adapting machine learning techniques to solve long-standing hardware security problems.** This dissertation has shown the capability and superior performance of ML techniques for solving existing hardware security problems. ML approaches are intriguing since they enable automated feature learning and have demonstrated unprecedented performance compared to expert-engineered counterparts. Additionally, my solutions suggest that future works shall explore software/hardware co-design to achieve optimal performance of the ML model on the pertinent platform while constraining the overhead.

Bibliography

[AAA$^+$16] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Qiang Cheng, Guoliang Chen, et al. Deep speech 2: End-to-end speech recognition in english and mandarin. In *International conference on machine learning*, pages 173–182, 2016.

[AAA17] Misiker Tadesse Aga, Zelalem Birhanu Aweke, and Todd Austin. When good protections go bad: Exploiting anti-dos measures to accelerate rowhammer attacks. In *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 8–13. IEEE, 2017.

[ABC$^+$18] Yossi Adi, Carsten Baum, Moustapha Cisse, Benny Pinkas, and Joseph Keshet. Turning your weakness into a strength: Watermarking deep neural networks by backdooring. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1615–1631, 2018.

[ACH$^+$10] Eyad Alkassar, Ernie Cohen, Mark Hillebrand, Mikhail Kovalev, and Wolfgang J Paul. Verifying shadow page table algorithms. In *Formal Methods in Computer Aided Design*, pages 267–270. IEEE, 2010.

[ADD21] Sigurd Frej Joel Jørgensen Ankergård, Edlira Dushku, and Nicola Dragoni. State-of-the-art software-based remote attestation: Opportunities and open issues for internet of things. *Sensors*, 21(5):1598, 2021.

[ADM$^+$10] Michel Agoyan, Jean-Max Dutertre, Amir-Pasha Mirbaha, David Naccache, Anne-Lise Ribotta, and Assia Tria. How to flip a bit? In *2010 IEEE 16th International On-Line Testing Symposium*, pages 235–239. IEEE, 2010.

[AG19] Uthman Alsaiari and Fayez Gebali. Hardware trojan detection using reconfigurable assertion checkers. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(7):1575–1586, 2019.

[AH04] Rajat Arora and Michael S Hsiao. Enhancing sat-based bounded model checking using sequential logic implications. In *17th International Conference on VLSI Design. Proceedings.*, pages 784–787. IEEE, 2004.

[AHTA03] Amr T Abdel-Hamid, Sofiene Tahar, and El Mostapha Aboulhamid. Ip watermarking techniques: Survey and comparison. In *The 3rd IEEE International Workshop on System-on-Chip for Real-Time Applications, 2003. Proceedings.*, pages 60–65. IEEE, 2003.

[AJO$^+$18] Oludare Isaac Abiodun, Aman Jantan, Abiodun Esther Omolara, Kemi Victoria Dada, Nachaat AbdElatif Mohamed, and Humaira Arshad. State-of-the-art in artificial neural network applications: A survey. *Heliyon*, 4(11):e00938, 2018.

[AKHS19] Kimia Zamiri Azar, Hadi Mardani Kamali, Houman Homayoun, and Avesta Sasan. Smt attack: Next generation attack on obfuscated circuits with capabilities and performance beyond the sat attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 97–122, 2019.

[AKHS20] Kimia Zamiri Azar, Hadi Mardani Kamali, Houman Homayoun, and Avesta Sasan. Nngsat: Neural network guided sat attack on logic locked complex structures. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9. IEEE, 2020.

[AKS93] Vishwani D Agrawal, Charles R Kime, and Kewal K Saluja. A tutorial on built-in self-test. i. principles. *IEEE Design & Test of Computers*, 10(1):73–82, 1993.

[AM18] Naveed Akhtar and Ajmal Mian. Threat of adversarial attacks on deep learning in computer vision: A survey. *Ieee Access*, 6:14410–14430, 2018.

[Ama22] Amazon. Free machine learning services on aws. https://aws.amazon.com/free/machine-learning/, 2022.

[AMAZ17] Saad Albawi, Tareq Abed Mohammed, and Saad Al-Zawi. Understanding of a convolutional neural network. In *2017 international conference on engineering and technology (ICET)*, pages 1–6. Ieee, 2017.

[AMHH15] Jun Chin Ang, Andri Mirzal, Habibollah Haron, and Haza Nuzly Abdull Hamed. Supervised, unsupervised, and semi-supervised feature selection: a review on gene selection. *IEEE/ACM transactions on computational biology and bioinformatics*, 13(5):971–989, 2015.

[ARC19] Mohammad Al-Rubaie and J Morris Chang. Privacy-preserving machine learning: Threats and solutions. *IEEE Security & Privacy*, 17(2):49–58, 2019.

[ASRF20] Zhezhi He Adnan Siraj Rakin and Deliang Fan. Tbt: Targeted neural network attack with bit trojan - cvpr2020. https://github.com/adnansirajrakin/TBT-CVPR2020, 2020. Accessed: 2021-03-15.

[ASSW13] Frederik Armknecht, Ahmad-Reza Sadeghi, Steffen Schulz, and Christian Wachsmann. A security framework for the analysis and design of software attestation. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1–12, 2013.

[ATG$^+$16] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure linux containers with intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016.

[AZB16] Scott Alfeld, Xiaojin Zhu, and Paul Barford. Data poisoning attacks against autoregressive models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016.

[Bar06] J. Christopher Bare. Attestation and trusted computing. https://courses.cs.washington.edu/courses/csep590/06wi/finalprojects/bare.pdf, 2006.

[Bau09] Alex Clark Baumgarten. *Preventing integrated circuit piracy using reconfigurable logic barriers*. Iowa State University, 2009.

[BBK89] Franc Brglez, David Bryan, and Krzysztof Kozminski. Combinational profiles of sequential benchmark circuits. In *IEEE international symposium on circuits and systems*, volume 3, pages 1929–1934, 1989.

[BBK06] Franc Brglez, David Bryan, and Krzysztof Kozminski. Iscas89 benchmark netlists. https://filebox.ece.vt.edu/~mhsiao/ISCAS89/, September 22, 2006.

[BBM12] R Bhattacharya, S Biswas, and S Mukhopadhyay. Fpga based chip emulation system for test development of analog and mixed signal circuits: A case study of dc–dc buck converter. *Measurement*, 45(8):1997–2020, 2012.

[BEMS$^+$15] Ferdinand Brasser, Brahim El Mahjoub, Ahmad-Reza Sadeghi, Christian Wachsmann, and Patrick Koeberl. Tytan: Tiny trust anchor for tiny devices. In *Proceedings of the 52nd annual design automation conference*, pages 1–6, 2015.

[BHBN14] Swarup Bhunia, Michael S Hsiao, Mainak Banga, and Seetharam Narasimhan. Hardware trojan attacks: Threat analysis and countermeasures. *Proceedings of the IEEE*, 102(8):1229–1247, 2014.

[BKK$^+$19] David Biancolin, Sagar Karandikar, Donggyu Kim, Jack Koenig, Andrew Waterman, Jonathan Bachrach, and Krste Asanovic. Fased: Fpga-accelerated simulation and evaluation of dram. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 330–339. ACM, 2019.

[BLM10] Cristiana Bolchini, Pier Luca Lanzi, and Antonio Miele. A multi-objective genetic algorithm framework for design space exploration of reliable fpga-based systems. In *IEEE Congress on Evolutionary Computation*, pages 1–8. IEEE, 2010.

[BMD$^+$17] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: Sgx cache attacks are practical. *arXiv*, 2017.

[BNL12] Battista Biggio, Blaine Nelson, and Pavel Laskov. Poisoning attacks against support vector machines. *arXiv preprint arXiv:1206.6389*, 2012.

[BNS06] Mikhail Belkin, Partha Niyogi, and Vikas Sindhwani. Manifold regularization: A geometric framework for learning from labeled and unlabeled examples. *Journal of machine learning research*, 7(Nov):2399–2434, 2006.

[BP13] Tiziano Bianchi and Alessandro Piva. Secure watermarking for multimedia content protection: A review of its benefits and open issues. *IEEE signal processing magazine*, 30(2):87–96, 2013.

[BW05] Fahiem Bacchus and Toby Walsh. *Theory and Applications of Satisfiability Testing: 8th International Conference, SAT 2005, St Andrews, Scotland, June 19-23, 2005, Proceedings*, volume 3569. Springer, 2005.

[BW12] Rick Boivie and Peter Williams. Secureblue++: Cpu support for secure execution. *IBM, IBM Research Division, RC25287 (WAT1205-070)*, pages 1–9, 2012.

[BWZ$^+$21] Jiawang Bai, Baoyuan Wu, Yong Zhang, Yiming Li, Zhifeng Li, and Shu-Tao Xia. Targeted attack against deep neural networks via flipping limited weight bits. *arXiv preprint arXiv:2102.10496*, 2021.

[Caf17] Caffe. Model zoo. https://github.com/BVLC/caffe/wiki/Model-Zoo, 2017.

[CB08] Rajat Subhra Chakraborty and Swarup Bhunia. Hardware protection and authentication through netlist level obfuscation. In *Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design*, pages 674–677. IEEE Press, 2008.

[CB14] Brice Colombier and Lilian Bossuet. Survey of hardware protection of design data for integrated circuits and intellectual properties. *IET Computers & Digital Techniques*, 8(6):274–287, 2014.

[CCB$^+$18] Bryant Chen, Wilka Carvalho, Nathalie Baracaldo, Heiko Ludwig, Benjamin Edwards, Taesung Lee, Ian Molloy, and Biplav Srivastava. Detecting backdoor attacks on deep neural networks by activation clustering. *arXiv preprint arXiv:1811.03728*, 2018.

[CCB19] Prabuddha Chakraborty, Jonathan Cruz, and Swarup Bhunia. Surf: Joint structural functional attack on logic locking. In *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 181–190. IEEE, 2019.

[CD06] Charles J Colbourn and Jeffrey H Dinitz. *Handbook of combinatorial designs*. CRC press, 2006.

[CELT78] TC Chen, Kapali P Eswaran, Vincent Y Lum, and C Tung. Simplified odd-even sort using multiple shift-register loops. *International Journal of Computer & Information Sciences*, 7(3):295–314, 1978.

[CES16] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. *ACM SIGARCH Computer Architecture News*, 44(3):367–379, 2016.

[CFR$^+$19] Huili Chen, Cheng Fu, Bita Darvish Rouhani, Jishen Zhao, and Farinaz Koushanfar. Deepattest: an end-to-end attestation framework for deep neural networks. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 487–498. IEEE, 2019.

[CFZK19a] Huili Chen, Cheng Fu, Jishen Zhao, and Farinaz Koushanfar. Deepinspect: A black-box trojan detection and mitigation framework for deep neural networks. In *IJCAI*, volume 2, page 8, 2019.

[CFZK19b] Huili Chen, Cheng Fu, Jishen Zhao, and Farinaz Koushanfar. Genunlock: An automated genetic algorithm framework for unlocking logic encryption. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2019.

[CFZK21] Huili Chen, Cheng Fu, Jishen Zhao, and Farinaz Koushanfar. Proflip: Targeted trojan attack with progressive bit flips. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 7718–7727, 2021.

[CHM$^+$15] Anna Choromanska, Mikael Henaff, Michael Mathieu, Gérard Ben Arous, and Yann LeCun. The loss surfaces of multilayer networks. In *Artificial Intelligence and Statistics*, 2015.

[CKLS96] Ingemar J Cox, Joe Kilian, Tom Leighton, and Talal Shamoon. Secure spread spectrum watermarking for images, audio and video. In *Proceedings of 3rd IEEE international conference on image processing*, volume 3, pages 243–246. IEEE, 1996.

[CKLS97] Ingemar J Cox, Joe Kilian, F Thomson Leighton, and Talal Shamoon. Secure spread spectrum watermarking for multimedia. *IEEE transactions on image processing*, 6(12):1673–1687, 1997.

[CLC+09] Shih-Lun Chen, Ho-Yin Lee, Chiung-An Chen, Hong-Yi Huang, and Ching-Hsing Luo. Wireless body sensor network with adaptive low-power design for biometrics and healthcare applications. *IEEE Systems Journal*, 3(4):398–409, 2009.

[CLD+17] Victor Costan, Ilia Lebedev, Srinivas Devadas, et al. Secure processors part ii: Intel sgx security analysis and mit sanctum architecture. *Foundations and Trends® in Electronic Design Automation*, 11(3):249–361, 2017.

[CLL+17] Xinyun Chen, Chang Liu, Bo Li, Kimberly Lu, and Dawn Song. Targeted backdoor attacks on deep learning systems using data poisoning. *arXiv preprint arXiv:1712.05526*, 2017.

[CLMZ21] Siyuan Cheng, Yingqi Liu, Shiqing Ma, and Xiangyu Zhang. Deep feature space trojan attack of neural networks by controlled detoxification. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 1148–1156, 2021.

[CM97] Stephen A Cook and David G Mitchell. Finding hard instances of the satisfiability problem. In *Satisfiability problem: theory and applications: DIMACS workshop*, volume 35, pages 1–17, 1997.

[CMB+07] Ingemar Cox, Matthew Miller, Jeffrey Bloom, Jessica Fridrich, and Ton Kalker. *Digital watermarking and steganography*. Morgan kaufmann, 2007.

[CMD+19] Brice Colombier, Alexandre Menu, Jean-Max Dutertre, Pierre-Alain Moëllic, Jean-Baptiste Rigaud, and Jean-Luc Danger. Laser-induced single-bit faults in flash memory: Instructions corruption on a 32-bit microcontroller. In *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 1–10. IEEE, 2019.

[CMN+20] Robby Costales, Chengzhi Mao, Raphael Norwitz, Bryan Kim, and Junfeng Yang. Live trojan attacks on deep neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pages 796–797, 2020.

[CMS12] Dan C Cireşan, Ueli Meier, and Jürgen Schmidhuber. Transfer learning for latin and chinese characters with deep neural networks. In *The 2012 International Joint Conference on Neural Networks (IJCNN)*, pages 1–6. IEEE, 2012.

[CNB09] Rajat Subhra Chakraborty, Seetharam Narasimhan, and Swarup Bhunia. Hardware trojan: Threats and emerging solutions. In *2009 IEEE International high level design validation and test workshop*, pages 166–171. IEEE, 2009.

[CRF+19] Huili Chen, Bita Darvish Rouhani, Cheng Fu, Jishen Zhao, and Farinaz Koushanfar. Deepmarks: A secure fingerprinting framework for digital rights management of deep learning models. In *Proceedings of the 2019 on International Conference on Multimedia Retrieval*, pages 105–113, 2019.

[CRK19] Huili Chen, Bita Darvish Rouhani, and Farinaz Koushanfar. Blackmarks: Blackbox multibit watermarking for deep neural networks. *arXiv preprint arXiv:1904.00344*, 2019.

[CRK20] Huili Chen, Bita Darvish Rouhani, and Farinaz Koushanfar. Specmark: A spectral watermarking framework for ip protection of speech recognition systems. In *INTERSPEECH*, pages 2312–2316, 2020.

[CTPB18] Edward Chou, Florian Tramèr, Giancarlo Pellegrino, and Dan Boneh. Sentinet: Detecting physical attacks against deep learning systems. *arXiv preprint arXiv:1812.00292*, 2018.

[CW08] Ronan Collobert and Jason Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th International Conference on Machine Learning*. ACM, 2008.

[CWB$^+$17] Jianning Chi, Ekta Walia, Paul Babyn, Jimmy Wang, Gary Groot, and Mark Eramian. Thyroid nodule classification in ultrasound images by fine-tuning deep convolutional neural network. *Journal of digital imaging*, 30(4):477–486, 2017.

[CWB$^+$21] Xinyun Chen, Wenxiao Wang, Chris Bender, Yiming Ding, Ruoxi Jia, Bo Li, and Dawn Song. Refit: a unified watermark removal framework for deep learning systems with limited data. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, pages 321–335, 2021.

[CWD$^+$18] Antonia Creswell, Tom White, Vincent Dumoulin, Kai Arulkumaran, Biswa Sengupta, and Anil A Bharath. Generative adversarial networks: An overview. *IEEE signal processing magazine*, 35(1):53–65, 2018.

[CWD$^+$19] Xinyun Chen, Wenxiao Wang, Yiming Ding, Chris Bender, Ruoxi Jia, Bo Li, and Dawn Song. Leveraging unlabeled data for watermark removal of deep neural networks. In *ICML workshop on Security and Privacy of Machine Learning*, pages 1–6, 2019.

[CWP$^+$09] Rajat Subhra Chakraborty, Francis Wolff, Somnath Paul, Christos Papachristou, and Swarup Bhunia. Mero: A statistical approach for hardware trojan detection. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 396–410. Springer, 2009.

[CZHK22] Huili Chen, Xinqiao Zhang, Ke Huang, and Farinaz Koushanfar. Adatest: Reinforcement learning and adaptive sampling for on-chip hardware trojan detection. *ACM Transactions on Embedded Computing Systems (TECS)*, 2022.

[DAR09] Gaurav Dhiman, Raid Ayoub, and Tajana Rosing. Pdram: A hybrid pram and dram main memory system. In *2009 46th ACM/IEEE Design Automation Conference*, pages 664–669. IEEE, 2009.

[DCLT18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[DCV93] C Dufaza, C Chevalier, and LFC Lew Yan Voon. Lfsrom: A hardware test pattern generator for deterministic iscas85 test sets. In *Proceedings of 1993 IEEE 2nd Asian Test Symposium (ATS)*, pages 160–165. IEEE, 1993.

[DDS$^+$09] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[Del04] Bethany Delman. *Genetic Algorithms in Cryptography*. PhD thesis, Citeseer, 2004.

[Dev22] Google Developers. Introduction to generative adversarial networks. https://developers.google.com/machine-learning/gan, 2022.

[DF19] Sophie Dupuis and Marie-Lise Flottes. Logic locking: A survey of proposed methods and evaluation metrics. *Journal of Electronic Testing*, 35(3):273–291, 2019.

[DOM02] Stephan Dreiseitl and Lucila Ohno-Machado. Logistic regression and artificial neural network classification models: a methodology review. *Journal of biomedical informatics*, 35(5-6):352–359, 2002.

[DPAM02] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.

[DRCK19] Bita Darvish Rouhani, Huili Chen, and Farinaz Koushanfar. Deepsigns: An end-to-end watermarking framework for ownership protection of deep neural networks. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 485–497, 2019.

[dRFV$^+$21] Finn de Ridder, Pietro Frigo, Emanuele Vannacci, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. {SMASH}: Synchronized many-sided rowhammer attacks from {JavaScript}. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1001–1018, 2021.

[DS19] Kemal Davaslioglu and Yalin E Sagduyu. Trojan attacks on wireless signal classification with adversarial machine learning. In *2019 IEEE International Symposium on Dynamic Spectrum Access Networks (DySPAN)*, pages 1–6. IEEE, 2019.

[dSAF13] Pedro Vieira dos Santos, José Carlos Alves, and João Canas Ferreira. A framework for hardware cellular genetic algorithms: An application to spectrum allocation in cognitive radio. In *2013 23rd International Conference on Field programmable Logic and Applications*, pages 1–4. IEEE, 2013.

[DZDW18] Happiness Ugochi Dike, Yimin Zhou, Kranthi Kumar Deveerasetty, and Qingtian Wu. Unsupervised learning based on artificial neural network: A review. In *2018 IEEE International Conference on Cyborg and Bionic Systems (CBS)*, pages 322–327. IEEE, 2018.

[EBGLOUM22] Xabier Echeberria-Barrio, Amaia Gil-Lerchundi, Raul Orduna-Urrutia, and Iñigo Mendialdua. Optimized parameter search approach for weight modification attack targeting deep learning models. *Applied Sciences*, 12(8):3725, 2022.

[EKN$^+$17] Andre Esteva, Brett Kuprel, Roberto A Novoa, Justin Ko, Susan M Swetter, Helen M Blau, and Sebastian Thrun. Dermatologist-level classification of skin cancer with deep neural networks. *Nature*, 542(7639):115, 2017.

[EMGT15] Mohamed El Massad, Siddharth Garg, and Mahesh V Tripunitara. Integrated circuit (ic) decamouflaging: Reverse engineering camouflaged ics within minutes. In *NDSS*, pages 1–14, 2015.

[Erb93] Randall J Erb. Introduction to backpropagation neural network computation. *Pharmaceutical research*, 10(2):165–170, 1993.

[FDFC$^+$18] C Fu, A Di Fulvio, SD Clarke, D Wentzloff, SA Pozzi, and HS Kim. Artificial neural network algorithms for pulse shape discrimination and recovery of piled-up pulses in organic scintillators. *Annals of Nuclear Energy*, 120:410–421, 2018.

[FG20] Barbara Franci and Sergio Grammatico. A game–theoretic approach for generative adversarial networks. In *2020 59th IEEE Conference on Decision and Control (CDC)*, pages 1646–1651. IEEE, 2020.

[FJR15] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. Model inversion attacks that exploit confidence information and basic countermeasures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1322–1333. ACM, 2015.

[FK04] Borko Furht and Darko Kirovski. *Multimedia security handbook.* CRC press, 2004.

[FM07] Zhaohui Fu and Sharad Malik. Extracting logic circuit structure from conjunctive normal form descriptions. In *20th International Conference on VLSI Design held jointly with 6th International Conference on Embedded Systems (VLSID'07)*, pages 37–42. IEEE, 2007.

[FSK$^+$17] Marc Fyrbiak, Sebastian Strauß, Christian Kison, Sebastian Wallat, Malte Elson, Nikol Rummel, and Christof Paar. Hardware reverse engineering: Overview and open challenges. In *2017 IEEE 2nd International Verification and Security Workshop (IVSW)*, pages 88–94. IEEE, 2017.

[FVH$^+$20] Pietro Frigo, Emanuele Vannacc, Hasan Hassan, Victor Van Der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Trrespass: Exploiting the many sides of target row refresh. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 747–762. IEEE, 2020.

[GBDL$^+$16] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *International Conference on Machine Learning*, pages 201–210, 2016.

[GESM17] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on intel sgx. In *Proceedings of the 10th European Workshop on Systems Security*, pages 1–6, 2017.

[GG12] Deepti Gupta and Shabina Ghafir. An overview of methods maintaining diversity in genetic algorithms. *International journal of emerging technology and advanced engineering*, 2(5):56–60, 2012.

[GGS14] Richa Gupta, Sunny Gupta, and Anuradha Singhal. Importance and techniques of information hiding: A review. *arXiv preprint arXiv:1404.3063*, 2014.

[GHZZ18] Shuqin Gu, Yuexian Hou, Lipeng Zhang, and Yazhou Zhang. Regularizing deep neural networks with an ensemble-based decorrelation method. In *IJCAI*, pages 2177–2183, 2018.

[GLDGG19] Tianyu Gu, Kang Liu, Brendan Dolan-Gavitt, and Siddharth Garg. Badnets: Evaluating backdooring attacks on deep neural networks. *IEEE Access*, 7:47230–47244, 2019.

[GLS$^+$18] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. Another flip in the wall of rowhammer defenses. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 245–261. IEEE, 2018.

[GM10] Baisa L Gunjal and RR Manthalkar. An overview of transform domain robust digital image watermarking algorithms. *Journal of Emerging Trends in Computing and Information Sciences*, 2(1):37–42, 2010.

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM (JACM)*, 43(3):431–473, 1996.

[GP18] Jia Guo and Miodrag Potkonjak. Watermarking deep neural networks for embedded systems. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2018.

[GP19] Jia Guo and Miodrag Potkonjak. Evolutionary trigger set generation for dnn black-box watermarking. *arXiv preprint arXiv:1906.04411*, 2019.

[GPAM$^+$20] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks. *Communications of the ACM*, 63(11):139–144, 2020.

[GSS14] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[GSW$^+$21] Jie Gui, Zhenan Sun, Yonggang Wen, Dacheng Tao, and Jieping Ye. A review on generative adversarial networks: Algorithms, theory, and applications. *IEEE Transactions on Knowledge and Data Engineering*, 2021.

[GT80] Lawrence H Goldstein and Evelyn L Thigpen. Scoap: Sandia controllability/observability analysis program. In *Proceedings of the 17th Design Automation Conference*, pages 190–196, 1980.

[GTCM20] Sorin Grigorescu, Bogdan Trasnea, Tiberiu Cocias, and Gigel Macesanu. A survey of deep learning techniques for autonomous driving. *Journal of Field Robotics*, 37(3):362–386, 2020.

[GTL14] Liucheng Guo, David B Thomas, and Wayne Luk. Automated framework for general-purpose genetic algorithms in fpgas. In *European Conference on the Applications of Evolutionary Computation*, pages 714–725. Springer, 2014.

[GTS$^+$18] Karan Grover, Shruti Tople, Shweta Shinde, Ranjita Bhagwan, and Ramachandran Ramjee. Privado: Practical and secure dnn inference with enclaves. *arXiv preprint arXiv:1810.00602*, 2018.

[GZQ$^+$20] Shangwei Guo, Tianwei Zhang, Han Qiu, Yi Zeng, Tao Xiang, and Yang Liu. Fine-tuning is not enough: A simple yet effective watermark removal attack for dnn models. *arXiv preprint arXiv:2009.08697*, 2020.

[Har17] Danny Harnik. Impressions of intel sgx performance. https://medium.com/@danny_harnik/impressions-of-intel-sgx-performance-22442093595a, 2017.

[HCC$^+$14] Awni Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, et al. Deep speech: Scaling up end-to-end speech recognition. *arXiv preprint arXiv:1412.5567*, 2014.

[Hei18] Dominiek Ter Heide. A closer look at ethereum signatures. https://hackernoon.com/a-closer-look-at-ethereum-signatures-5784c14abecc, Feb, 2018.

[HFK$^+$19] Sanghyun Hong, Pietro Frigo, Yiğitcan Kaya, Cristiano Giuffrida, and Tudor Dumitraș. Terminal brain damage: Exposing the graceless degradation in deep neural networks under hardware fault attacks. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 497–514, 2019.

[HK99] Frank Hartung and Martin Kutter. Multimedia watermarking techniques. *Proceedings of the IEEE*, 87(7):1079–1107, 1999.

[HK19] Siam U Hussain and Farinaz Koushanfar. Fase: Fpga acceleration of secure function evaluation. In *Field-Programmable Custom Computing Machines*, 2019.

[HKM$^+$17] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, et al. Ese: Efficient speech recognition engine with sparse lstm on fpga. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 75–84, 2017.

[HLZ$^+$15] Johann Hauswald, Michael A Laurenzano, Yunqi Zhang, Cheng Li, Austin Rovinski, Arjun Khurana, Ronald G Dreslinski, Trevor Mudge, Vinicius Petrucci, Lingjia Tang, et al. Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 223–238, 2015.

[HMD15] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[HN05] Wang Hongjun and Li Na. An algorithm of digital image watermark based on multiresolution wavelet analysis. In *Proceedings of 2005 IEEE International Workshop on VLSI Design and Video Technology, 2005.*, pages 272–275. IEEE, 2005.

[Hoy18] Matthew B Hoy. Alexa, siri, cortana, and more: an introduction to voice assistants. *Medical reference services quarterly*, 37(1):81–88, 2018.

[HPTD15] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, pages 1135–1143, 2015.

[HRGK18] Siam U Hussain, Bita Darvish Rouhani, Mohammad Ghasemzadeh, and Farinaz Koushanfar. Maxelerator: Fpga accelerator for privacy preserving multiply-accumulate (mac) on cloud servers. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2018.

[HRL$^+$20] Zhezhi He, Adnan Siraj Rakin, Jingtao Li, Chaitali Chakrabarti, and Deliang Fan. Defending and harnessing the bit-flip based adversarial weight attack. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 14095–14103, 2020.

[HTGW18] Ehsan Hesamifard, Hassan Takabi, Mehdi Ghasemi, and Rebecca N Wright. Privacy-preserving machine learning as a service. *Proc. Priv. Enhancing Technol.*, 2018(3):123–142, 2018.

[HYH99] Mark C Hansen, Hakan Yalcin, and John P Hayes. Unveiling the iscas-85 benchmarks: A case study in reverse engineering. *IEEE Design & Test of Computers*, 16(3):72–80, 1999.

[HYR21] Zhaokun Han, Muhammad Yasin, and Jeyavijayan JV Rajendran. Does logic locking work with {EDA} tools? In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1055–1072, 2021.

[HYS$^+$06] Mike Hutton, Richard Yuan, Jay Schleicher, Gregg Baeckler, Sammy Cheung, Kar Keng Chua, and Hee Kong Phoon. A methodology for fpga to structured-asic synthesis and verification. In *Proceedings of the Design Automation & Test in Europe Conference*, volume 2, pages 1–6. IEEE, 2006.

[HZC$^+$17] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint*, 2017.

[HZRS16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[Int17] Intel. Intel software guard extensions sdk. https://software.intel.com/en-us/sgx-sdk-dev-reference-sgx-get-trusted-time, 2017.

[IZZE17] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A Efros. Image-to-image translation with conditional adversarial networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1125–1134, 2017.

[JPM$^+$18] Animesh Jain, Amar Phanishayee, Jason Mars, Lingjia Tang, and Gennady Pekhimenko. Gist: Efficient data encoding for deep neural network training. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 776–789. IEEE, 2018.

[JSF$^+$20] Mojan Javaheripi, Mohammad Samragh, Gregory Fields, Tara Javidi, and Farinaz Koushanfar. Cleann: Accelerated trojan shield for embedded neural networks. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9. IEEE, 2020.

[JVC18] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. GAZELLE: A low latency framework for secure neural network inference. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1651–1669, 2018.

[JWL$^+$20] Ying Jin, Yunbo Wang, Mingsheng Long, Jianmin Wang, S Yu Philip, and Jiaguang Sun. A multi-player minimax game for generative adversarial networks. In *2020 IEEE International Conference on Multimedia and Expo (ICME)*, pages 1–6. IEEE, 2020.

[KAFT22] Hadi Mardani Kamali, Kimia Zamiri Azar, Farimah Farahmandi, and Mark Tehranipoor. Advances in logic locking: Past, present, and prospects. *Cryptology ePrint Archive*, 2022.

[Kan17] Hyeonwoo Kang. Pytorch implementation of conditional generative adversarial networks (cgan) and conditional deep convolutional generative adversarial networks (cdcgan). https://github.com/znxlwm/pytorch-MNIST-CelebA-cGAN-cDCGAN, 2017.

[Kat16] Yash Katariya. Mnist cnn benchmark. https://github.com/yashk2810/MNIST-Keras/tree/master/Notebook, 2016.

[KB18] Veton Kepuska and Gamal Bohouta. Next-generation of virtual personal assistants (microsoft cortana, apple siri, amazon alexa and google home). In *2018 IEEE 8th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 99–103. IEEE, 2018.

[KDK$^+$14] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. *ACM SIGARCH Computer Architecture News*, 42(3):361–372, 2014.

[KGB$^+$16] Alexey Kurakin, Ian Goodfellow, Samy Bengio, et al. Adversarial examples in the physical world, 2016.

[KGGY20] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. Rambleed: Reading bits in memory without accessing them. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 695–711. IEEE, 2020.

[KH$^+$09] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[KHF$^+$19] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2019.

[KK04] Deepa Kundur and Kannan Karthik. Video fingerprinting and encryption principles for digital rights management. *Proceedings of the IEEE*, 92(6):918–932, 2004.

[KK09] Hiam M Khoury and Vineet R Kamat. Evaluation of position tracking technologies for user localization in indoor construction environments. *Automation in Construction*, 18(4):444–457, 2009.

[KKP$^+$14] Joonho Kong, Farinaz Koushanfar, Praveen K Pendyala, Ahmad-Reza Sadeghi, and Christian Wachsmann. Pufatt: Embedded platform attestation based on novel processor-based pufs. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2014.

[KLM96] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[KLMS$^+$98] Andrew B Kahng, John Lach, William H Mangione-Smith, Stefanus Mantik, Igor L Markov, Miodrag Potkonjak, Paul Tucker, Huijuan Wang, and Gregory Wolfe. Watermarking techniques for intellectual property protection. In *Proceedings of the 35th annual Design Automation Conference*, pages 776–781, 1998.

[KLMS$^+$01] Andrew B Kahng, John Lach, William H Mangione-Smith, Stefanus Mantik, Igor L Markov, Miodrag Potkonjak, Paul Tucker, Huijuan Wang, and Gregory Wolfe. Constraint-based watermarking techniques for design ip protection. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(10):1236–1252, 2001.

[KM03] Darko Kirovski and Henrique S Malvar. Spread-spectrum watermarking of audio signals. *IEEE transactions on signal processing*, 51(4):1020–1033, 2003.

[KM09] Negar Kiyavash and Pierre Moulin. Performance of orthogonal fingerprinting codes under worst-case noise. *IEEE Transactions on Information Forensics and Security*, 4(3):293–301, 2009.

[KPY$^+$15] Yong-Deok Kim, Eunhyeok Park, Sungjoo Yoo, Taelim Choi, Lu Yang, and Dongjun Shin. Compression of deep convolutional neural networks for fast and low power mobile applications. *arXiv preprint arXiv:1511.06530*, 2015.

[KRR12] Ramesh Karri, Jeyavijayan Rajendran, and Kurt Rosenfeld. Trojan taxonomy. In *Introduction to hardware security and trust*, pages 325–338. Springer, 2012.

[KRRT10] Ramesh Karri, Jeyavijayan Rajendran, Kurt Rosenfeld, and Mohammad Tehranipoor. Trustworthy hardware: Identifying and classifying hardware trojans. *Computer*, 43(10):39–46, 2010.

[KSSV14] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. Trustlite: A security architecture for tiny embedded devices. In *Proceedings of the Ninth European Conference on Computer Systems*, pages 1–14, 2014.

[Kum13] Anit Kumar. Encoding schemes in genetic algorithm. *International Journal of Advanced Research in IT and Engineering*, 2(3):1–7, 2013.

[KVH00] Martin Kutter, Sviatoslav V Voloshynovskiy, and Alexander Herrigel. Watermark copy attack. In *Security and Watermarking of Multimedia Contents II*, volume 3971, pages 371–380. SPIE, 2000.

[KZR04] Ahmed Usman Khalid, Zeljko Zilic, and Katarzyna Radecka. Fpga emulation of quantum circuits. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings.*, pages 310–315. IEEE, 2004.

[LBBH98] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436, 2015.

[LBM06] Julia A Lasserre, Christopher M Bishop, and Thomas P Minka. Principled hybrids of generative and discriminative models. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, volume 1, pages 87–94. IEEE, 2006.

[Lei21] Chen Lei. Deep reinforcement learning. In *Deep Learning and Practice with MindSpore*, pages 217–243. Springer, 2021.

[LHM14] Yu Liu, Ke Huang, and Yiorgos Makris. Hardware trojan detection through golden chip-free statistical side-channel fingerprinting. In *Proceedings of the 51st Annual Design Automation Conference*, pages 1–6, 2014.

[LIM09] ARM LIMITED. Arm security technology - building a secure system using trustzone technology. 2009.

[LKG$^+$09] Lang Lin, Markus Kasper, Tim Güneysu, Christof Paar, and Wayne Burleson. Trojan side-channels: Lightweight hardware trojans through side-channel engineering. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 382–395. Springer, 2009.

[LLP$^+$19] Taegyeong Lee, Zhiqi Lin, Saumay Pushp, Caihua Li, Yunxin Liu, Youngki Lee, Fengyuan Xu, Chenren Xu, Lintao Zhang, and Junehwa Song. Occlumency: Privacy-preserving remote deep-learning inference using sgx. In *The 25th Annual International Conference on Mobile Computing and Networking*, pages 1–17, 2019.

[LLZ16] He Li, Qiang Liu, and Jiliang Zhang. A survey of hardware trojan threat and defense. *Integration*, 55:426–437, 2016.

[LMA$^+$18] Yingqi Liu, Shiqing Ma, Yousra Aafer, Wen-Chuan Lee, Juan Zhai, Weihang Wang, and X. Zhang. Trojaning attack on neural networks. In *NDSS*, 2018.

[LMPT20] Erwan Le Merrer, Patrick Perez, and Gilles Trédan. Adversarial frontier stitching for remote neural network watermarking. *Neural Computing and Applications*, 32(13):9233–9244, 2020.

[LNM$^+$17] Shuangchen Li, Dimin Niu, Krishna T Malladi, Hongzhong Zheng, Bob Brennan, and Yuan Xie. Drisa: A dram-based reconfigurable in-situ accelerator. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 288–301. IEEE, 2017.

[LSG$^+$17] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside sgx enclaves with branch shadowing. In *26th USENIX Security Symposium, USENIX Security*, pages 16–18, 2017.

[LSM$^+$17] Meng Li, Kaveh Shamsi, Travis Meade, Zheng Zhao, Bei Yu, Yier Jin, and David Z Pan. Provably secure camouflaging strategy for ic protection. *IEEE transactions on computer-aided design of integrated circuits and systems*, 38(8):1399–1412, 2017.

[LSN$^+$17] Bo Li, Tara N Sainath, Arun Narayanan, Joe Caroselli, Michiel Bacchiani, Ananya Misra, Izhak Shafran, Hasim Sak, Golan Pundak, Kean K Chin, et al. Acoustic modeling for google home. In *Interspeech*, pages 399–403, 2017.

[LST$^+$16] Geert Litjens, Clara I Sánchez, Nadya Timofeeva, Meyke Hermsen, Iris Nagtegaal, Iringo Kovacs, Christina Hulsbergen-Van De Kaa, Peter Bult, Bram Van Ginneken, and Jeroen Van Der Laak. Deep learning as a tool for increased accuracy and efficiency of histopathological diagnosis. *Scientific reports*, 6(1):1–11, 2016.

[LTA16] Darryl Lin, Sachin Talathi, and Sreekanth Annapureddy. Fixed point quantization of deep convolutional networks. In *International conference on machine learning*, pages 2849–2858. PMLR, 2016.

[Lu04] Chun-Shien Lu. *Multimedia Security: Steganography and Digital Watermarking Techniques for Protection of Intellectual Property: Steganography and Digital Watermarking Techniques for Protection of Intellectual Property*. Igi Global, 2004.

[LVHM15] Yu Liu, Georgios Volanis, Ke Huang, and Yiorgos Makris. Concurrent hardware trojan detection in wireless cryptographic ics. In *2015 IEEE International Test Conference (ITC)*, pages 1–8. IEEE, 2015.

[LWL17] Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. Thinet: A filter level pruning method for deep neural network compression. In *Proceedings of the IEEE international conference on computer vision*, pages 5058–5066, 2017.

[LWS12] Wenchao Li, Zach Wasson, and Sanjit A Seshia. Reverse engineering circuits using behavioral pattern mining. In *2012 IEEE international symposium on hardware-oriented security and trust*, pages 83–88. IEEE, 2012.

[LXS17] Yuntao Liu, Yang Xie, and Ankur Srivastava. Neural trojans. In *2017 IEEE International Conference on Computer Design (ICCD)*, pages 45–48. IEEE, 2017.

[LZZ$^+$08] Na Li, Xiaoshi Zheng, Yanling Zhao, Huimin Wu, and Shifeng Li. Robust algorithm of digital image watermarking based on discrete wavelet transform. In *2008 International Symposium on Electronic Commerce and Security*, pages 942–945. IEEE, 2008.

[MAK$^+$17] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. {ROTE}: Rollback protection for trusted execution. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1289–1306, 2017.

[Man12] Theodore W. Manikas. Mcnc benchmark netlists. https://s2.smu.edu/~manikas/Benchmarks/MCNC_Benchmark_Netlists.html, June 28, 2012.

[Mar21] Dale Markowitz. Transformers, explained: Understand the model behind gpt-3, bert, and t5. https://daleonai.com/transformers-explained, May 2021.

[McC85] Edward J McCluskey. Built-in self-test techniques. *IEEE Design & Test of Computers*, 2(2):21–28, 1985.

[Mem02] Nasir Memon. Information hiding, digital watermarking and steganography. https://eeweb.engineering.nyu.edu/~yao/EE4414/memon_F05_v2.pdf, 202.

[MG$^+$95] Brad L Miller, David E Goldberg, et al. Genetic algorithms, tournament selection, and the effects of noise. *Complex systems*, 9(3):193–212, 1995.

[MGBD$^+$17] Luis Muñoz-González, Battista Biggio, Ambra Demontis, Andrea Paudice, Vasin Wongrassamee, Emil C Lupu, and Fabio Roli. Towards poisoning of deep learning algorithms with back-gradient optimization. In *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, pages 27–38, 2017.

[MGDC$^+$18] Pieter Maene, Johannes Götzfried, Ruan De Clercq, Tilo Müller, Felix Freiling, and Ingrid Verbauwhede. Hardware-based trusted computing architectures for isolation and attestation. *IEEE Transactions on Computers*, 67(3):361–374, 2018.

[MGK$^+$17] Rowan McAllister, Yarin Gal, Alex Kendall, Mark Van Der Wilk, Amar Shah, Roberto Cipolla, and Adrian Weller. Concrete problems for autonomous vehicle safety: Advantages of bayesian deep learning. International Joint Conferences on Artificial Intelligence, Inc., 2017.

[MHZ$^+$08] Maciej A Mazurowski, Piotr A Habas, Jacek M Zurada, Joseph Y Lo, Jay A Baker, and Georgia D Tourassi. Training neural network classifiers for medical decision making: The effects of imbalanced datasets on classification performance. *Neural networks*, 21(2-3):427–436, 2008.

[Mig17] Szymon Migacz. 8-bit inference with tensorrt. In *GPU technology conference*, volume 2, page 5, 2017.

[MK01] JR Hernandez Martin and Martin Kutter. Information retrieval in digital watermarking. *IEEE Communications magazine*, 39(8):110–116, 2001.

[MK19] Onur Mutlu and Jeremie S Kim. Rowhammer: A retrospective. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(8):1555–1571, 2019.

[MKG$^+$15] Samer Moein, Salman Khan, T Aaron Gulliver, Fayez Gebali, and M Watheq El-Kharashi. An attribute based classification of hardware trojans. In *2015 Tenth International Conference on Computer Engineering & Systems (ICCES)*, pages 351–356. IEEE, 2015.

[MKS$^+$13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[MMS$^+$17] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. *arXiv preprint arXiv:1706.06083*, 2017.

[MO03] Pierre Moulin and Joseph A O'Sullivan. Information-theoretic analysis of information hiding. *IEEE Transactions on information theory*, 49(3):563–593, 2003.

[MP14] HN Manjesh and GR Pradyumna. Fpga prototyping of universal asynchronous receiver transmitter (uart) with low power tpg based bist architecture. 2014.

[MPL$^+$17] Azalia Mirhoseini, Hieu Pham, Quoc V Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device placement optimization with reinforcement learning. In *International Conference on Machine Learning*, pages 2430–2439. PMLR, 2017.

[MR95] Rajeev Motwani and Prabhakar Raghavan. *Randomized algorithms*. Cambridge university press, 1995.

[MRIP17] Amin Malekpour, Roshan Ragel, Aleksandar Ignjatovic, and Sri Parameswaran. Trojanguard: Simple and effective hardware trojan mitigation techniques for pipelined mpsocs. In *Proceedings of the 54th Annual Design Automation Conference 2017*, pages 1–6, 2017.

[Mut17] Onur Mutlu. The rowhammer problem and other issues we may face as memory becomes denser. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 1116–1121. IEEE, 2017.

[MV07] Panagiotis Manolios and Daron Vroon. Efficient circuit to cnf conversion. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 4–9. Springer, 2007.

[MZJ16] Travis Meade, Shaojie Zhang, and Yier Jin. Netlist reverse engineering for high-level functionality reconstruction. In *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 655–660. IEEE, 2016.

[Nar20] Sean Naren. Speech recognition using deepspeech2. https://github.com/SeanNaren/deepspeech.pytorch, 2020.

[NDC$^+$12] Seetharam Narasimhan, Dongdong Du, Rajat Subhra Chakraborty, Somnath Paul, Francis G Wolff, Christos A Papachristou, Kaushik Roy, and Swarup Bhunia. Hardware trojan detection by multiple-parameter side-channel analysis. *IEEE Transactions on computers*, 62(11):2183–2195, 2012.

[NER$^+$19] Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, Michael Steiner, and Gene Tsudik. Vrased: A verified hardware/software co-design for remote attestation. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1429–1446, 2019.

[NFH18] MA Nourian, Mahdi Fazeli, and David Hély. Hardware trojan detection using an advised genetic algorithm based logic testing. *Journal of Electronic Testing*, 34(4):461–470, 2018.

[NIS] NIST. Trojai: Trojans in artificial intelligence. https://pages.nist.gov/trojai/. Accessed: 2021-03-12.

[NKFL18] Anusha Nagabandi, Gregory Kahn, Ronald S Fearing, and Sergey Levine. Neural network dynamics for model-based deep reinforcement learning with model-free fine-tuning. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 7559–7566. IEEE, 2018.

[NR13] Jordi Nin and Sergio Ricciardi. Digital watermarking techniques and security issues in the information and communication society. In *2013 27th International Conference on Advanced Information Networking and Applications Workshops*, pages 1553–1558. IEEE, 2013.

[NUSS18] Yuki Nagai, Yusuke Uchida, Shigeyuki Sakazawa, and Shin'ichi Satoh. Digital watermarking for deep neural networks. *International Journal of Multimedia Information Retrieval*, 7(1):3–16, 2018.

[Obi98] Marek Obitko. Introduction to genetic algorithms. https://www.obitko.com/tutorials/genetic-algorithms/, 1998.

[OMPZ] Andrea Vedaldi Omkar M. Parkhi and Andrew Zisserman. Vgg face dataset. https://www.robots.ox.ac.uk/~vgg/data/vgg_face/. Accessed: 2019-02-10.

[OSGP$^+$17] Frans A Oliehoek, Rahul Savani, Jose Gallego-Posada, Elise Van der Pol, Edwin D De Jong, and Roderich Groß. Gangs: Generative adversarial network games. *arXiv preprint arXiv:1712.00679*, 2017.

[OW12] Martijn van Otterlo and Marco Wiering. Reinforcement learning and markov decision processes. In *Reinforcement learning*, pages 3–42. Springer, 2012.

[PAK99] Fabien AP Petitcolas, Ross J Anderson, and Markus G Kuhn. Information hiding-a survey. *Proceedings of the IEEE*, 87(7):1062–1078, 1999.

[Pat14] Shon Patil. *Fundamentals of Digital Watermarking*. PhD thesis, University of Houston, 2014.

[PBR18] Michael J Pennock, Douglas A Bodner, and William B Rouse. Lessons learned from evaluating an enterprise modeling methodology. *IEEE Systems Journal*, 12(2):1219–1229, 2018.

[PFPG09] Luis Pérez-Freire and Fernando Pérez-González. Spread-spectrum watermarking security. *IEEE Transactions on Information Forensics and Security*, 4(1):2–24, 2009.

[PGM$^+$16] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. {DRAMA}: Exploiting {DRAM} addressing for cross-cpu attacks. In *25th {USENIX} security symposium ({USENIX} security 16)*, pages 565–581, 2016.

[PM21] Zhixin Pan and Prabhat Mishra. Automated test generation for hardware trojan detection using reinforcement learning. In *Proceedings of the 26th Asia and South Pacific Design Automation Conference*, pages 408–413, 2021.

[PMJ$^+$16] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings. In *2016 IEEE European symposium on security and privacy (EuroS&P)*, pages 372–387. IEEE, 2016.

[PPA$^+$13] Angshuman Parashar, Michael Pellauer, Michael Adler, Bushra Ahsan, Neal Crago, Daniel Lustig, Vladimir Pavlov, Antonia Zhai, Mohit Gambhir, Aamer

Jaleel, et al. Triggered instructions: a control paradigm for spatially-programmed architectures. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 142–153. ACM, 2013.

[PTLK18] Fabio Pardo, Arash Tavakoli, Vitaly Levdik, and Petar Kormushev. Time limits in reinforcement learning. In *International Conference on Machine Learning*, pages 4045–4054. PMLR, 2018.

[PVZ15] Omkar M Parkhi, Andrea Vedaldi, and Andrew Zisserman. Deep face recognition. 2015.

[PZ98] Christine I Podilchuk and Wenjun Zeng. Image-adaptive watermarking using visual models. *IEEE Journal on selected areas in communications*, 16(4):525–539, 1998.

[QP07] Gang Qu and Miodrag Potkonjak. *Intellectual property protection in VLSI designs: theory and practice*. Springer Science & Business Media, 2007.

[RDGF16] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[RET$^+$17] David Ross, Brian Elmenhurst, Mark Tocci, John Forbes, and Heather Wheelock Ross. Digital fingerprinting track and trace system, February 28 2017. US Patent 9,582,714.

[RGB$^+$16] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. Flip feng shui: Hammering a needle in the software stack. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 1–18, 2016.

[RHF19] Adnan Siraj Rakin, Zhezhi He, and Deliang Fan. Bit-flip attack: Crushing neural network with progressive bit search. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 1211–1220, 2019.

[RHF20] Adnan Siraj Rakin, Zhezhi He, and Deliang Fan. Tbt: Targeted neural network attack with bit trojan. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 13198–13207, 2020.

[RHL$^+$21] Adnan Siraj Rakin, Zhezhi He, Jingtao Li, Fan Yao, Chaitali Chakrabarti, and Deliang Fan. T-bfa: Targeted bit-flip adversarial weight attack. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2021.

[RKM08] Jarrod A Roy, Farinaz Koushanfar, and Igor L Markov. Epic: Ending piracy of integrated circuits. In *Proceedings of the conference on Design, automation and test in Europe*, pages 1069–1074. ACM, 2008.

[RNH⁺09] Benjamin IP Rubinstein, Blaine Nelson, Ling Huang, Anthony D Joseph, Shing-hon Lau, Satish Rao, Nina Taft, and JD Tygar. Stealthy poisoning attacks on pca-based anomaly detectors. *ACM SIGMETRICS Performance Evaluation Review*, 37(2):73–74, 2009.

[Ros13] Peter Rosenmai. Using the median absolute deviation to find outliers. https://eurekastatistics.com/using-the-median-absolute-deviation-to-find-outliers/, 2013.

[RPSK12] Jeyavijayan Rajendran, Youngok Pino, Ozgur Sinanoglu, and Ramesh Karri. Security analysis of logic obfuscation. In *Proceedings of the 49th Annual Design Automation Conference*, pages 83–89. ACM, 2012.

[RRK18] Bita Darvish Rouhani, M. Sadegh Riazi, and Farinaz Koushanfar. Deepsecure: Scalable provably-secure deep learning. In *Proceedings of the 55th Annual Design Automation Conference*, pages 2:1–2:6. ACM, 2018.

[RSC⁺19] M Sadegh Riazi, Mohammad Samragh, Hao Chen, Kim Laine, Kristin Lauter, and Farinaz Koushanfar. Xonn: Xnor-based oblivious deep neural network inference. *USENIX Security*, 2019.

[RSJ⁺18] Bita Darvish Rouhani, Mohammad Samragh, Mojan Javaheripi, Tara Javidi, and Farinaz Koushanfar. Deepfense: Online accelerated defense against adversarial deep learning. In *Proceedings of the International Conference on Computer-Aided Design*, page 134. ACM, 2018.

[RSSK13] Jeyavijayan Rajendran, Michael Sam, Ozgur Sinanoglu, and Ramesh Karri. Security analysis of integrated circuit camouflaging. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 709–720. ACM, 2013.

[RWT⁺18] M Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M Songhori, Thomas Schneider, and Farinaz Koushanfar. Chameleon: A hybrid secure computation framework for machine learning applications. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*. ACM, 2018.

[RZZ⁺15] Jeyavijayan Rajendran, Huan Zhang, Chi Zhang, Garrett S Rose, Youngok Pino, Ozgur Sinanoglu, and Ramesh Karri. Fault analysis-based logic encryption. *IEEE Transactions on computers*, 64(2):410–424, 2015.

[Sam14] Seyyed Mohammad Saleh Samimi. Testability measurement tool. https://sourceforge.net/projects/testabilitymeasurementtool/, 2014.

[SB18] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[SC13] Prabhishek Singh and RS Chadha. A survey of digital watermarking techniques, applications and attacks. *International Journal of Engineering and Innovative Technology (IJEIT)*, 2(9):165–175, 2013.

[Sch15] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[Sch17] Ilan Schnell. pycosat 0.6.3. https://pypi.org/project/pycosat/, Nov 9, 2017.

[SCL$^+$16] Yao Shi, Myungjoon Choi, Ziyun Li, Gyouho Kim, Zhiyoong Foo, Hun-Seok Kim, David Wentzloff, and David Blaauw. 26.7 a 10mm3 syringe-implantable near-field radio system on glass substrate. In *2016 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 448–449. IEEE, 2016.

[SCN$^+$15] Sayandeep Saha, Rajat Subhra Chakraborty, Srinivasa Shashank Nuthakki, Debdeep Mukhopadhyay, et al. Improved test pattern generation for hardware trojan detection using genetic algorithm and boolean satisfiability. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 577–596. Springer, 2015.

[SD15] Mark Seaborn and Thomas Dullien. Exploiting the dram rowhammer bug to gain kernel privileges. *Black Hat*, 15:71, 2015.

[SEJ$^+$20] Andrew W Senior, Richard Evans, John Jumper, James Kirkpatrick, Laurent Sifre, Tim Green, Chongli Qin, Augustin Žídek, Alexander WR Nelson, Alex Bridgland, et al. Improved protein structure prediction using potentials from deep learning. *Nature*, 577(7792):706–710, 2020.

[SESL18] Yi Shi, Tugba Erpek, Yalin E Sagduyu, and Jason H Li. Spectrum data poisoning with adversarial deep learning. In *MILCOM 2018-2018 IEEE Military Communications Conference (MILCOM)*, pages 407–412. IEEE, 2018.

[Shi99] HISASHI Shimodaira. A diversity-control-oriented genetic algorithm (dcga): development and initial experimental results. *Trans. Inf. Process. Soc. Jpn.(Japan)*, 40(6):2708–2716, 1999.

[SHK$^+$14] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[SHN$^+$18] Ali Shafahi, W Ronny Huang, Mahyar Najibi, Octavian Suciu, Christoph Studer, Tudor Dumitras, and Tom Goldstein. Poison frogs! targeted clean-label poisoning attacks on neural networks. In *Advances in Neural Information Processing Systems*, pages 6103–6113, 2018.

[SHS$^+$15] Ebrahim M Songhori, Siam U Hussain, Ahmad-Reza Sadeghi, Thomas Schneider, and Farinaz Koushanfar. Tinygarble: Highly compressed and scalable sequential garbled circuits. In *2015 IEEE Symposium on Security and Privacy*, 2015.

[SHS$^+$17] Bicky Shakya, Tony He, Hassan Salmani, Domenic Forte, Swarup Bhunia, and Mark Tehranipoor. Benchmarking of hardware trojans and maliciously affected circuits. *Journal of Hardware and Systems Security*, 1(1):85–102, 2017.

[Sim04] Donald Simon. Conjunctive normal form, 2004.

[SJK19] Mohammad Samragh, Mojan Javaheripi, and Farinaz Koushanfar. Codex: Bit-flexible encoding for streaming-based fpga acceleration of dnns. *arXiv preprint arXiv:1901.05582*, 2019.

[SK16] Tim Salimans and Durk P Kingma. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. In *Advances in neural information processing systems*, pages 901–909, 2016.

[SKL06] Emanuele Stomeo, Tatiana Kalganova, and Cyrille Lambert. A novel genetic algorithm for evolvable hardware. In *2006 IEEE International Conference on Evolutionary Computation*, pages 134–141. IEEE, 2006.

[SL19] Rodrigo Vieira Steiner and Emil Lupu. Towards more practical software-based attestation. *Computer networks*, 149:43–55, 2019.

[SLM$^+$17] Kaveh Shamsi, Meng Li, Travis Meade, Zheng Zhao, David Z Pan, and Yier Jin. Appsat: Approximately deobfuscating integrated circuits. In *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 95–100. IEEE, 2017.

[SMC$^+$17] Felipe Petroski Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth O Stanley, and Jeff Clune. Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. *arXiv preprint arXiv:1712.06567*, 2017.

[SPJ19] Kaveh Shamsi, David Z Pan, and Yier Jin. On the impossibility of approximation-resilient circuit locking. In *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 161–170. IEEE, 2019.

[SPS$^+$18] Hardik Sharma, Jongse Park, Naveen Suda, Liangzhen Lai, Benson Chau, Vikas Chandra, and Hadi Esmaeilzadeh. Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 764–775. IEEE, 2018.

[SPVDK04] Arvind Seshadri, Adrian Perrig, Leendert Van Doorn, and Pradeep Khosla. Swatt: Software-based attestation for embedded devices. In *IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004*, pages 272–282. IEEE, 2004.

[SQLC17] Linghao Song, Xuehai Qian, Hai Li, and Yiran Chen. Pipelayer: A pipelined reram-based accelerator for deep learning. In *2017 IEEE international symposium on high performance computer architecture (HPCA)*, pages 541–552. IEEE, 2017.

[SRG$^+$16] Hoo-Chang Shin, Holger R Roth, Mingchen Gao, Le Lu, Ziyue Xu, Isabella Nogues, Jianhua Yao, Daniel Mollura, and Ronald M Summers. Deep convolutional neural networks for computer-aided detection: Cnn architectures, dataset characteristics and transfer learning. *IEEE transactions on medical imaging*, 35(5):1285–1298, 2016.

[SRM15] Pramod Subramanyan, Sayak Ray, and Sharad Malik. Evaluating the security of logic encryption algorithms. In *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 137–143. IEEE, 2015.

[SS01] Bernhard Scholkopf and Alexander J Smola. *Learning with kernels: support vector machines, regularization, optimization, and beyond*. MIT press, 2001.

[SS20] Deepak Sirone and Pramod Subramanyan. Functional analysis attacks on logic locking. *IEEE Transactions on Information Forensics and Security*, 15:2514–2527, 2020.

[SSTF19] Bicky Shakya, Haoting Shen, Mark Tehranipoor, and Domenic Forte. Covert gates: Protecting integrated circuits with undetectable camouflaging. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 86–118, 2019.

[Sta] Princeton Stanford. Imagenet. http://image-net.org/. Accessed: 2021-03-13.

[STP11] Hassan Salmani, Mohammad Tehranipoor, and Jim Plusquellic. A novel technique for improving hardware trojan detection and reducing trojan activation time. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 20(1):112–125, 2011.

[SVI$^+$16] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016.

[SW15] Takahiro Shinozaki and Shinji Watanabe. Structure discovery of deep neural network based on evolutionary algorithms. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4979–4983. IEEE, 2015.

[Syn17] Synopsys. Designware pipelined aes-gcm/ctr core. https://www.synopsys.com/dw/ipdir.php?ds=security-aes-gcm-ctr., 2017.

[SZ14a] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[SZ14b] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[TB18] Florian Tramer and Dan Boneh. Slalom: Fast, verifiable and private execution of neural networks in trusted hardware. In *International Conference on Learning Representations*, 2018.

[TDP19] Ian Tenney, Dipanjan Das, and Ellie Pavlick. Bert rediscovers the classical nlp pipeline. *arXiv preprint arXiv:1905.05950*, 2019.

[TGBR18] Andrei Tatar, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Defeating software mitigations against rowhammer: a surgical precision hammer. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 47–66. Springer, 2018.

[TJ09] Randy Torrance and Dick James. The state-of-the-art in ic reverse engineering. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 363–381. Springer, 2009.

[TK10] Mohammad Tehranipoor and Farinaz Koushanfar. A survey of hardware trojan taxonomy and detection. *IEEE design & test of computers*, 27(1):10–25, 2010.

[TKL$^+$20] Benjamin Tan, Ramesh Karri, Nimisha Limaye, Abhrajit Sengupta, Ozgur Sinanoglu, Md Moshiur Rahman, Swarup Bhunia, Danielle Duvalsaint, Amin Rezaei, Yuanqi Shen, et al. Benchmarking at the frontier of hardware security: Lessons from logic locking. *arXiv preprint arXiv:2006.06806*, 2020.

[TMKH96] Kit-Sang Tang, Kim-Fung Man, Sam Kwong, and Qun He. Genetic algorithms and their applications. *IEEE signal processing magazine*, 13(6):22–37, 1996.

[TMLL06] Richard Ta-Min, Lionel Litty, and David Lie. Splitting interfaces: Making trust between applications and operating systems configurable. In *Proceedings of the 7th symposium on Operating systems design and implementation*, 2006.

[TSG$^+$16] Nima Tajbakhsh, Jae Y Shin, Suryakanth R Gurudu, R Todd Hurst, Christopher B Kendall, Michael B Gotway, and Jianming Liang. Convolutional neural networks for medical image analysis: Full training or fine tuning? *IEEE transactions on medical imaging*, 35(5):1299–1312, 2016.

[TSK$^+$18] Chuanqi Tan, Fuchun Sun, Tao Kong, Wenchang Zhang, Chao Yang, and Chunfang Liu. A survey on deep transfer learning. In *International conference on artificial neural networks*, pages 270–279. Springer, 2018.

[TTAH12] Maryam Tanha, Seyed Dawood Sajjadi Torshizi, Mohd Taufik Abdullah, and Fazirulhisyam Hashim. An overview of attacks against digital watermarking and their respective countermeasures. In *Proceedings Title: 2012 International Conference on Cyber Security, Cyber Warfare and Digital Forensic (CyberSec)*, pages 265–270. IEEE, 2012.

[TW11] Mohammad Tehranipoor and Cliff Wang. *Introduction to hardware security and trust*. Springer Science & Business Media, 2011.

[TWL02] Wade Trappe, Min Wu, and KJ Ray Liu. Collusion-resistant fingerprinting for multimedia. In *Acoustics, Speech, and Signal Processing (ICASSP), 2002 IEEE International Conference on*, volume 4, pages IV–3309. IEEE, 2002.

[TWWL03] Wade Trappe, Min Wu, Z Jane Wang, and KJ Ray Liu. Anti-collusion fingerprinting for multimedia. *IEEE Transactions on Signal Processing*, 51(4):1069–1087, 2003.

[UB05] Ilkay Ulusoy and Christopher M Bishop. Generative versus discriminative methods for object recognition. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, volume 2, pages 258–265. IEEE, 2005.

[Uch17] Yusuke Uchida. Embedding watermarks into deep neural networks. https://github.com/yu4u/dnn-watermark, 2017.

[Uni] Stanford University. The street view house numbers (svhn) dataset. http://ufldl.stanford.edu/housenumbers/. Accessed: 2021-03-13.

[UNSS17] Yusuke Uchida, Yuki Nagai, Shigeyuki Sakazawa, and Shin'ichi Satoh. Embedding watermarks into deep neural networks. In *Proceedings of the 2017 ACM on international conference on multimedia retrieval*, pages 269–277, 2017.

[Urs02] Rasmus K Ursem. Diversity-guided evolutionary algorithms. In *International Conference on Parallel Problem Solving from Nature*, pages 462–471. Springer, 2002.

[VDVFL$^+$16] Victor Van Der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic rowhammer attacks on mobile platforms. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 1675–1689, 2016.

[VM19] Matthew Veres and Medhat Moussa. Deep learning for intelligent transportation systems: A survey of emerging trends. *IEEE Transactions on Intelligent transportation systems*, 21(8):3152–3168, 2019.

[VVB18] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. Graviton: Trusted execution environments on gpus. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 681–696, 2018.

[Wan03] Sun-Chong Wang. Artificial neural network. In *Interdisciplinary computing in java programming*, pages 81–100. Springer, 2003.

[WBA17] Ofir Weisse, Valeria Bertacco, and Todd Austin. Regaining lost cycles with hotcalls: A fast interface for sgx secure enclaves. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 81–93. ACM, 2017.

[WGY$^+$16] Chao Wang, Lei Gong, Qi Yu, Xi Li, Yuan Xie, and Xuehai Zhou. Dlau: A scalable deep learning accelerator unit on fpga. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(3):513–517, 2016.

[WK21] Tianhao Wang and Florian Kerschbaum. Riga: Covert and robust white-box watermarking of deep neural networks. In *Proceedings of the Web Conference 2021*, pages 993–1004, 2021.

[WLW$^+$16] Jiaxiang Wu, Cong Leng, Yuhang Wang, Qinghao Hu, and Jian Cheng. Quantized convolutional neural networks for mobile devices. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4820–4828, 2016.

[WM03] D Randall Wilson and Tony R Martinez. The general inefficiency of batch training for gradient descent learning. *Neural networks*, 16(10):1429–1451, 2003.

[WPB$^+$21] Emily Wenger, Josephine Passananti, Arjun Nitin Bhagoji, Yuanshun Yao, Haitao Zheng, and Ben Y Zhao. Backdoor attacks against deep learning systems in the physical world. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6206–6215, 2021.

[WSS13] Adam Waksman, Matthew Suozzo, and Simha Sethumadhavan. Fanci: identification of stealthy malicious logic using boolean functional analysis. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 697–708, 2013.

[WSTP08] Xiaoxiao Wang, Hassan Salmani, Mohammad Tehranipoor, and Jim Plusquellic. Hardware trojan detection and isolation using current integration and localized current analysis. In *2008 IEEE international symposium on defect and fault tolerance of VLSI systems*, pages 87–95. IEEE, 2008.

[WTWL04] Min Wu, Wade Trappe, Z Jane Wang, and KJ Ray Liu. Collusion-resistant multimedia fingerprinting: a unified framework. In *Security, Steganography,* *and Watermarking of Multimedia Contents VI*, volume 5306, pages 748–760. International Society for Optics and Photonics, 2004.

[WVO12] Marco A Wiering and Martijn Van Otterlo. Reinforcement learning. *Adaptation, learning, and optimization*, 12(3):729, 2012.

[WWZ$^+$03] Z Jane Wang, Min Wu, Hong Zhao, KJ Ray Liu, and Wade Trappe. Resistance of orthogonal gaussian fingerprints to collusion attacks. In *2003 International Conference on Multimedia and Expo. ICME'03. Proceedings (Cat. No. 03TH8698)*, volume 1, pages I–617. IEEE, 2003.

[WWZ$^+$05] Z Jane Wang, Min Wu, H Vicky Zhao, Wade Trappe, and KJ Ray Liu. Anti-collusion forensics of multimedia fingerprinting using orthogonal modulation. *IEEE transactions on image processing*, 14(6):804–821, 2005.

[WX18] Rey Wiyatno and Anqi Xu. Maximal jacobian-based saliency map attack. *arXiv preprint arXiv:1808.07945*, 2018.

[WYS$^+$19] Bolun Wang, Yuanshun Yao, Shawn Shan, Huiying Li, Bimal Viswanath, Haitao Zheng, and Ben Y Zhao. Neural cleanse: Identifying and mitigating backdoor attacks in neural networks. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 707–723. IEEE, 2019.

[XFJ$^+$16] Kan Xiao, Domenic Forte, Yier Jin, Ramesh Karri, Swarup Bhunia, and Mohammad Tehranipoor. Hardware trojans: Lessons learned after one decade of research. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 22(1):1–23, 2016.

[XS18] Yang Xie and Ankur Srivastava. Anti-sat: Mitigating sat attack on logic locking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(2):199–207, 2018.

[XSLH16] Bin Cedric Xing, Mark Shanahan, and Rebekah Leslie-Hurd. Intel&reg; software guard extensions (intel&reg; sgx) software support for dynamic memory allocation inside an enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy*. ACM, 2016.

[XT13] Kan Xiao and Mohammed Tehranipoor. Bisa: Built-in self-authentication for preventing hardware trojan insertion. In *2013 IEEE international symposium on hardware-oriented security and trust (HOST)*, pages 45–50. IEEE, 2013.

[XWL21] Mingfu Xue, Jian Wang, and Weiqiang Liu. Dnn intellectual property protection: Taxonomy, attacks and evaluations. In *Proceedings of the 2021 on Great Lakes Symposium on VLSI*, pages 455–460, 2021.

[YHPC18] Tom Young, Devamanyu Hazarika, Soujanya Poria, and Erik Cambria. Recent trends in deep learning based natural language processing. *ieee Computational intelligenCe magazine*, 13(3):55–75, 2018.

[YHZL19] Xiaoyong Yuan, Pan He, Qile Zhu, and Xiaolin Li. Adversarial examples: Attacks and defenses for deep learning. *IEEE transactions on neural networks and learning systems*, 30(9):2805–2824, 2019.

[YLCZ10] Yongsheng Yu, Hongwei Lu, Xiaosu Chen, and Zhiguang Zhang. Group-oriented anti-collusion fingerprint based on bibd code. In *e-Business and Information System Security (EBISS), 2010 2nd International Conference on*, pages 1–5. IEEE, 2010.

[YMRS16] Muhammad Yasin, Bodhisatwa Mazumdar, Jeyavijayan JV Rajendran, and Ozgur Sinanoglu. Sarlock: Sat attack resistant logic locking. In *2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 236–241. IEEE, 2016.

[YMSR16] Muhammad Yasin, Bodhisatwa Mazumdar, Ozgur Sinanoglu, and Jeyavijayan Rajendran. Camoperturb: Secure ic camouflaging for minterm protection. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2016.

[YMSR17a] Muhammad Yasin, Bodhisatwa Mazumdar, Ozgur Sinanoglu, and Jeyavijayan Rajendran. Removal attacks on logic locking and camouflaging techniques. *IEEE Transactions on Emerging Topics in Computing*, 8(2):517–532, 2017.

[YMSR17b] Muhammad Yasin, Bodhisatwa Mazumdar, Ozgur Sinanoglu, and Jeyavijayan Rajendran. Security analysis of anti-sat. In *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 342–347. IEEE, 2017.

[YRF20] Fan Yao, Adnan Siraj Rakin, and Deliang Fan. Deephammer: Depleting the intelligence of deep neural networks through targeted chain of bit flips. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 1463–1480, 2020.

[YRSK15] Muhammad Yasin, Jeyavijayan JV Rajendran, Ozgur Sinanoglu, and Ramesh Karri. On improving the security of logic locking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(9):1411–1424, 2015.

[YRSK16] Muhammad Yasin, Jeyavijayan JV Rajendran, Ozgur Sinanoglu, and Ramesh Karri. On improving the security of logic locking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(9):1411–1424, 2016.

[YS17] Muhammad Yasin and Ozgur Sinanoglu. Evolution of logic locking. In *2017 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, pages 1–6. IEEE, 2017.

[YSN$^+$17] Muhammad Yasin, Abhrajit Sengupta, Mohammed Thari Nabeel, Mohammed Ashraf, Jeyavijayan Rajendran, and Ozgur Sinanoglu. Provably-secure logic locking: From theory to practice. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1601–1618, 2017.

[Yua15] Zeying Yuan. *Sequential Equivalence Checking of Circuits with Different State Encodings by Pruning Simulation-based Multi-Node Invariants*. PhD thesis, Virginia Tech, 2015.

[YYC$^+$20] Yipei Yang, Jing Ye, Yuan Cao, Jiliang Zhang, Xiaowei Li, Huawei Li, and Yu Hu. Survey: Hardware trojan detection for netlist. In *2020 IEEE 29th Asian Test Symposium (ATS)*, pages 1–6. IEEE, 2020.

[YYS$^+$13] Dong Yu, Kaisheng Yao, Hang Su, Gang Li, and Frank Seide. Kl-divergence regularized deep neural network adaptation for improved large vocabulary speech recognition. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 7893–7897. IEEE, 2013.

[ZGJ$^+$18] Jialong Zhang, Zhongshu Gu, Jiyong Jang, Hui Wu, Marc Ph Stoecklin, Heqing Huang, and Ian Molloy. Protecting intellectual property of deep neural networks with watermarking. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 159–172. ACM, 2018.

[ZK16] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016.

[ZT11] Xuehui Zhang and Mohammad Tehranipoor. Ron: An on-chip ring oscillator network for hardware trojan detection. In *2011 Design, Automation & Test in Europe*, pages 1–6. IEEE, 2011.

[ZWLZ19] Zhisheng Zhong, Fangyin Wei, Zhouchen Lin, and Chao Zhang. Ada-tucker: Compressing deep neural networks via adaptive dimension adjustment tucker decomposition. *Neural Networks*, 110:104–115, 2019.

[ZWN$^+$16] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.

[ZYL$^+$17] Qingchen Zhang, Laurence T Yang, Xingang Liu, Zhikui Chen, and Peng Li. A tucker deep computation model for mobile multimedia feature learning. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, 13(3s):1–18, 2017.

[ZYW$^+$15] Jie Zhang, Feng Yuan, Linxiao Wei, Yannan Liu, and Qiang Xu. Veritrust: Verification for hardware trust. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(7):1148–1161, 2015.

[ZZT+15] Bin Zhou, Wei Zhang, Srikanthan Thambipillai, Jason Teo Kian Jin, Vivek Chaturvedi, and Tao Luo. Cost-efficient acceleration of hardware trojan detection through fan-out cone analysis and weighted random pattern technique. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(5):792–805, 2015.