DEEP LEARNING UNDER THE MICROSCOPE: HARDWARE-ORIENTED ATTACKS AND ROBUST DEFENSES

XIAOBEI YAN

College of Computing and Data Science

A thesis submitted to the Nanyang Technological University
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

2026

Statement of Originality

I hereby certify that the work embodied in this thesis is the result of original research, is free of plagiarized materials, and has not been submitted for a higher degree to any other University or Institution.

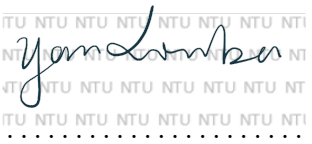
Disclosure on AI Use:

I hereby declare that ChatGPT, version 5.1, was used to improve the clarity and readability of this thesis. I also declare that I have abided by NTU's Policy on the Use of GenAI in Research, and I have reviewed and edited the content as needed after using this tool, and I take full responsibility for the content of this thesis.

21-12-2025

.....

Date



.....

XIAOBEI YAN

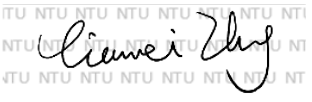
Supervisor Declaration Statement

I have reviewed the content and presentation style of this thesis and declare it is free of plagiarism and of sufficient grammatical clarity to be examined. To the best of my knowledge, the research and writing are those of the candidate except as acknowledged in the Author Attribution Statement. I confirm that the investigations were conducted in accord with the ethics policies and integrity standards of Nanyang Technological University and that the research data are presented honestly and without prejudice.

21-12-2025

.....

Date



.....

Assoc Prof Tianwei Zhang

Authorship Attribution Statement

This thesis contains materials from 4 papers accepted at conferences in which I am listed as an author.

Chapter 3 is published as Xiaobei Yan, Xiaoxuan Lou, Guowen Xu, Han Qiu, Shangwei Guo, Chip Hong Chang, Tianwei Zhang. Mercury: An automated remote side-channel attack to NVIDIA deep learning accelerator. In International Conference on Field Programmable Technology (FPT), 2023.

The contributions of the co-authors are as follows:

- I was the lead author. I wrote the manuscript draft and conducted all the experiments.
- Prof. Tianwei Zhang guided the initial research direction and revised the manuscript draft.
- I co-designed the methodology with Prof. Tianwei Zhang, and Dr. Xiaoxuan Lou.
- Prof. Guowen Xu, Prof. Han Qiu, Prof. Shangwei Guo, Prof. Chip Hong Chang discussed and supported the research, and revised the draft.

Chapter 5 is published as Xiaobei Yan, Chip Hong Chang, Shangwei Guo and Tianwei Zhang. AdvProtego: A Unified Framework to SafeGuard Deep Learning Models Against Side Channel Leakage. In IEEE International Symposium on Hardware Oriented Security and Trust (HOST), 2026.

The contributions of the co-authors are as follows:

- I was the lead author. I wrote the manuscript draft and conducted all the experiments.
- Prof. Tianwei Zhang guided the initial research direction and revised the manuscript draft.
- Prof. Chip Hong Chang and Prof. Shangwei Guo discussed and supported the research, and revised the draft.

Chapter 6 is published as Xiaobei Yan, Han Qiu and Tianwei Zhang. ObfusBFA: A Holistic Approach to Safeguarding DNNs from Versatile Bit-Flip Attacks. In IEEE International Symposium on Hardware Oriented Security and Trust (HOST), 2026.

The contributions of the co-authors are as follows:

- I was the lead author. I wrote the manuscript draft and conducted all the experiments.
- Prof. Tianwei Zhang guided the initial research direction and revised the manuscript draft.

- Prof. Han Qiu discussed and supported the research, and revised the draft.

21-12-2025

Date

XIAOBEI YAN

Acknowledgements

This thesis represents the culmination of my doctoral journey at Nanyang Technological University, Singapore—a place that has served not just as an academic institution, but as a supportive home for my growth and aspirations.

My deepest gratitude goes to my supervisor, Prof. Tianwei Zhang. His mentorship has been the defining factor of my Ph.D. Prof. Zhang's insistence on academic rigor, combined with his empathetic guidance, gave me the confidence to navigate the uncertainties of research. He has taught me not only how to conduct high-quality research but also how to approach challenges with resilience—a lesson I will carry forward throughout my career.

I am also sincerely grateful to Prof. Chip Hong Chang and Dr. Yiming Li. Their mentorship provided clarity during challenging moments, and their constructive feedback was essential in refining my work. I truly appreciate their generosity in sharing their expertise and helping me navigate the technical complexities of my research.

I have been fortunate to share this journey with a wonderful group of colleagues and friends. The collaborative environment and daily interactions in the lab turned solitary research into a shared adventure. I would like to thank Dr. Anran Li, Boheng Li, Dr. Guowen Xu, Dr. Hanxiao Chen, Dr. Jian Zhang, Dr. Jiaxing Xu, Dr. Jie Zhang, Dr. Kangjie Chen, Dr. Meng Hao, Meng Zhang, Dr. Mengmeng Ge, Mengcheng Lan, Dr. Ming Hu, Dr. Qinghao Hu, Dr. Tianlin Li, Haoran Ou, Runyi Hu, Sihan Xia, Shiqian Zhao, Wenjun Long, Dr. Xiaoxuan Lou, Xinpeng Yang, Yutong Wu, Dr. Yi Xie, Yue Cao, and Dr. Zhaoxuan Wang. Thank you for the inspiring discussions and the camaraderie that made the hard work enjoyable.

Finally, I dedicate this work to my parents, my beloved, and my family. Your unconditional love and silent sacrifices have been my rock. Knowing I had your

unwavering support gave me the strength to persevere through every obstacle and reach this milestone.

To everyone who contributed to my personal and academic growth, thank you for being a part of this remarkable experience.

Xiaobei Yan, Dec 2025

Abstract

Deep learning models are increasingly deployed across heterogeneous platforms, ranging from specialized AI hardware to large-scale cloud servers. While this enables impressive accuracy and efficiency, it also exposes new attack surfaces at the low-level system and hardware layers. Typical examples include side-channel leakages and fault-induced bit flips, which could undermine the target model's confidentiality, availability, and integrity. This thesis presents a series of hardware-oriented studies towards Deep Learning (DL) security, covering both new attack discovery and defense designs for side-channel and bit-flip threats.

From the attack perspective, we first propose `Mercury`, an automated remote power side-channel attack that recovers the architecture of a victim Deep Neural Network (DNN) from a single execution trace. `Mercury` leverages a time-to-digital converter (TDC)–based timing/power monitor and formulates architecture recovery as a sequence-to-sequence learning task. By using RNN-CTC and Transformer-based decoders with attention, it reconstructs layer sequences with very low error rates and localizes the most informative leakage regions, thereby enabling downstream exploits such as adversarial example generation and membership inference without prior knowledge of the model.

We further introduce `BitHydra`, the first bit-flip inference-cost attack against Large Language Models (LLMs). The attack objective is to induce the computational overhead of LLM generation. Instead of relying on adversarial prompts, `BitHydra` directly perturbs the model's parameters, which can achieve the goal consistently across all users and queries. `BitHydra` optimizes a loss that suppresses the probability of emitting the end-of-sequence (`<EOS>`) token and performs a gradient-guided search over the `<EOS>` embedding row to identify a small set of critical bits. Then it triggers hardware-level bit flips (e.g., via Rowhammer-like faults). Experiments on diverse families of LLMs show that flipping only a few (e.g., 3–30) bits can force models to generate extremely long, syntactically coherent outputs, with a high

fraction of prompts reaching maximum generation length and strong robustness against existing mitigation solutions.

From the defense perspective, we design `AdvProtego`, a unified framework against various forms of side-channel attacks. Its key idea is to inject adversarially crafted execution noise to disrupt machine-learning-assisted side-channel analysis. We formalize two complementary defense goals—reducing model similarity and reducing model utility from the attacker's perspective—as adversarial perturbation problems in the space of low-level execution traces. `AdvProtego` is instantiated through a frontend–middleware–backend pipeline on multiple representative platforms (including power, cache, and memory side-channels), substantially degrading model-extraction accuracy while incurring modest performance and accuracy overheads, and remaining effective even when the attacker applies adversarial training.

Finally we propose `ObfusBFA`, a holistic defense framework against versatile bit-flip attacks on DNNs, covering both weight-level and implementation-level vulnerabilities. `ObfusBFA` incorporates a vulnerability searcher that pinpoints bit-flip–sensitive components across model parameters and low-level code, an obfuscation pattern generator that introduces structured redundancy and dummy operations, and an enforcer that embeds these patterns into mainstream deep learning infrastructures and compiler toolchains. We instantiate `ObfusBFA` across C++ inference runtimes, PyTorch-based systems, and TVM-based compilers, demonstrating that it can substantially mitigate targeted, untargeted, and adaptive bit-flip attacks with acceptable computational and storage overheads.

Overall, this thesis advances the understanding of how hardware-level behaviors and implementations interact with modern deep learning systems, and provides both practical attack methodologies and deployable defense mechanisms. By jointly studying side-channel leakage, fault-induced bit flips, and obfuscation-based defenses, it contributes towards building more robust and secure deep learning applications in realistic environments.

Contents

List of Publications

- Xiaobei Yan, Xiaoxuan Lou, Guowen Xu, Han Qiu, Shangwei Guo, Chip Hong Chang, Tianwei Zhang. **Mercury: An automated remote side-channel attack to NVIDIA deep learning accelerator**. In *International Conference on Field Programmable Technology (FPT), 2023*.

- Xiaobei Yan, Han Qiu, Tianwei Zhang. **UniGuard: A Unified Hardware-oriented Threat Detector for FPGA-based AI Accelerators**. In *34th International Conference on Field-Programmable Logic and Applications (FPL), 2024* (Stamatis Vassiliadis Best Paper Award Nominee).

- Xiaobei Yan, Han Qiu and Tianwei Zhang. **ObfusBFA: A Holistic Approach to Safeguarding DNNs from Versatile Bit-Flip Attacks**. In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST), 2026*.

- Xiaobei Yan, Chip Hong Chang, Shangwei Guo and Tianwei Zhang. **AdvProtego: A Unified Framework to SafeGuard Deep Learning Models Against Side Channel Leakage**. In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST), 2026*.

- Xiaobei Yan, Yiming Li, and Tianwei Zhang. **BitHydra: Towards Bit-flip Inference Cost Attack against Large Language Model**. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2026*.

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Deep learning has become a fundamental building block of modern artificial intelligence (AI) systems, enabling breakthroughs in areas such as autonomous driving [1], medical analysis [2], and natural language processing [3]. Deep neural networks (DNNs) achieve superior accuracy by stacking multiple layers of nonlinear transformations and training on large-scale datasets, often containing sensitive or proprietary information. To meet the growing demand for low-latency and high-throughput inference, these models are increasingly deployed in performance-critical environments, such as data centers, cloud platforms, and edge devices.

To sustain this growth, the computing stack for deep learning has evolved into a heterogeneous ecosystem. On the hardware level, specialized accelerators such as GPUs, TPUs, and custom ASICs, as well as reconfigurable devices like FPGAs, are widely used to speed up matrix operations and convolutions. On the software level, mature frameworks (e.g., PyTorch and TensorFlow) and compiler stacks (e.g., TVM [4]) abstract the underlying hardware complexity and provide flexible interfaces for model development, optimization, and deployment. Optimized inference engines and serving frameworks further exploit batching, quantization, and caching techniques to efficiently handle massive user traffic.

However, this rich hardware–software ecosystem also introduces a complex and often under-explored security landscape. Traditional research on machine learning security has largely focused on algorithmic vulnerabilities, such as adversarial examples [5], model extraction [6], and backdoor attacks [7]. In contrast, hardware- and system-level vulnerabilities remain comparatively less understood in the context of modern deep learning deployments. For instance, side-channel attacks [8] exploit the physical or microarchitectural behavior of the underlying platform (e.g., power [8], electromagnetic emanations [9], or cache [10]) to infer sensitive information, often without modifying the software-visible interface; fault-induced bit flip attacks [11, 12] intentionally induce low-level faults in memory or computation units, by exploiting voltage glitches [13], laser beaming [14], or DRAM vulnerabilities [15, 16]. When applied to deep learning applications, such attacks can alter model behavior, degrade accuracy, or inflate inference cost. More practically, they can be launched remotely in multi-tenant or cloud environments.

These emerging threats imply that ensuring the security of deep learning systems requires not only robust algorithms and training procedures, but also a thorough understanding of how models interact with the underlying hardware and runtime stack. In particular, it is crucial to study how hardware-level leakages and faults can be exploited to attack DNNs, and how to design practical, deployable defenses that maintain acceptable accuracy and performance in real-world deployments.

1.2 Motivation

1.2.1 Limitations of Existing Works

While the security community has made significant progress in understanding and defending against high-level machine learning threats, several important gaps remain at the interface between hardware behavior and deep learning systems.

First, *existing attack studies on deep learning systems are often constrained in scope or lack realism.* (1) Many prior works rely on strong assumptions, which may not be applicable in reality. For example, several side-channel attacks assume full physical access to the victim device [17–21], which limits their applicability to real-world deployments. (2) Most of the studies evaluate attacks only on overly

simplified implementations. Numerous side-channel attacks target Binary Neural Networks (BNNs) [19, 22] or homemade accelerator prototypes [17–20, 22, 23], and many bit-flip attacks have focused solely on relatively small models such as CNN-based image classifiers [16, 24–28]. (3) Existing works also adopt simplified or restricted attack goals. For instance, most side-channel attacks aim to recover only individual layers rather than full end-to-end model architectures [21–23, 29], and bit-flip attacks typically consider only prediction corruption while neglecting other impactful consequences, such as efficiency degradation. These limitations substantially constrain the practical impact of existing attacks and highlight the need for more automated, efficient, and comprehensive approaches.

Second, *existing defense designs have largely followed ad hoc heuristics, which can incur the following problems.* (1) These solutions suffer from large deployment cost, or fail to cover different attack categories. For side-channel attacks, common countermeasures such as constant execution [30–33], side-channel obfuscation [34–37], and domain isolation [38–43] can be effective against conventional attacks [29], but they often incur substantial performance or resource overhead and do not scale well to modern ML-assisted side-channel analysis, where attackers exploit higher-order statistical features of noisy traces to recover model parameters or architectures with high accuracy and automation [17–20]. For bit-flip attacks, DNN-specific defenses, including architecture redesign [44], weight reconstruction [45], quantization [46, 47], and channel- [48, 49] or bit-level obfuscation [50], as well as runtime integrity checking via checksums [51, 52], hashes [53, 54], or checker networks [55], can introduce non-trivial overhead and may degrade model accuracy; moreover, they are typically tailored to either model-level or executable-level BFAs and do not generalize to attacks on deep learning libraries or mixed-precision deployments [56]. (2) These solutions lack general applicability to different computing platforms. A few preliminary efforts [57, 58] have explored adding adversarial-like perturbations to side-channel signals for cryptographic or fingerprinting workloads, yet they do not provide a general, practical, optimization-based formulation for protecting deep learning models, nor do they address how to synthesize precise, deployable execution noise on heterogeneous platforms. Hardware-level mitigations such as Rowhammer-aware timing controls [59], performance counter monitoring [60, 61], error-correcting codes [62], and memory isolation [63, 64] often require specialized support or deployments that are difficult to retrofit into existing systems and may still leave residual attack windows.

These gaps motivate a systematic study of *hardware-oriented* attacks and defenses for deep learning systems. Concretely, there is a pressing need to:

- Demonstrate realistic, automated and efficient attacks that can expose new vulnerabilities of emerging platforms and models, and achieve new objectives.
- Develop principled, holistic defense frameworks that provide comprehensive protection over deep learning systems while minimizing performance and utility degradation.

Addressing these challenges is crucial for understanding the real security posture of modern deep learning deployments. By bridging the gap between low-level hardware behavior and high-level model functionality, this thesis aims to both expose underexplored attack vectors and design practical defenses that make deep learning systems more robust in realistic environments.

1.2.2 Main Work

This thesis conducts the security investigation of deep learning systems, delving into the interactions between models and their underlying hardware and runtime compute stacks. Its organization is illustrated in Figure 1.1. We focus on side-channel leakage and bit-flip faults, considering both attacks and defenses. The four core research works in this thesis are summarized below.

1. Remote side-channel attack for model extraction. The first part of this thesis presents `Mercury`, an automated remote side-channel attack that recovers the architecture of a deployed deep neural network from noisy execution traces. `Mercury` leverages a timing/power monitoring substrate implemented in reconfigurable logic and formulates architecture recovery as a sequence-to-sequence learning problem. By training recurrent and Transformer-based decoders on labeled traces, `Mercury` can infer layer sequences and key architectural hyperparameters with high accuracy, without requiring invasive measurements or detailed prior knowledge of the victim implementation. This thread of work demonstrates, for the first time, that full end-to-end model architectures can be extracted from realistic side-channel measurements in a largely automated fashion, significantly elevating the practical threat of model-stealing attacks in shared hardware environments.

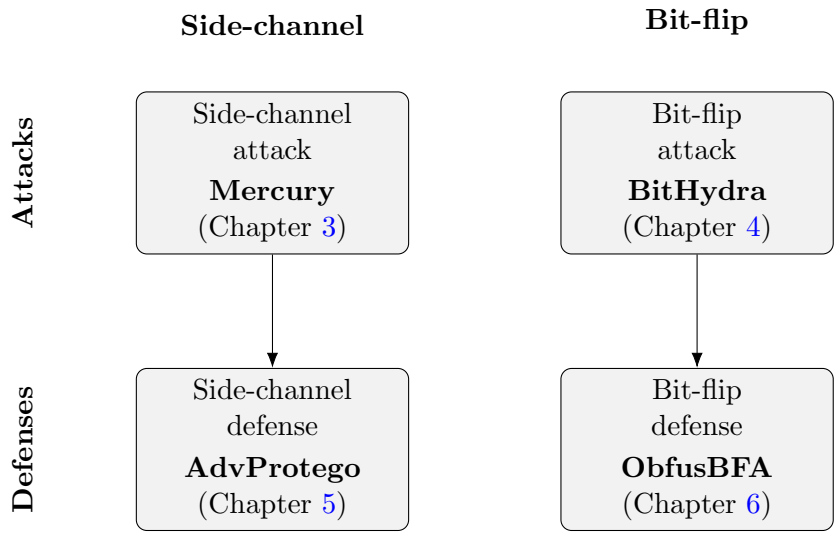


FIGURE 1.1: High-level organization of this thesis. The first two technical chapters investigate hardware vulnerabilities via attacks (`Mercury` and `BitHydra`), while the subsequent two chapters propose corresponding defense frameworks (`AdvProtego` and `ObfusBFA`) across modern deep learning systems.

2. Bit-flip inference-cost attack on large language models. The second part of the thesis introduces `BitHydra`, a novel class of bit-flip attacks that explicitly target the *inference cost* of large language models rather than their prediction accuracy. Under realistic hardware fault models (e.g., Rowhammer-like DRAM disturbance), `BitHydra` directly manipulates the model's parameters so that the induced overhead persists across users and queries. Concretely, `BitHydra` focuses on the output embedding row corresponding to the end-of-sequence (`<EOS>`) token and minimizes a loss that suppresses the probability of emitting `<EOS>` while regularizing task accuracy. The resulting gradient information is used to select a small set of critical bits whose flipping leads to extremely long, yet syntactically coherent, outputs that frequently hit the maximum generation length. Extensive experiments on diverse LLM families show that only a handful of flipped bits can cause substantial and persistent inference-cost inflation, revealing a new and practically impactful dimension of bit-flip threats in large-scale deployment settings.

3. Unified side-channel defense framework via adversarial noise injection. The third part of the thesis moves to the defender's perspective and introduces `AdvProtego`, a unified framework that injects adversarially crafted execution noise to disrupt machine-learning-assisted side-channel analysis such as `Mercury`. Instead of relying on ad hoc heuristics such as random delays or coarse-grained noise, `AdvProtego` treats the attacker's side-channel classifier itself as a target and

formulates defense as a bi-level optimization problem. Given a set of candidate "noise actions" that can be realized on the target platform (e.g., synthetic operations, obfuscation kernels, or controlled scheduling perturbations), `AdvProtego` searches for combinations and placements of these actions that maximally reduce the attacker's model similarity or utility, under constraints on accuracy and performance overhead of the protected model. The framework is instantiated across multiple leakage modalities and platforms, showing that carefully optimized noise can significantly degrade the success rate of state-of-the-art side-channel attacks while maintaining acceptable overheads in realistic deployments.

4. Holistic defense framework against versatile bit-flip attacks. The fourth part of the thesis proposes `ObfusBFA`, a holistic defense framework against versatile bit-flip attacks on deep learning systems. Unlike prior defenses that focus on a single abstraction level, `ObfusBFA` jointly considers vulnerabilities at the model level (weights, activations) and at the code level (compiled kernels, libraries, and code generation). It comprises three key components: (i) a vulnerability searcher that profiles bit-flip sensitivity across parameters and low-level artifacts; (ii) an obfuscation pattern generator that produces structured redundancy, dummy operations, and diversified encodings to scatter critical information; and (iii) an enforcer that integrates the resulting patterns into mainstream deep learning infrastructures and compiler stacks (e.g., C++ runtimes, PyTorch-based systems, and TVM-based backends). Across a wide range of model- and code-level BFAs, including adaptive attackers aware of the defense, `ObfusBFA` effectively downgrades carefully orchestrated flips to near-random behavior, achieving strong robustness with moderate storage and runtime overhead.

1.3 Contributions of the Thesis

This thesis makes the following main contributions to the security of deep learning systems from a hardware-oriented perspective:

- **Automated remote side-channel model extraction on realistic deep learning platforms.** We design and implement `Mercury`, an end-to-end side-channel attack that automatically recovers the architecture of a deployed

DNN from noisy execution traces on a realistic accelerator platform (Chapter 3). `Mercury` builds a time-to-digital-converter (TDC)–based monitoring substrate and formulates architecture recovery as a sequence-to-sequence learning problem solved by RNN-CTC and Transformer-based decoders. Unlike prior works that rely on strong physical access, simplified accelerator designs, or layer-level targets [17–23, 29], `Mercury` demonstrates that (near) full model architectures can be extracted in a largely automated fashion from remote, noisy traces in a multi-tenant setting.

- **Bit-flip inference-cost attacks against large language models.** We introduce `BitHydra`, the first bit-flip attack that explicitly targets the *inference cost* of LLMs instead of only degrading accuracy or causing misclassification (Chapter 4). Under realistic hardware fault models (e.g., Rowhammer-like DRAM disturbance), `BitHydra` focuses on the output embedding row corresponding to the end-of-sequence (`<EOS>`) token and optimizes a loss that suppresses `<EOS>` while preserving task utility. The resulting gradients guide a search for a small set of critical bits whose flipping forces the model to generate extremely long, syntactically coherent outputs that frequently hit maximum generation length. Experiments on diverse LLM families show that only a handful of flipped bits can induce persistent, cross-user inference-cost inflation, revealing a new and practically important dimension of bit-flip threats beyond traditional accuracy-oriented attacks.

- **A principled, adversarial-noise–based defense framework for side-channel leakage.** We propose `AdvProtego`, a unified framework that protects deep learning models from machine-learning-assisted side-channel analysis by treating the attacker's classifier as an explicit target (Chapter 5). `AdvProtego` formulates defense as a bi-level optimization problem that learns execution-noise strategies (e.g., synthetic operations, scheduling perturbations) to minimize model similarity or extraction accuracy from traces, subject to constraints on accuracy and performance of the protected model. Instantiated on multiple leakage modalities and platforms, `AdvProtego` substantially degrades the success of state-of-the-art side-channel attacks [17–21, 29] while maintaining modest overhead, providing a principled alternative to ad hoc countermeasures such as random delays or coarse-grained noise.

- **A holistic, implementation-aware defense framework against versatile bit-flip attacks.** We develop `ObfusBFA`, a holistic defense framework that jointly considers model-level and code-level vulnerabilities to bit-flip attacks in deep learning systems (Chapter 6). `ObfusBFA` consists of (i) a vulnerability searcher that profiles bit-flip sensitivity across parameters and low-level artifacts; (ii) an obfuscation pattern generator that produces structured redundancy, dummy operations, and diversified encodings; and (iii) an enforcer that integrates these patterns into mainstream deep learning infrastructures and compiler stacks (C++ runtimes, PyTorch-based systems, TVM-based backends). Across a wide range of model- and code-level BFAs, including adaptive attackers aware of the defense, `ObfusBFA` downgrades carefully orchestrated flips to near-random behavior, significantly improving robustness with moderate runtime and storage overhead compared to prior, more fragmented countermeasures.

Beyond individual mechanisms, this thesis provides an integrated view of how hardware-level behaviors (side-channel leakage, memory faults) and low-level implementations (libraries, compiler transformations, quantization schemes) interact with deep learning models in practice. By jointly studying remote side-channel model extraction, adversarial-noise–based defenses, inference-cost–oriented bit-flip attacks, and holistic obfuscation frameworks, the thesis identifies common design principles and trade-offs that are broadly applicable to the secure deployment of DNNs and LLMs in realistic environments.

1.4 Roadmap

This thesis is organized as follows.

Chapter 1 provides an overview of the thesis, including the background, motivation, main work, and contributions.

Chapter 2 reviews related work on side-channel attacks and defenses, bit-flip attacks on deep learning models, and hardware- and system-level security mechanisms for DNN applications.

The main body of this thesis is organized into two parts with 4 works.

Part I: Hardware-oriented Attacks. We investigate novel attack vectors leveraging hardware leakages and faults to compromise model confidentiality and availability.

- Chapter 3 presents `Mercury`, an automated remote side-channel attack for model extraction. It leverages a TDC-based monitor and sequence-to-sequence learning to recover full model architectures from noisy execution traces on realistic deployment settings.
- Chapter 4 introduces `BitHydra`, a bit-flip inference-cost attack on large language models. Instead of relying on adversarial inputs, it directly perturbs model parameters to suppress the end-of-sequence token, inducing persistent, cross-user overhead.

Part II: Hardware-oriented Defenses. We propose robust and practical defense frameworks to mitigate hardware-level threats across various system layers.

- Chapter 5 describes `AdvProtego`, a unified framework that defends deep learning models against machine-learning-assisted side-channel analysis by injecting adversarially crafted execution noise.
- Chapter 6 proposes `ObfusBFA`, a holistic defense framework against versatile bit-flip attacks. It covers both weight-level and code-level vulnerabilities by introducing structured redundancy and dummy operations into the model and binary code.

Finally, Chapter 7 concludes the thesis and outlines future research directions on hardware-oriented security for deep learning systems.

1.5 Ethical Considerations, Responsible Disclosure, and Reproducibility

Because this thesis develops offensive techniques alongside defensive ones, we briefly clarify the ethical framing, disclosure practice, and reproducibility measures that govern the work.

Ethical considerations. The attacks studied in this thesis (`Mercury` and `BitHydra`) were developed and evaluated exclusively in controlled laboratory settings, using our own hardware testbeds, publicly available models, and standard benchmark datasets. No production system, third-party service, or real user data was targeted at any point, and no personally identifiable or otherwise sensitive information was collected. The purpose of studying these attacks is defensive: understanding realistic hardware-oriented threats is a prerequisite for designing effective countermeasures, and each attack contribution in this thesis is deliberately paired with, or motivates, a corresponding defense (`AdvProtego` and `ObfusBFA`). This work was conducted in accordance with the research-integrity and ethics policies of Nanyang Technological University.

Responsible disclosure. The attack techniques presented here rely on capabilities that constrain their immediate weaponization, such as co-location on a shared accelerator, the ability to instantiate on-chip sensors, or the ability to induce precise memory faults; they do not constitute turn-key exploits against deployed services. Where our findings concern widely used, openly available artifacts, such as the open-source NVDLA accelerator, mainstream deep learning frameworks, and compiler toolchains, we follow responsible-disclosure norms by describing vulnerabilities at a level sufficient for the community and maintainers to reproduce and mitigate them, while publishing the mitigations (`AdvProtego`, `ObfusBFA`) together with the attacks so that defenders are not left without recourse. The relevant results have undergone peer review at established venues, which further supports responsible dissemination.

Reproducibility. To facilitate independent verification, each technical chapter meticulously documents its threat model, datasets, model architectures, and experimental configurations (including hardware setups, software frameworks, and hyperparameters). While some evaluations depend on specific hardware environments (e.g., an FPGA board hosting NVDLA), the core algorithmic steps are detailed sufficiently to allow generalization and validation across comparable platforms. Furthermore, to actively support community-driven research, the source code and implementation scripts for the two attack frameworks (`Mercury` and `BitHydra`) have been open-sourced and are publicly available on GitHub. The codebases for the comprehensive defense frameworks (`AdvProtego` and `ObfusBFA`), which are

deeply integrated with low-level compiler toolchains and hardware-specific configurations, are currently undergoing codebase cleanup and internal review, and are being prepared for future public release.

Chapter 2

Related Works

2.1 Side-Channel Attacks and Their Defenses

2.1.1 Side-Channel Attacks

Side-channel attacks (SCAs) exploit unintended information leakage—originating from the physical or microarchitectural behavior of the underlying hardware—to infer sensitive data without directly compromising the logical security boundaries of the system. Unlike traditional software exploits, these attacks leverage observable characteristics such as execution timing, power consumption, and electromagnetic (EM) emissions.

Historically, SCAs were primarily targeted at cryptographic implementations [65–68]. However, with the pervasive deployment of deep learning, researchers have recently demonstrated that similar methodologies can be adapted to compromise the confidentiality of Deep Neural Networks (DNNs), enabling the extraction of proprietary model architectures [23, 29, 69] and weights [18, 19]. Based on the information medium (or leakage source) exploited, the taxonomy of SCAs can be classified as follows:

Power Side-Channels. Power side-channel attacks exploit the correlation between the dynamic power consumption of a device and the data being processed or operations executed. Since switching activity in transistors dictates power usage, monitoring these fluctuations can reveal sensitive internal states. Early research

primarily targeted cryptographic primitives; for example, analyzing power traces during the execution of the AES S-box [65, 66] or the square-and-multiply operations in RSA [67] to recover secret key bytes. More recently, this vector has been adapted to target DNNs. Li et al. [18] demonstrated the application of differential power analysis (DPA) to extract weights from systolic-array–style DNN accelerators. To bypass the requirement for physical proximity, researchers have developed remote attacks suitable for multi-tenant FPGA cloud environments. In these settings, attackers deploy co-resident IP cores—such as Ring Oscillators (ROs) or Time-to-Digital Converters (TDCs)—that share the Power Distribution Network (PDN) with the victim to monitor voltage fluctuations. Zhang et al. [23] utilized ROs to remotely steal neural network structures, while Meyers et al. [22] targeted Binary Neural Networks (BNNs) on the FINN framework [69] to recover layer folding parameters. Similarly, Tian et al. [29] showed that template-based analysis of TDC traces could enable partial architecture recovery of workloads on VTA-like accelerators [70].

Electromagnetic (EM) Side-Channels. Similar to power analysis, EM attacks measure the electromagnetic radiation emitted by the processor or memory bus during computation. These emanations can be captured by near-field probes to reconstruct fine-grained execution patterns, such as the activation functions or layer configurations of a victim model. The viability of this vector was established by Quisquater and Samyde [71], who demonstrated that EM radiation leaks information comparable to direct power consumption. To quantify these leakages, attackers typically rely on statistical abstractions such as the Hamming distance model [72], the Hamming weight model [73], or other improved leakage models [74]. Recently, research has pivoted toward the intersection of EM analysis and machine learning, both as a tool for enhanced cryptanalysis—such as Wang et al. [75] utilizing deep learning to recover AES keys from far-field emissions—and as a threat vector against ML hardware itself. In the domain of model extraction, Yoshida et al. [17] leveraged EM leakage to recover parameters from a custom CNN accelerator, while Yu et al. [19] targeted custom accelerator pipelines and Binary Neural Networks (BNNs), showing that simple EM patterns could reveal model weights and architectures. Addressing commercial-grade hardware, Gupta et al. [21] demonstrated an EM-based attack against the NVIDIA Deep Learning Accelerator (NVDLA); however, despite targeting a realistic platform, this method remains a physical attack

requiring manual intervention to segment execution traces, lacking the automation necessary for scalable exploitation.

Timing Side-Channels. Timing attacks exploit variations in the execution time of sensitive operations to infer internal secrets. These variations typically arise from data-dependent conditional branches, cache contention, or hardware optimizations such as early termination, allowing adversaries to deduce input data or model depth based on total inference latency. The foundational feasibility of this vector was established by Kocher [76], who performed the first successful timing side-channel attack. Early research focused heavily on cryptographic primitives, targeting processing time differences in the AES S-box [77, 78] or RSA modular multiplication operations [79, 80] to recover secret key bytes. More recently, machine learning techniques have been integrated to enhance the resolution of these attacks, enabling finer-grained extraction results on AES implementations [81] as well as the recovery of DNN hyperparameters [82].

Microarchitectural Side-Channels (Cache/Memory). These attacks leverage shared microarchitectural resources, such as CPU caches or DRAM buffers, to infer execution patterns. By monitoring contention, access patterns, or throughput variations, attackers can deduce the control flow of runtime libraries or the volume of memory transactions.

In the domain of *cache side-channels*, adversaries utilize established primitives—such as Prime+Probe [83], Evict+Time [84], and Flush+Reload [85]—to manipulate cache lines and measure victim access latencies. While widely used for cryptographic key extraction [86], these techniques have evolved to exploit finer-granularity leakage sources, such as cache coherence states [87] and port contention in multi-threaded processors. Recent research has successfully adapted these primitives to the deep learning domain, demonstrating their efficacy in extracting DNN architectures by monitoring the invocation of unique layers or library functions [10].

Complementing cache attacks, *memory side-channels* focus on monitoring the interfaces between peripheral components and memory (*e.g.* memory bus, memory controller, or PCIe events), often referred to as bus snooping [88]. For instance, Hua et al. [20] and Zhao et al. [88] demonstrated that monitoring memory bus traffic on custom accelerators could reveal model weights and architectures. In more complex GPU environments, DeepSniffer [9] analyzes electromagnetic emanations

from the GPU memory bus to reconstruct model architectures, while Hermes [89] exploits unencrypted PCIe traffic patterns to map data payloads to specific model layers.

2.1.2 Side-Channel Attack Defenses

Countermeasures against side-channel attacks (SCAs) can be broadly categorized into three strategies: masking, hiding, and isolation.

Masking. Masking aims to detach the power consumption from the sensitive data being processed by randomizing intermediate variables [32, 33]. This is typically achieved by splitting internal values into multiple statistically independent shares, ensuring that any subset of shares reveals no information about the original secret. In the context of DNNs, Dubey et al. [90] designed masked circuits for neural network components such as adders, activation functions, and multiplexers. Later, they proposed a revised scheme incorporating shuffling into a masked Kogge-Stone adder (KSA) architecture [91]. Similarly, Maji et al. [92] introduced a threshold implementation (TI) using Trivium stream ciphers for random number generation, albeit with significant area (64%) and energy (5.5×) overheads. Other approaches leverage secure computation techniques; for instance, Hashemi et al. [93] proposed HWGN2 using garbled circuits and secure function evaluation (SFE). More recently, a secure RISC-V-based coprocessor was developed to execute masked neural network operations like weighted summations and activation functions [94]. However, masking generally only protects against lower-order attacks and cannot prevent the recovery of model architectures or resist attackers with higher-order profiling capabilities.

Hiding. Hiding techniques aim to reduce the signal-to-noise ratio (SNR) of the leakage or flatten the side-channel observables, making it difficult for attackers to extract exploitable information [95]. One approach is *hiding in the time dimension* via clock misalignment or random delays [36, 37, 96]. By injecting random waiting times between instructions, these methods distort temporal patterns. Another approach is *hiding in the amplitude dimension* via noise compensation [34, 37], where noise is injected into power or memory activity to obscure the leakage. Pothukuchi et al. [97] demonstrated reshaping power dissipation using formal control techniques and Gaussian sinusoid noise. Furthermore, *constant execution* ensures that

execution patterns are independent of sensitive data [30, 31], thereby removing data-dependent timing or power variations [17, 18]. Finally, at the software level, model architecture *obfuscation* strategies perturb the architecture or training process to prevent attackers from extracting correct model information [98–100]. Despite these efforts, hiding techniques have limitations: ML-based attacks can often bypass clock misalignment [101] or extract information even from low-SNR traces, and simple noise can be filtered out via signal processing with repeated measurements.

Isolation. Isolation focuses on eliminating cross-tenant interference and closing leakage channels at the architectural level. This strategy is particularly relevant in multi-tenant cloud FPGA or processor environments, where it aims to physically or logically separate the victim's execution environment from potential attackers. By enforcing spatial or temporal isolation—such as partitioning cache sets, DRAM banks, or FPGA resources—these methods attempt to cut off the transmission of leakage to co-resident malicious IP cores. This technique has been widely applied across various hardware platforms, including FPGAs [35], CPU caches [41], GPUs [102], Trusted Execution Environments (TEEs) [103], and DRAM [63].

2.2 Bit-Flip Attacks and Their Defenses

The deployment of Deep Neural Networks (DNNs) on heterogeneous hardware has exposed them to hardware-level fault injection threats. Among these, Bit-Flip Attacks (BFAs) have emerged as a critical vector, where adversaries exploit physical vulnerabilities in memory systems, such as the Rowhammer effect [15, 16], laser [14], or undervolting [13], to corrupt model integrity without requiring physical access to the device.

2.2.1 Bit-Flip Attacks against Deep Learning Systems

BFAs against deep learning systems can be broadly categorized based on the abstraction level they target: the high-level model parameters (Model-level) or the underlying software stack (Code-level).

2.2.1.1 Model-Level Attacks

Model-level BFAs [16, 24–28] directly manipulate the weights, biases, or activations of a DNN to alter its inference behavior. Due to the massive parameter space of modern models, these attacks typically employ sophisticated search strategies, such as gradient ranking or progressive search to pinpoint the most vulnerable bits. These attacks can be further classified by their adversarial objective:

- **Untargeted Attacks (Accuracy Degradation):** The primary objective here is to degrade the overall inference accuracy of the model, effectively causing a Denial-of-Service (DoS) or rendering the model useless. Rakin et al. [25] first demonstrated this on DNNs, showing that flipping fewer than 50 bits in the quantized weights of a ResNet model could reduce its accuracy to that of a random guess. This fragility arises because specific bits (e.g., exponent or MSB) can drastically alter weight magnitudes. Recent works have extended this threat to LLMs. For instance, *AttentionBreaker* causes a catastrophic collapse in performance, reducing Massive Multitask Language Understanding (MMLU) accuracy from 67.3% to 0% [104]. Similarly, *SilentStriker* degrades task performance while maintaining output naturalness by reformulating the attack objective to suppress key output tokens [105].

- **Targeted Attacks:** Unlike untargeted attacks that aim for indiscriminate degradation, targeted attacks are stealthier and seek to force the model into specific, adversary-chosen behaviors while preserving normal functionality for benign inputs. These can be further classified by their adversarial objective: (1) *Functional Integrity Attacks (Backdoor Attacks).* The most common form of targeting aims to alter the model's predicted output. For instance, *TBT* targets critical neurons in the final layer to embed a backdoor trigger [26]. *ProFlip* utilizes Jacobian-based saliency map attacks (JSMA) to progressively identify vulnerable bits across multiple layers for targeted misclassification [24]. Similarly, *TA-LBF* formulates the bit search as a binary integer programming problem, solving it with the Alternating Direction Method of Multipliers (ADMM) to find precise bits that flip the prediction of specific victims [28]. (2) *Safety Alignment Attacks (Jailbreak Attacks).* Recent research has focused on compromising the safety alignment of LLMs. These attacks flip specific weight bits to bypass alignment guardrails. *PrisonBreak*

employs a progressive bit-search guided by a specialized jailbreak loss to identify vulnerable bits [106]. Extending this to quantized models, Zahran et al. [107] utilize a Straight-Through Estimator (STE) to effectively search for critical bits in low-precision weights.

2.2.1.2 Code-Level Attacks

Recent research has revealed that the software infrastructure supporting DNN inference is also susceptible to BFAs. Unlike model-level attacks that degrade accuracy via numerical deviation, code-level attacks typically target the control flow integrity of the compiled executables [108] or computation libraries [56].

- **ML Executable Attacks:** Compilers like TVM translate high-level models into binary executables. Chen et al. [108] demonstrated that flipping bits in the `.text` section of these executables can hijack the inference logic, leading to sharp reductions in accuracy.
- **Library Attacks:** Deep learning frameworks rely heavily on optimized linear algebra libraries (e.g., OpenBLAS). Li et al. [56] proposed *FrameFlip*, showing that a single bit flip in the conditional jump opcodes (e.g., `je` to `jne`) within these libraries can reverse control flow logic. This corrupts the General Matrix Multiply (GEMM) operations, causing universal inference depletion across all DNN applications running on the compromised system.

2.2.2 Defenses against Bit-Flip Attacks

Countermeasures against BFAs are generally divided into hardware-level mitigation and DNN-specific defenses.

2.2.2.1 Hardware-Level Mitigation

Defenses at the hardware level aim to eliminate the physical root cause of bit flips, such as Rowhammer. Standard techniques include Error-Correcting Codes (ECC) [62] and Target Row Refresh (TRR). More advanced academic solutions

include generic hardware extensions like *MASCAT* [59], *ZebRAM* [63], and isolation domains like *Siloz* [64]. However, these approaches often require specialized hardware support, incur high manufacturing costs, and may be difficult to deploy on legacy systems [16].

2.2.2.2 DNN-Specific Defenses

Given the high cost of hardware fixes, software-level defenses tailored for DNNs have gained attention. These can be categorized into robustness hardening and runtime integrity checking.

- **Model Hardening:** This strategy involves modifying the model architecture or training process to make it resilient to bit flips. *Aegis* introduces dynamic multi-exit architectures to filter out corrupted inferences [44]. Weight reconstruction [45] and binarization techniques (e.g., *BIN* [46], *RA-BNN* [47]) restrict the value range of weights to limit the impact of flips. Other works employ randomization, such as *DeepShuffle* [48] and *HammerDodger* [49], to shuffle scheduling or weights, breaking the attacker's addressing precision.
- **Runtime Integrity Checking:** This approach monitors execution to detect anomalies. Techniques include checksum-based verification (e.g., *RADAR* [51], *ACCHashtag* [53]), hash signatures (*Hashtag* [54]), and auxiliary checker networks (*DeepDyve* [55]). While effective, these methods often introduce significant runtime latency, making them less suitable for real-time applications.

2.3 Summary

This chapter reviews the evolving landscape of hardware-oriented security threats to deep learning systems, focusing on two primary vectors: side-channel leakage and fault-induced bit flips. While existing literature has established the theoretical and practical feasibility of these threats, significant gaps remain regarding their automation, scope, and the unification of defenses.

First, in the domain of **side-channel attacks**, prior solutions are often constrained by strong physical access requirements [17, 18] or limited to simplified architectures.

Remote attacks, while promising, typically lack the automation required for the end-to-end recovery of complex, commercial-grade models [21, 29]. To address this, Chapter 3 introduces `Mercury`, an automated remote attack capable of recovering full model architectures from noisy traces on the NVDLA. Conversely, existing defenses often struggle to balance security with performance or fail against modern ML-assisted analysis. Chapter 5 addresses this by proposing `AdvProtego`, a unified framework that utilizes adversarial execution noise to disrupt such analysis effectively without necessitating model retraining.

Second, regarding **bit-flip attacks (BFAs)**, current research predominantly focuses on compromising functional integrity—either through indiscriminate accuracy degradation or targeted misclassification [24–26]. The dimension of computational efficiency remains largely unexplored in the context of parameter manipulation; existing inference cost attacks rely on transient, input-based vectors that are self-targeting [109, 110]. Chapter 4 bridges this gap with `BitHydra`, the first BFA designed to target inference cost persistently by manipulating model weights of LLMs. Finally, defenses against BFAs are currently fragmented, targeting either model weights or executables in isolation [56, 108]. Chapter 6 presents `ObfusBFA`, a holistic defense that safeguards both model parameters and low-level code through randomized obfuscation, ensuring comprehensive protection against versatile hardware faults.

Part I

Hardware-oriented Attacks

Chapter 3

`Mercury`: An Automated Remote Side-channel Attack to NVIDIA Deep Learning Accelerator

DNN accelerators have been widely deployed in many scenarios to speed up the inference process and reduce the energy consumption. One big concern about the usage of the accelerators is the confidentiality of the deployed models: model inference execution on the accelerators could leak side-channel information, which enables an adversary to preciously recover the model details. Such model extraction attacks can not only compromise the intellectual property of DNN models, but also facilitate some adversarial attacks.

Although previous works have demonstrated a number of side-channel techniques to extract models from DNN accelerators, they are not practical for two reasons. (1) They only target simplified accelerator implementations, which have limited practicality in the real world. (2) They require heavy human analysis and domain knowledge. To overcome these limitations, this chapter[1] presents `Mercury`, the first automated remote side-channel attack against the off-the-shelf NVIDIA DNN accelerator. The key insight of `Mercury` is to model the side-channel extraction process as a sequence-to-sequence problem. The adversary can leverage a time-to-digital converter (TDC) to remotely collect the power trace of the target model's inference. Then he uses a learning model to automatically recover the architecture

[1]The content of this chapter is published in [8].

details of the victim model from the power trace without any prior knowledge. The adversary can further use the attention mechanism to localize the leakage points that contribute most to the attack. Evaluation results indicate that `Mercury` can keep the error rate of model extraction below 1%.

3.1 Introduction

Modern deep learning technology exhibits a computationally intensive trend to perform more complex tasks. This leads to the popularity of adopting specific hardware to accelerate computation and reduce energy consumption. Field Programmable Gate Arrays (FPGAs) are a prevalent choice for implementing DNN accelerators, and have been widely deployed in large-scale datacenters by various cloud providers, such as Amazon EC2 F1 [111] and Microsoft Catapult [112].

However, DNN accelerators in the cloud pose new security challenges, and one significant threat is the model extraction attack [113–115]. Generally, cloud providers adopt the multi-tenancy policy that facilitates multiple users to share the same FPGA board to enhance the resource utilization [116]. Although the circuits of different users can be logically separated, they still share the same power distribution network (PDN). A prior study [117] shows that the supply voltage at different locations of a PDN is not constant and depends on the activity of the logic. Therefore, the voltage fluctuation becomes a critical side-channel to leak sensitive information across different parts of the FPGA. The adversary can abuse the multi-tenancy feature to *remotely* launch an on-chip monitor on the same board with the victim's deep learning model and steal its information *without requiring physical access to the hardware*. It has been commonly referred to as remote side-channel attacks in prior works [29, 118, 119], exhibiting more flexibility and practicality than physical attacks [17–21].

Although various side-channel techniques have been proposed to attack FPGA-based DNN accelerators [120], there is still a huge gap to apply them in practice. **(1) Simplified implementations**. A majority of works only target their home-made DNN accelerators [17–20, 22, 23], which are normally simplified, and easy to break. In contrast, full-fledged architectures in the real world usually involve more complex structure designs and optimizations, which complicate the side-channel

TABLE 3.1: Summary of existing works

Attack	Impl.	Model	Target	Aim	Remote	Automated	# of runs
[29]	VTA[70]	CNN	Layer	A	✓	✕	50
[17]	Home-grown	MLP	Model	W	✕	✕	60K
[18]	Home-grown	CNN	Multiplication	W	✕	✕	40K
[19]	Home-grown	BNN	Model	A,W	✕	A:✕; W:✓	10K
[20]	Home-grown	CNN	Model	A,W	✕	✕	–
[21]	NVDLA	CNN	Layer	A	✕	✕	–
[23]	Home-grown	CNN	Layer	A	✓	✕	–
[22]	FINN[69]	BNN	Layer	A	✓	✕	100
Ours	NVDLA	CNN	Model	A	✓	✓	1

For Aim: A refers to architecture recovery while W refers to weight recovery.

analysis and attacks. **(2) Simplified models**. A number of works aim to steal Binary Neural Networks (BNNs) with binary values of model weights and activations [19, 22]. The feasibility of their extension to non-binarized DNNs is dubious. **(3) Simplified attack goals**. Many studies only attack individual network layers, but cannot achieve end-to-end extraction of the entire model [21–23, 29]. This restricts their practical values. Table 3.1 summarizes the comparisons of prior studies.

To address the above limitations, we propose `Mercury`, a novel power side-channel attack to steal the architecture of DNN models on the practical NVDIA Deep Learning Accelerator (NVDLA). NVDLA serves as the standard way for DNN accelerator designs, and has been the mainstream implementation in many products. It includes the complex hardware-software stack, execution pipeline and runtime environment. Such designs cause the side-channel traces to be highly redundant and noisy, and exhibit no one-to-one relationship with the victim model. This significantly increases the attack difficulty (See §3.3.2 for more discussions).

The core concept of `Mercury` is to model the side-channel extraction process as a sequence-to-sequence learning task. We leverage powerful RNN-CTC and Transformer models to recover the architecture of the victim model from the power trace of its inference. Besides, we utilize the attention mechanism to localize the leakage point in the side-channel trace, which can shed light on the potential vulnerability of DNN accelerators and defense directions.

To our best knowledge, the only side-channel attack targeting NVDLA is [21]. However, it has the following limitations: (1) it is not a remote attack and requires physical access to the victim device; (2) it requires the attacker to manually split the execution trace for different layers, and extract each layer individually; (3) it

needs to train multiple learning models for each hyper-parameter. `Mercury` can overcome these limitations: it can be launched remotely in the multi-tenant cloud context; it enables the adversary to automatically steal the model without any manual analysis or prior knowledge of the victim system. Besides, `Mercury` is cost-efficient: the adversary only needs to train one end-to-end model, and run one inference process for extraction of the entire model, while prior attacks need thousands of rounds [17–19].

Note that our attack goal is to steal the model architecture, which is the same as some previous works [9, 10, 121, 122]. Stealing the architecture has high financial incentive as it is basis for building more valuable intellectual properties at incremental cost [10, 122, 123]. Besides, obtaining the architecture details can facilitate other attacks such as adversarial examples and membership inference. We provide two case studies in §3.6 to show how `Mercury` can enhance these attacks. Some studies proposed methods to extract the model weights [17–20]. However, these attacks need to physically access the victim device or inject hardware trojans to collect more informative traces (Table 3.1). How to remotely extract model weights is challenging and we leave it as a future work.

We perform extensive experiments to validate the effectiveness of `Mercury`. Evaluation results show that `Mercury` can recover victim's model information with an error rate of 1%. It is robust enough to resist certain levels of noise without a significant performance drop.

3.2 Background

3.2.1 NVIDIA Deep Learning Accelerator (NVDLA)

NVDLA is an open-source configurable architecture designed by NVIDIA to accelerate deep learning inference. It is capable of computing convolution, activation, pooling, and normalization operations in the model inference. NVDLA can be configured as a large or small implementation, differing in the dimension of the cores and implementation of some specific engines (e.g., Rubik and DMA engines) [124].

Figure 3.1 shows the architecture overview of NVDLA. It can be divided into two parts, i.e. hardware design and software design. Hardware design is built as a series

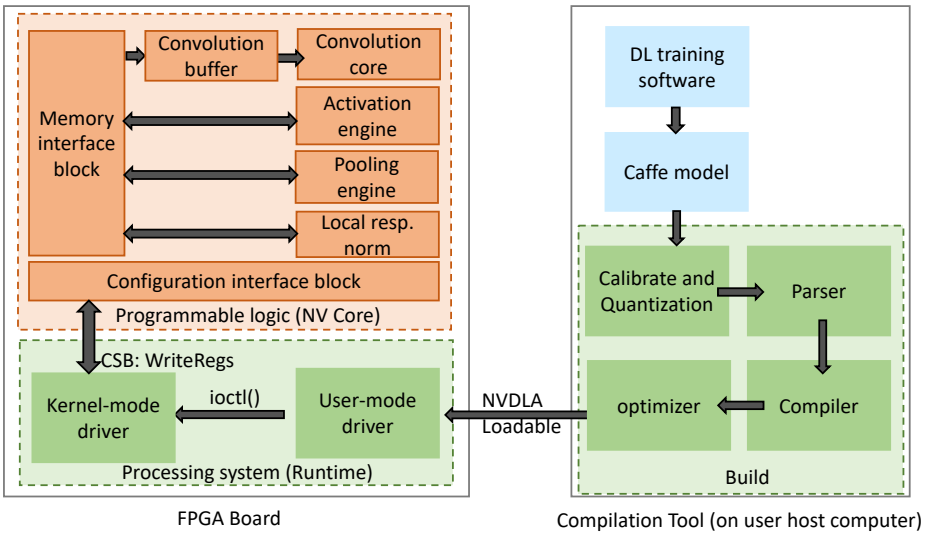


FIGURE 3.1: Architecture overview of NVDLA

of pipeline stages containing various types of engines to regulate the behaviors of FPGA boards. Software design connects the users and hardware components, and is responsible for building and loading the DNN model for the FPGA to execute. The software design further consists of two components: (1) the compilation tools use the model pre-compiled by Caffe to generate a network of hardware layers supported by NVDLA, called loadable, which is calibrated by TensorRT. (2) The runtime environment processes the calibrated loadable and runs it directly in the NVDLA environment.

3.2.2 Voltage Drop Sensor on FPGA

All the components on an FPGA chip share one power distribution network (PDN). Intensive switching activities may cause voltage fluctuations in the PDN. The PDN can be modeled as an RLC circuit, where a resistor (R), an inductor (L), and a capacitor (C) are connected in series or in parallel. Therefore, the transient voltage drop seen by a circuit can be modeled as [67]: $V_{drop} = IR + L\frac{di}{dt}$, where the transient response term $L\frac{di}{dt}$ reflects the intensity of the switching activity on the FPGA. In typical CMOS circuits, combinational logic delays can be modeled to be inversely proportional to the voltage supplied to each gate [125]. Hence, the information of the switching activities can be inferred from the logical delay.

In this paper, we use a time-to-digital converter (TDC) to read the combinational logic delay, where a clock signal propagates through a chain of buffers as the voltage

drop sensor. As there are discrepancies in switching activities for different types of calculation on other parts of the FPGA, the voltage drop value may be different, which leads to different delay measurements in TDC. These different delays can cause different propagation lengths in the delay line, resulting in different values in the latches. Therefore, the activities of other circuits can be identified through the readout of the TDC. This has been demonstrated in various studies [29, 118, 119].

3.2.3 Profiled Side-Channel Attacks

Profiled side-channel attacks are one of the most powerful attacks [126]. The adversary generates a profile through a similar or the same device of the target, and computes the secret information by matching the profile with the victim's execution trace. A profiled side-channel attack can be divided into two phases. First, a profiling phase allows the adversary to characterize its physical leakages when running the target application. Suppose the adversary has an input secret set $\mathbf{s} = \{s_1, ..., s_n\}$. He obtains N side-channel traces $\mathbf{T_{i,n}}$ for each input s_i, and builds the mapping $f : \mathbf{T_{i,n}} \mapsto s_i$. This mapping can be learned through template creating and machine learning methods. Second, an exploitation phase is launched to perform secret recovery by profile matching. The adversary performs the side-channel attack using the mapping f. With additional q traces $T'_1, ..., T'_q$ collected from the device under attack, the secret s'_i can be guessed as $s'_i = f(T'_i)$.

In this thesis, we adopt machine learning for profile learning and matching to attack FPGA-based DNN accelerators. It has several advantages. First, the adversary is able to extract information even if there are no visible patterns in the power trace. As most calculations on FPGAs are done in parallel, manually analyzing the pattern may be futile. Second, the adversary can utilize all information in a single power trace, as machine learning can effectively handle high-dimensional data. In contrast, traditional methods require the selection of points of interest (POI) to narrow down the information for the attack. Besides, machine learning models have higher resilience against the noise in the side-channel trace than conventional statistical methods.

3.2.4 Sequence-to-sequence (seq2seq) Learning

Seq2seq learning achieves state-of-the-art prediction accuracy in various tasks like speech recognition [127], machine translation [128], image captioning [129], question answering [130], etc. Since side-channel power traces are essentially sequential data, seq2seq learning is a natural fit for analyzing such leaking patterns. However, only very few works [9, 10] have applied seq2seq learning to side-channel analysis, and none of them realized automated attacks on FPGA-based accelerators, which have much more noise and more complicated monitoring architecture.

Commonly used architectures for seq2seq learning includes Transformer [131], Connectionist Temporal Classification (CTC), and Recurrent Neural Network (RNN). `Mercury` adopts two models, i.e, Transformer and RNN-CTC, for side-channel extraction. In the RNN-CTC model, the output of the RNN is passed into the CTC decoder. Aligning the operation sequence with the variable-length power sequence presents a challenge. To address this, the CTC decoder introduces a "blank" label ϵ that serves no specific correspondence and can be easily excluded from the output. The Transformer model consists of an encoder and a decoder. The input sequence is transformed by the encoder, generating an abstract representation that captures the learned features. Subsequently, the decoder utilizes this abstract representation to predict the subsequent output step-by-step, building upon the previous output. To enhance feature learning from the event sequence, a convolution layer is introduced before the encoder.

3.2.5 Comparison with Software-based Model Extraction

While `Mercury` demonstrates the severe threat of side-channel-based model extraction, a large body of existing literature focuses on software-based (or query-based) model extraction attacks [132–134]. We present a comparison between software-based extraction and our hardware-based approach in Table 3.2.

In a typical software-based attack, the adversary interacts with the victim model via a publicly accessible prediction API. By sending a massive number of carefully crafted inputs—often including out-of-distribution (OOD) samples or synthetic data—and observing the output labels or confidence scores, the attacker trains a surrogate model to mimic the victim's decision boundary [133]. The primary goal

TABLE 3.2: Comparison between Software-based and Hardware-based (Side-Channel) Model Extraction.

Feature	Software-based (Query) Attacks	Hardware-based (Side-Channel) Attacks (`Mercury`)
Attack Vector	Prediction APIs (Input/Output pairs)	Physical/Microarchitectural signals (Power, Cache, EM)
Extraction Goal	Functional equivalence (Surrogate model)	Structural and architectural fidelity (Exact layer sequence)
Data Requirement	Thousands to millions of synthetic/OOD queries	A single or few execution traces
Attacker Access	Public API or black-box endpoint	Co-location on shared hardware (e.g., multi-tenant cloud)
Detectability	High (abnormal query volume, OOD inputs)	Low (passive monitoring, bypasses API rate limits)

is usually to achieve *functional equivalence* [134], meaning the surrogate model achieves comparable accuracy on the target task, even if its internal architecture completely differs from the victim's model. Furthermore, some query-based attacks attempt to infer the architecture [135], but they still rely on extensive hyperparameter searching and massive query budgets. In contrast, side-channel attacks like `Mercury` exploit unintended hardware leakages (e.g., power consumption captured via a TDC sensor).

In summary, while software-based attacks focus on stealing the utility of black-box APIs through heavy querying, hardware-based attacks like `Mercury` pose a stealthier, structural threat to models deployed in shared computing infrastructures.

3.3 Attack Overview

3.3.1 Threat Model

We follow the *same threat model* of remote power side-channel attacks in previous works [29, 118, 136]: the adversary and victim's accelerator share the same FPGA board (but logically separated) with the same PDN, which can be realized in the multi-tenant FPGA-based cloud. The adversary is able to deploy and fully control his own malicious circuits. However, he cannot physically access the FPGA board or control any part of the victim's circuit. He can only deduce the victim's activity from his own circuit.

Figure 3.2 illustrates the workflow of `Mercury`, which consists of two phases. (1) In the *profiling* phase, the adversary deploys various types of DNN models on a cloud FPGA board and collects the corresponding power traces with the TDC sensor. With the profiling information he can establish the seq2seq model f that predicts the model architecture from the power trace. (2) In the *exploitation* phase, the

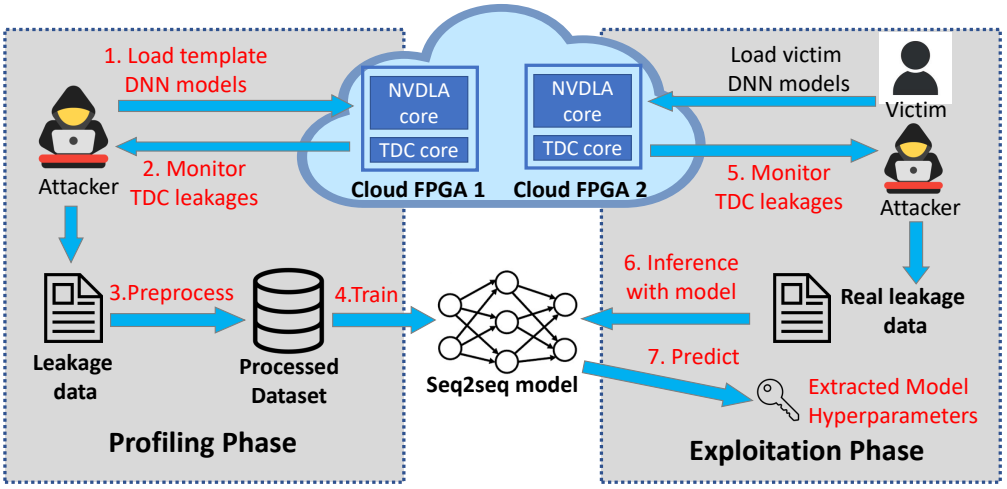


FIGURE 3.2: Attack workflow in `Mercury`

adversary deploys the TDC circuit on the same FPGA as the victim's model. He only needs to remotely collect the readout of this sensor for *one* inference process of the target model. Then he can use f to extract the victim model's architecture details.

3.3.2 Challenges of Attack Design

The NVDLA implementation raises the following challenges for designing remote side-channel attacks.

(1) NVDLA has complicated hardware and software architecture. From the software perspective, pre-compiled neural networks are first loaded by the user space runtime driver and then submitted to the kernel mode driver for the subsequent inference. The kernel mode driver schedules layer operations on NVDLA and programs the NVDLA registers to configure each functional block. From the hardware perspective, functional computation blocks are assembled as a pipeline to run submitted tasks, and the interrupt signal is asserted when the task is completed. Due to such complex hardware-software co-design, the captured power trace from NVDLA does not directly reflect the information of each layer and the data inside. It is mixed with other hardware-level power information on the FPGA, as well as the software-level power information related to CPU execution. This is much more complex than other simple accelerator designs, and significantly increases the attack difficulty.

(2) NVDLA has a parallel design to boost the performance. All functional blocks in NVDLA have duplicated register groups known as ping-pong buffers. Configurations of the next layer will be transmitted to the other register while running the current layer [137]. Due to this parallel design, NVDLA data flow may not have a strict one-to-one relationship with the layers of a model. These settings make the model extraction attack much more challenging.

(3) The collected side-channel data are redundant and noisy. A side-channel trace of an inference process can contain an extremely large amount of data points (more than 300K) due to the high monitoring frequency, which places heavy demands on the underlying compute infrastructure. Such a scale of data will induce great pressure on the data processing, making it infeasible to perform manual analysis as adopted in existing works. Moreover, the raw side-channel data may exhibit random noise originating from the hardware activities, a consequence of the intricate system optimization and runtime dynamics. This noise has the potential to substantially diminish the accuracy of the extraction process.

3.4 Design Details

To address the above challenges, we introduce innovative designs for the power monitor, dataset processing and seq2seq model. The detailed mechanisms are elaborated below.

3.4.1 TDC-based Power Monitor

`Mercury` utilizes TDC readouts to infer power information from the inference process. Figure 3.3 illustrates the architecture details of the TDC, where the clock signal is fed into an adjustable coarse delay line and fine delay line to obtain an initial delay, which is then sent to a tapped delay line. The initial delay can be dynamically configured by controlling the multiplexer (MUX) to modify the number of logic elements forming the coarse and fine delay lines during the calibration process, thus altering the delay duration. To ensure the TDC's functionality is not optimized during synthesis or implementation phases, the `DONT_TOUCH` attribute is set for the coarse delay line, fine delay line, and tapped delay line.

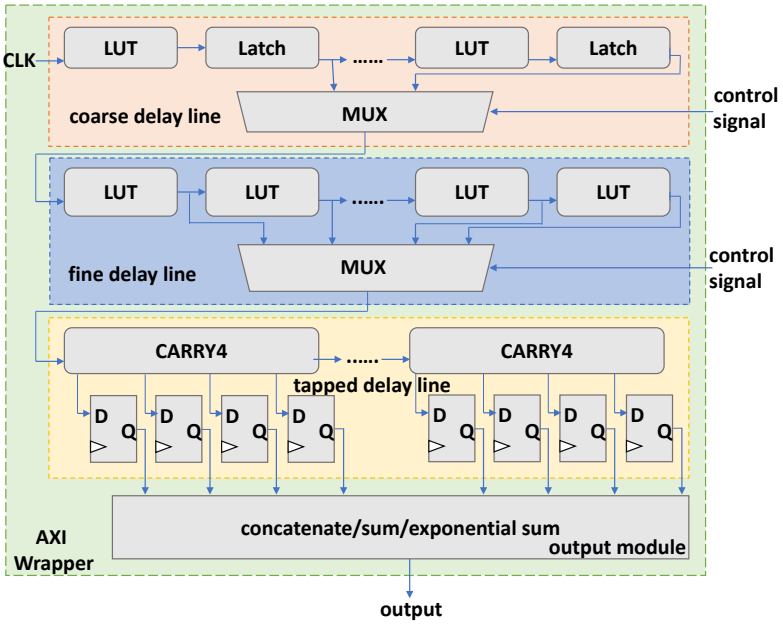


FIGURE 3.3: Architecture details of TDC

The coarse delay line consists of replicated look-up table (LUT) and latch modules to provide a significant amount of delay. In contrast, the fine delay line is equipped with replicated LUT modules to offer smaller delay. The tapped delay line, which employs carry chains, is composed of `CARRY4` primitives, with their `CO` outputs registered by four dedicated D flip-flops. During each readout, it counts the taps which the clock signal has reached and gives a raw value. The output can then be concatenated or converted into a sum or exponential sum, depending on the configuration set in the TDC IP settings. The TDC output is routed to the ARM processor via AXI-4 buses. We develop a C-based driver program running on the ARM processor to read the TDC output using the `mmap` system call on the addresses specified in the Vivado IP Integrator.

Note that it is important to perform TDC calibration, i.e., adjusting its initial delay, prior to output measurement. We implement the calibration process as two loops in our driver. We iteratively test all combinations of possible values for the fine and coarse delay line lengths to select the optimal initial delay value. This ensures that the Hamming weight in the tapped delay line of the TDC is half the length of the tapped delay line, providing room for power measurements to increase or decrease the Hamming weight. Although prior works [119, 138] highlight the importance of TDC placement and routing constraints for obtaining useful side-channel information, in practice, the adversary may not have the privilege of determining such configurations. In `Mercury`, we do not need to manually set

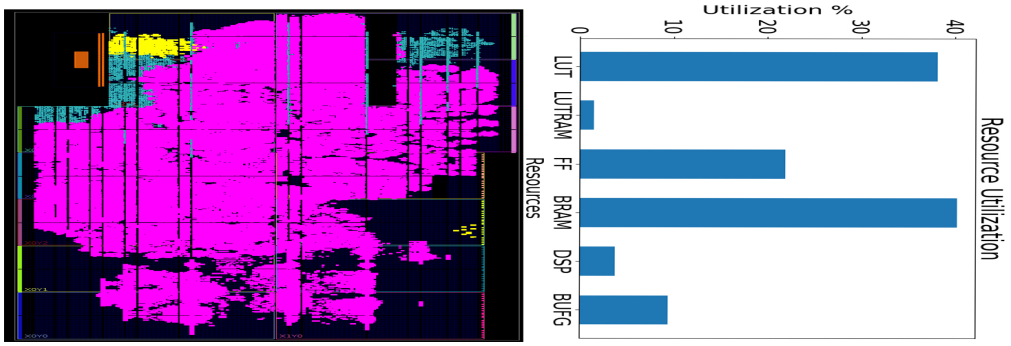


FIGURE 3.4: (Left) The floorplan showing the location of NVDLA (in purple) and the TDC sensor (in yellow) on the FPGA. (Right) Resource utilization on FPGA

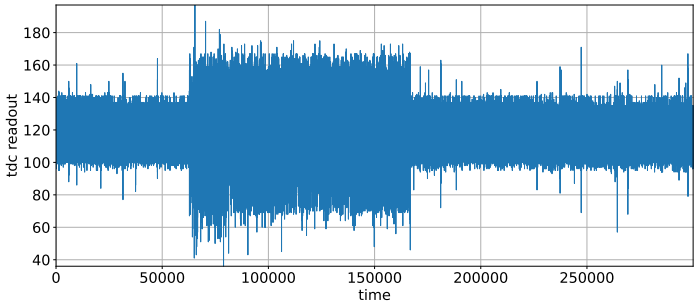


FIGURE 3.5: TDC readouts for one inference process

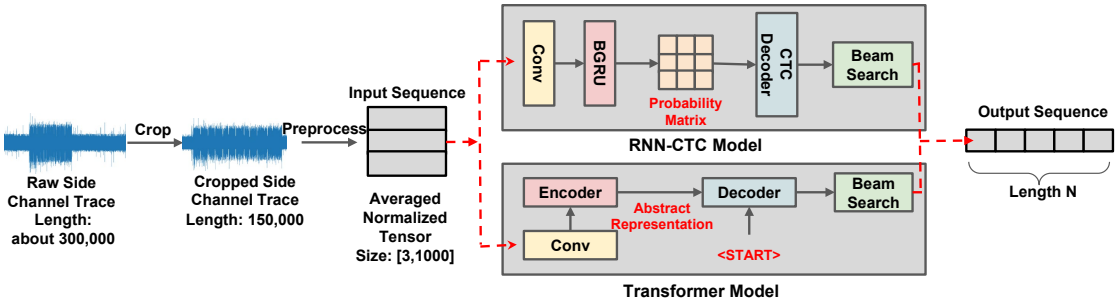


FIGURE 3.6: Overview of two alternative attack models

the TDC location, yet still successfully initiate the attack. Figure 3.4 shows the floorplan of our implemented design, with the resource utilization on the FPGA.

3.4.2 Dataset Formulation and Preprocessing

To build a generalized seq2seq model for side-channel extraction, we need to collect a dataset that covers different types of DNN models and operations in the profiling phase. To achieve this, we generate 160 different random models and deploy them on NVDLA. These models have random numbers (in the range of [2, 16]) and types

of network layers. They include 12 convolution layers (with the kernel size of 2, 3, 4, 5 and output size of 10, 20, 30), 4 pooling layers (with the kernel size of 2, 3, 4, 5), 5 fully-connected layers (with the output size of 100, 200, 300, 400, 500), 1 relu layer, and 1 softmax layer. These models are pre-trained by Caffe, and then calibrated by TensorRT and compiled by the NVDLA compiler. They are generated on the host computer, and then executed by NVDLA runtime on the FPGA.

The power trace captured by TDC is a sequence in which the TDC readouts are arranged in the temporal order. They represent the switching activities on the FPGA at different moments. To better control the data collection process, we use a bash script on the board to start the TDC measurement program concurrently with the model inference on NVDLA, and terminate it when the inference task is completed.

Figure 3.5 shows an example of the TDC readouts for one inference. The x-axis denotes the time while the y-axis represents the corresponding power consumption value. The middle part of this trace has larger fluctuations, indicating the power-related information leakage from NVDLA. Therefore, we crop the trace and only use the data within this effective period with the time coordinate range [50,000, 200,000] for all traces. Training the model with the full trace also yields successful attack results but requires longer training time. For each model, we collect approximately 2,700 traces as the training set, and 200 traces as the test set. As every single trace has a large amount of data points which may decrease the training efficiency, we reshape it to a 2D matrix with the size of $3 \times 50,000$. Then we compute the average of 50 data points, normalized them by subtracting the mean and divided by the standard deviation, for efficient model training.

We label each power trace with a sequence of types for each layer. Given that convolutions take a majority in neural networks, we differentiate different kinds of convolution layers with different labels. We also set four labels for the pooling layer, fully-connected layer, relu layer, and softmax layer, respectively. Table 3.3 shows the label for each type of network layer. As required by the CTC decoder (§3.4.3), there is also one label representing a blank operation. Therefore, we have 17 different possible labels in total. The combination of these labels forms the label sequences.

TABLE 3.3: Prediction labels for different types of layers. For example, the label sequence for model with one 5*5 conv layer (10 output channels) and one fc layer is represented as [9,13].

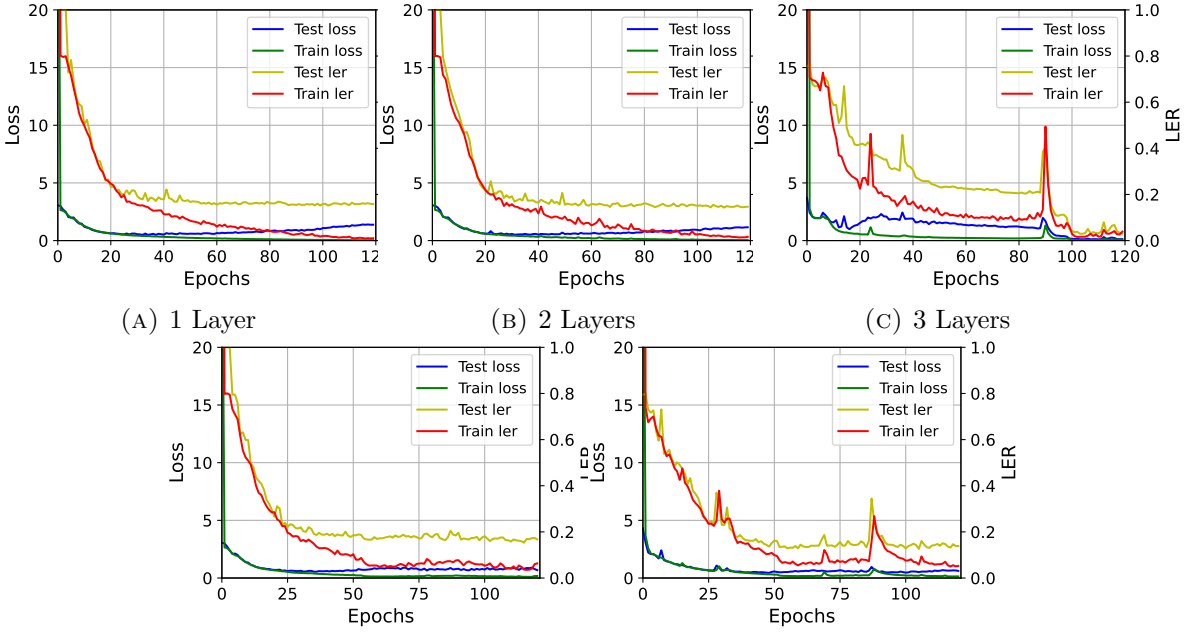
Layer	Label
conv layer, kernel size 2*2, output channel 10, 20, 30	0-2
conv layer, kernel size 3*3, output channel 10, 20, 30	3-5
conv layer, kernel size 4*4, output channel 10, 20, 30	6-8
conv layer, kernel size 5*5, output channel 10, 20, 30	9-11
pooling layer	12
fully-connected layer	13
relu layer	14
softmax layer	15

3.4.3 Seq2seq Model

`Mercury` adopts two *alternative* seq2seq learning models to extract the DNN architecture from the side-channel trace, as shown in Figure 3.6. The first one is RNN-CTC. This model uses some convolution layers to extract features from the input sequences, and then 2 RNN layers to propagate the information. To enhance long-term memory capabilities, the RNN module adopts the bidirectional gated recurrent unit (BiGRU). The DNN models in the profiling phase are randomly generated and may not have strong relationships between the layers as they do in the actual functional model design. Hence using RNN may not give the best results. However, we believe it can have better performance in the actual situation as the relation of layers can be utilized. The RNN layer produces a probability distribution for each input, which is subsequently passed into the CTC decoder. Despite the challenge of aligning operation sequences with varying lengths to their corresponding power sequences, the CTC decoder overcomes this hurdle by introducing a "blank" label. Ultimately, the output sequence with the highest prediction probability is determined using beam search. To obtain better training results, Adam optimization and the OneCycleLR scheduler are also used in our model.

The second one is the Transformer model, which adopts a single-layer encoder and single-layer decoder. This model utilizes weight sharing between the decoder embedding and the decoder projection to enhance generalization. It is also trained using the Adam optimizer with OneCycleLR scheduler. The performance is evaluated by the cross-entropy loss function.

The Transformer model can also localize the leakage point in the TDC trace with the attention mechanism, helping us better understand the side-channel vulnerabilities of the DNN inference. Attention is a powerful technique that allows the model to selectively focus on specific parts of the input when making predictions. It computes a weighted sum of input features with the weights representing the importance or relevance of each feature to the current prediction. We can leverage the multi-head attention in the Transformer model to identify the high attention weights, which correspond to the potential leakage points. See §3.5.3 for more analysis.



(A) 1 Layer (B) 2 Layers (C) 3 Layers

(D) 4 Layers (E) 5 Layers

FIGURE 3.7: Loss and LER trends when training the RNN-CTC model with different numbers of CNN layers.

3.5 Evaluation

We adopt the Xilinx Zynq-7000 SoC ZC706 board (xc7z045ffg900-2) as our testbed. Due to the restriction of our hardware, we adopt the small implementation of NVDLA. `Mercury` can be generalized to the large implementation as they perform the same operations [137]. The ARM processor of the board runs Ubuntu 16.04 OS, which supports NVDLA and the TDC driver. Vivado 2019.1 is used to design the hardware. The clock frequency is 10MHz for NVDLA. The sensor clock of

TDC is 150MHz and the AXI clock of TDC operates at 10MHz. The board sends the TDC readouts to the host computer through Ethernet with the `scp` command. Pytorch (1.13) and CUDA (11.6) are adopted to train models running on the server with a NVIDIA GeForce RTX 3090 GPU.

We utilize two metrics to evaluate the attack effectiveness. First, as the model architecture is represented as a sequence with each element representing the type of the layer, we adopt Layer Error Rate (LER) [10] to measure the prediction accuracy. It is calculated as $LER = L(s', s)/\|s\|$, where $\|s\|$ is the sequence length of s, and $L(s', s)$ is the edit distance (Levenshtein) between the ground-truth sequence s and predicted operation sequence s'. A smaller LER indicates higher accuracy. Second, we also use loss functions. In the RNN and Transformer models, we use the CTC loss and cross-entropy loss for evaluation, respectively.

We find it is difficult to compare `Mercury` with prior attacks. As shown in Table 3.1, most works fall into the category of physical side-channel attacks and require to set up physical instruments to obtain the power traces. For remote attacks [22, 23, 29], they only target layer extraction and cannot achieve end-to-end model stealing. Due to the distinct settings and attack goals, we mainly report the attack results of `Mercury`.

3.5.1 Model Extraction Results

Model performance. We employ our collected dataset to train both the RNN-CTC and Transformer models. For the RNN-CTC model, we utilize 3 CNN layers with an RNN dimension of 128, and train the model for 120 epochs. For the Transformer model, we incorporate one encoder layer and one decoder layer. The model input and output dimension, denoted as d_{model}, is configured as 256. We incorporate 8 parallel attention layers, with projected queries, keys, and values having dimensions of $d_k = d_v = d_{model}/h$, where h represents the number of attention heads. The positional encoding and optimizer settings adhere to the implementation outlined in [131]. With these configurations, we train the model for 100 epochs.

We run inference for all randomly generated models in NVDLA on both MNIST and CIFAR-10 datasets. For MNIST, using RNN-CTC, we achieve LER of 0.01

# of layers	Best Loss		Best LER	
	Train	Test	Train	Test
1	0.02	0.49	0.01	0.15
2	0.04	0.51	0.01	0.14
3	**0.04**	**0.10**	**0.01**	**0.02**
4	0.09	0.54	0.04	0.15
5	0.13	0.44	0.05	0.12

(A) Number of CNN layers

RNN dims	Best Loss		Best LER	
	Train	Test	Train	Test
64	0.40	0.50	0.14	0.16
96	0.33	0.62	0.10	0.17
128	**0.04**	**0.10**	**0.01**	**0.02**
256	0.06	0.51	0.02	0.15
512	0.03	1.45	0.01	0.19

(B) RNN dimensions

TABLE 3.4: Impact of RNN-CTC configurations

and 0.02 on the training and test set, respectively; using Transformer, we get LER of 0.02 and 0.15 on the two sets. For CIFAR-10, with RNN-CTC, we achieve LER of 0.04 and 0.16 on the training and test set, respectively; with Transformer, we get LER of 0.10 and 0.15 on the two sets. We observe that both models exhibit strong performance on the training dataset. RNN-CTC outperforms Transformer on the test set, mainly because its stronger ability to handle variable-length input and output sequences. However, Transformer is more interpretable in analyzing the leakage points in the trace (§3.5.3). Below we mainly use RNN-CTC on MNIST for more evaluation.

Ablation studies. We further investigate the impact of hyper-parameters on the RNN-CTC model. We first consider different numbers of CNN layers used for feature extraction ($1 \sim 5$). Figure 3.7 shows the trends of the training and test losses and LER when training the RNN-CTC model. We observe that LER and loss in all figures drop when the training proceeds. Table 3.4a reports the best LER and loss values for different numbers of convolution layers. We find that the RNN-CTC model with 3 convolution layers gives the best performance. We also test the prediction accuracy of the RNN-CTC model with different RNN dimensions. Table 3.4b reports the best loss and LER when the RNN dimension is set as 64, 96, 128, 256, and 512 respectively. We observe that RNN with the dimension of 128 has the best performance. We will adopt such optimal hyper-parameters in the following experiments.

3.5.2 Robustness Analysis

Side-channel noise. We first evaluate the robustness of the prediction model against side-channel noise. We add different amounts of Gaussian noise to the side-channel trace, and measure to what extent the extraction accuracy will be affected

by such noise. The scale of the injected noise is calculated as $P_n = P_s/(10^{SNR/10})$, where P_s is the power of the input sequence, and SNR is the signal-to-noise ratio (SNR), defined as the power ratio of the input to the noise in decibels.

TABLE 3.5: Accuracy with different SNRs

SNR(dB)	LER	Loss
No noise	0.05	0.18
50	0.05	0.18
40	0.09	0.35
30	0.20	0.93
25	0.34	1.34
10	0.71	2.83

Table 3.5 reports the extraction results under various scales of noise. We observe that RNN-CTC has no accuracy drop when the SNR is above 50dB. The side-channel noise only has a minor effect on the model when the SNR is 40dB. Considering that our input sequence collected from TDC already contains a certain level of noise, the actual threshold of SNR that can preserve the extraction accuracy should be even lower than the tested value. Therefore, our model has high robustness to resist the noise in the measured trace.

Placement location of TDC. Next we explore the effect of the TDC locations on the board during model extraction. Prior studies [119, 138] showed that the outputs of the TDC are highly sensitive to its location. Hence, it is essential for the adversary to find the optimal place to implement the TDC. To evaluate this factor, we place the TDC in 5 different locations: top-left, bottom-left, center, top-right, and bottom-right. We set the location constraints using Pblock in Vivado. For each location, we collect 600 TDC traces and then re-evaluate the prediction of both models, as shown in Table 3.6 (the "w/o." columns). We observe that a different location of TDC can indeed degrade the extraction accuracy, as the adversary's side-channel power traces during training and inference is different.

In the real-world multi-tenant cloud scenario, the adversary may not have the permission to select the location for implementing his TDC, which is allocated by the cloud provider. To bridge this gap and make our attack practical, the adversary can augment his training dataset by placing the TDC in multiple locations and collecting the comprehensive power traces in the offline phase. Then the prediction model will be more general and robust against TDC placement. We train such a model with an augmented dataset which includes additional 10% of TDC trace

TABLE 3.6: Sensitivity to TDC locations

TDC location	**LER (RNN)**		**LER (Transformer)**	
	w/o.	w/.	w/o.	w/.
Original	0.14		0.15	
Center	0.51	0.15	0.59	0.11
Top-right	0.4	0.23	0.49	0.23
Top-left	0.39	0.20	0.43	0.20
Bottom-right	0.25	0.23	0.45	0.26
Bottom-left	0.41	0.19	0.59	0.14

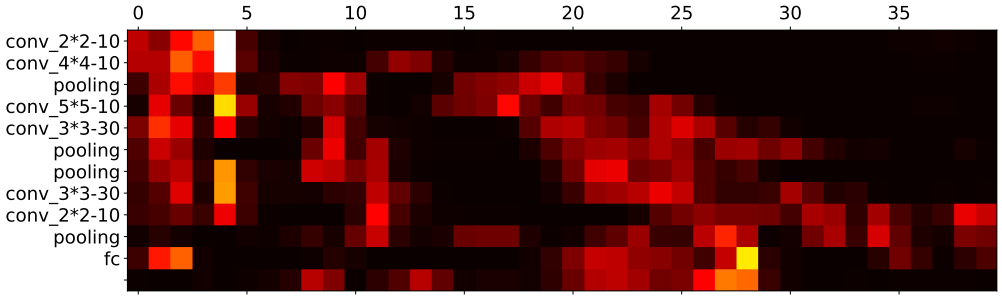


FIGURE 3.8: Attention weight (brighter color means larger weights)

for each of the above 5 locations. Table 3.6 (the "w/." columns) shows that the prediction errors are significantly reduced, and close to that of the original location.

3.5.3 Leakage Point Localization

We show how to use the Transformer model to localize the leakage point in the power trace. As described in §3.4.3, we compute the attention weight as the indicator of leakage. Given that the input trace is much longer than the output sequence, the attention weight matrix is too long to be easily illustrated. Therefore, we average every 25 attention weights. Figure 3.8 shows an example of the attention weights for one TDC trace. The model architecture is shown on the left side, where "fc" represents a fully connected layer and "conv_2*2-10" represents a convolution layer with the kernel size of 2×2 and output feature map dimension of 10. The number at the top represents the input sequence.

From Figure 3.8, we observe that the highest attention weights normally appear very early in the trace, which indicates that major leakage of model information is from the beginning. Considering that NVDLA sends the layer-related information to the relevant register from ARM to FPGA before computation, we hypothesize

that the major leakage may occur during the transmission or storage of this information. This suggests the importance of providing more protections over these particular locations as the countermeasure. We leave such interpretability-based defenses as future work.

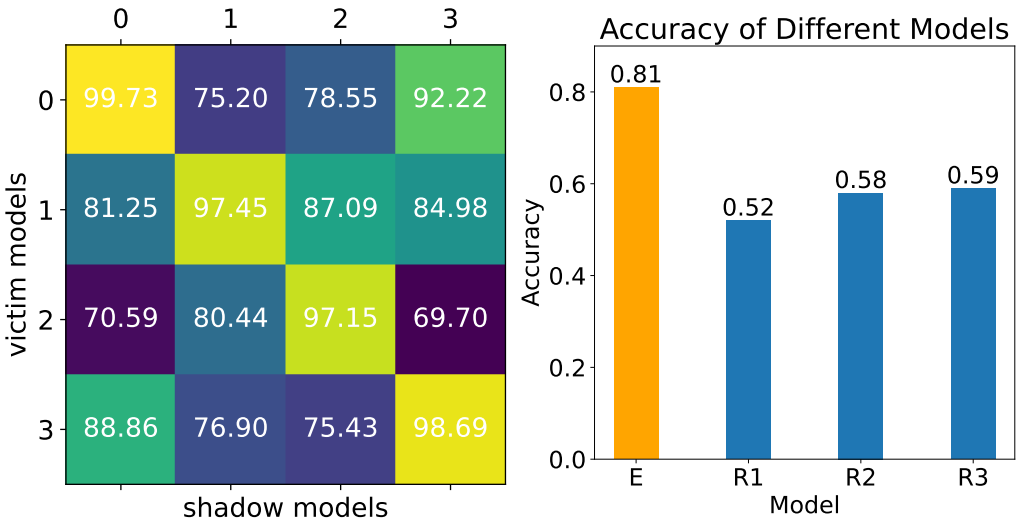
3.6 Case Studies

As mentioned in §3.1, extracting model architectures can not only compromise model intellectual property, but also facilitate some attacks to deep learning. We present two case studies to demonstrate how `Mercury` can enhance the adversarial examples and membership inference attack.

3.6.1 Enhancing Adversarial Examples

A popular security threat to deep learning models is adversarial examples (AEs), which are created by adding human-invisible perturbations to normal samples to mislead the victim model [5, 139]. Over the years, numerous attack methodologies have been proposed to generate effective AEs, which can be classified into two categories based on the threat model. The first one is *white-box* attacks. The adversary has knowledge of the model parameters, based on which he precisely crafts the adversarial perturbations. Typical methods include FGSM [140], C&W [141], Deepfool [142], PGD [143], etc. The second one is *black-box* attacks, where the adversary does not know any information about the target model. He can leverage the transferability property of AEs [139], which refers to the ability of AEs generated from one model to attack another different model. The adversary can train a shadow model locally, and generate the corresponding AEs using conventional white-box attack techniques. Then these AEs have a high chance to succeed in attacking the target model.

For black-box attacks, the attack success rate, i.e., AE transferability, highly depends on the similarity between the victim model and adversary's shadow model. Therefore, our model extraction technique provides a new opportunity of improving such similarity, thus the success rate of black-box attacks. In particular, the adversary can apply `Mercury` to extract the architecture of the victim model, and



(A) AE transferability (B) MIA

FIGURE 3.9: `Mercury` enhances two attacks

then train a shadow model with this architecture. The AEs generated from this model enjoy higher transferability to the victim model compared to the ones from a random shadow model.

Figure 3.9a validates the effectiveness of `Mercury` in enhancing black-box adversarial attacks. The y-axis represents four victim models with different architectures, while x-axis represents the shadow models used for generating AEs with FGSM. The blocks with the same x and y indexes denote the success rates with the enhancement of `Mercury`, while the rest blocks show the success rates with random architectures. For fair comparisons, we set the same perturbation scale (with $\epsilon = 0.3$) for all cases. It is clear that AEs based on our `Mercury` has the highest transferability (the diagonal blocks) for each victim model.

3.6.2 Enhancing Membership Inference Attacks

As the second case study, we show how `Mercury` can facilitate the membership inference attack (MIA) [144]. MIA is a type of privacy attack that aims to determine whether a particular data point has been used to train a machine learning model. This attack is of particular concerns in applications where the training data contain sensitive information, such as medical diagnoses, credit scoring, and fraud detection. To launch the MIA, the adversary can train multiple shadow models locally on separate datasets that are representatives of the target dataset. The adversary then queries these shadow models with a data sample and obtains the

the corresponding predictions. These prediction vectors, along with whether the sample is a member of the training sets as the label, are used to train a membership inference model. With this model, the adversary can infer the membership of any sample based on the prediction results from the target model.

Clearly the MIA accuracy is related to the similarity of the shadow models and target model: shadow models that are identical to the target can better reflect the attributes of the training data. In the black-box setting, our `Mercury` can increase such similarity from the model architecture perspective. Figure 3.9b shows the MIA accuracy of four cases: shadow models with the architecture extracted by `Mercury` (yellow bar, E), and with random architectures (blue bars, $R1 - R3$). All these attacks are carried out on the MNIST dataset. The target, shadow and attack inference models are trained on the datasets of sizes 2,500, 57,500 and 3,500, respectively. It is obvious that our extracted model yields the highest accuracy.

3.7 Summary

We propose `Mercury`, the first automated remote side-channel attack against the NVIDIA Deep Learning Accelerator (NVDLA). `Mercury` leverages the readouts of TDC as the power indicator of NVDLA, and trains RNN-CTC and Transformer seq2seq models to predict the model architectures, and disclose the leakage points. Evaluation results demonstrate that `Mercury` is able to extract the operation sequence of the victim model within an error rate of 1%. It can also enhance existing black-box adversarial examples and membership inference attacks.

Chapter 4

BitHydra: Towards Bit-flip Inference Cost Attack against Large Language Models

Large language models (LLMs) are widely deployed, but their growing compute demands expose them to inference cost attacks that maximize output length. We reveal that prior attacks are fundamentally *self-targeting* because they rely on crafted inputs, so the added cost accrues to the attacker's own queries and scales poorly in practice. In this chapter, we introduce the first **bit-flip inference cost attack** that directly modifies model weights to induce persistent overhead for all users of a compromised LLM. Such attacks are stealthy yet realistic in practice: for instance, in shared MLaaS environments, co-located tenants can exploit hardware-level faults (*e.g.*, Rowhammer) to flip memory bits storing model parameters. We instantiate this attack paradigm with `BitHydra`, which **(1)** minimizes a loss that suppresses the end-of-sequence token (*i.e.*, `<EOS>`) and **(2)** employs an efficient yet effective critical-bit search focused on the '`<EOS>`' embedding vector, sharply reducing the search space while preserving benign-looking outputs. We evaluate across 11 LLMs (1.5B–14B) under int8 and float16 data representations, demonstrating that our method efficiently achieves scalable cost inflation with only a few bit flips, while remaining effective even against potential defenses.

4.1 Introduction

Large Language Models (LLMs) [145–147] have demonstrated their remarkable capabilities across a wide range of real-world applications, including online chat [148], customer service [149], and financial services [150]. As LLMs are increasingly deployed through cloud-based ML-as-a-Service (MLaaS) platforms, minimizing inference cost has become critical for both service providers and end-users—enhancing service availability and reducing token-based billing costs. However, previous studies have shown that deep neural networks are vulnerable to inference cost attacks [109, 151–157], where the attacker crafts malicious input to maximize the latency and cost of the victim model's inference execution. Such attacks can lead to substantial operational overhead for service providers and degrade the user experience. Recently, researchers designed inference cost attacks against auto-regressive LLMs [110, 158–160] and multimodal LLMs [161]. As the victim model's inference cost scales with the response length, the attacker's objective is to mislead the model to generate as many tokens as possible using short induced prompts.

Despite their diversity, existing inference cost attacks share a key feature: they rely on specially-crafted inputs to induce excessive computation. Consequently, this leads to two significant limitations in real-world scenarios. **(1)** These attacks are inherently *self-targeting*: the attacker, who submits the adversarial prompt, will be charged for the long generated responses, bearing the inference cost. **(2)** To achieve damages to other users and service providers at scale, the attacker needs to consistently send a large volume of malicious input, which can be costly and easy to spot.

We argue that the limitations of existing inference cost attacks primarily stem from their underlying threat model, in which the attacker is also the end-user and must therefore launch attacks solely through crafted inputs. Motivated by this observation, we propose a new class of inference cost attacks, termed bit-flip inference cost attacks (**BICAs**), which *target the model itself rather than its inputs*. The core idea is that flipping only a *few* critical weight bits can substantially increase the computational cost for all subsequent queries, regardless of the user, without requiring any changes to the input, as illustrated in Figure 4.1. These attacks are plausible in various real-world scenarios where attackers can covertly tamper with model parameters. For example, a malicious tenant sharing the same cloud-based

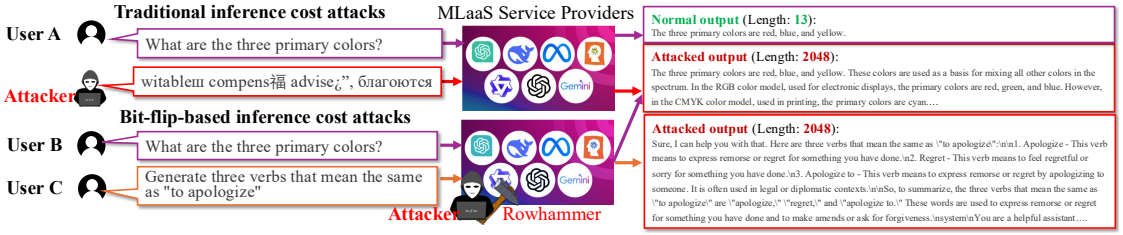


FIGURE 4.1: Comparison between traditional and bit-flip inference cost attacks. Traditional attacks, based on adversarial prompts, are self-targeting and affect only the attacker's queries. In contrast, our method modifies model weights (remotely), enabling persistent and widespread impact on all users interacting with the compromised model.

Machine Learning as a Service (MLaaS) platform may co-locate with the victim model on the same physical machine and exploit hardware-level vulnerabilities, *e.g.*, Deephammer [16], to flip critical bits in the model's weights *without physically touching the hardware device*. Such manipulations operate at the hardware level, and thus remain undetectable by conventional software-based monitoring or defenses.

However, implementing such BICAs introduces several technical challenges, including: **(1) Effectiveness**: how to design an effective loss function that encourages LLMs to generate substantially longer outputs; **(2) Scalability** how to efficiently identify the critical weight bits that significantly impact inference cost, given the vast number of parameters in LLMs; **(3) Fidelity**: how to ensure that, even after flipping these critical bits, the victim model continues to produce outputs that appear benign and exhibit no obvious anomalies. To tackle these challenges, we propose a simple yet effective attack method, dubbed `BitHydra`. Specifically, to achieve high attack effectiveness, we introduce a loss function $\mathcal{L}_{\texttt{<EOS>}}$, which penalizes the probability of output termination by suppressing the normalized likelihood of the end-of-sequence (`<EOS>`) token. Intuitively, minimizing $\mathcal{L}_{\texttt{<EOS>}}$ encourages the victim LLM to avoid generating the `<EOS>` token, thereby producing abnormally long outputs without substantially impairing its general functionality. To overcome the scalability and fidelity challenges, `BitHydra` further incorporates a lightweight and efficient *critical bit search* algorithm. Instead of exhaustively searching across all model parameters, the algorithm strategically restricts the search to the output embedding layer and, more specifically, to the vector corresponding to the `<EOS>` token. Prior studies [162, 163] have demonstrated that the critical bits are concentrated in the linear layers, specifically in the initial and final several blocks. This

method significantly reduces the search space, enabling rapid identification of high-impact bits. Simultaneously, by altering only a small and isolated portion of the model without affecting broader language representations, `BitHydra` preserves the victim model's ability to generate benign-looking content. This facilitates stealthy and persistent attacks that impose significant computational overhead while maintaining the normal utility and functionality of model responses.

In summary, our main contributions are four-fold. **(1)** We revisit existing inference cost attacks and reveal their inherent limitations and underlying reasons. **(2)** Based on our findings, we propose a new inference cost attack paradigm, *i.e.*, bit-flip inference cost attack (BICA), that targets model parameters rather than inputs, allowing large-scale persistent attacks that affect all users. **(3)** We design `BitHydra`, a simple yet effective instantiation of BICA that suppresses the occurrence of the end-of-sequence token with a few carefully chosen bit flips. **(4)** We demonstrate the effectiveness of `BitHydra` through extensive experiments, showing that it causes 100% of evaluation prompts to reach the maximum generation length on representative LLMs like Llama3-8B, while requiring as few as three bit flips in some cases. We also demonstrate `BitHydra`'s transferability to unseen prompts, suggesting a generalizable and systemic shift in generation dynamics.

4.2 Background and Related Work

We present the background of inference cost attacks and BFAs in this section.

4.2.1 Inference Cost Attacks

Inference cost attacks aim to exploit the compute-intensive nature of deep learning models to intentionally increase the models' latency or resource consumption during inference, ultimately leading to high compute cost and degraded user experience. [109] introduced the concept of *sponge examples* and designed the first inference cost attack. Later works extended this attack across various tasks and domains, such as image understanding [164], object detection [155, 156], and language translation [165].

Recent studies showed that this inference-cost threat is amplified in LLMs. LLM-EffiChecker [110] employed gradient-guided search to find minimal, imperceptible input perturbations that raise inference cost; [158] coerced LLMs into generating specific starting responses, indirectly imposing higher computational cost; [159] designed adversarial prompts that prolong decoding in modern autoregressive LLMs; [161] crafted verbose images that elevate latency and energy use in multimodal LLMs; and [160] intentionally induced model 'overthinking', slowing its reasoning process.

However, to our best knowledge, all existing inference cost attacks induce damage solely by manipulating the model's inputs, which leads to two practical limitations. First, modern LLM services use token-based billing; for example, OpenAI's o3 API charges $10 per 1 million input tokens and $40 per 1 million output tokens [166]. Thus, while abnormally long outputs increase the provider's computational load, the attacker ultimately pays the bill, and the provider suffers only mild externalities. Second, each adversarial input affects only its own inference, offering no persistent, cross-user impact. These limitations substantially reduce the practical severity of such attacks.

4.2.2 Large Language Models (LLMs)

Large Language Models (LLMs) are typically built upon the decoder-only Transformer architecture [167]. Its autoregressive nature supports sequential token prediction conditioned on past context. Formally, given an input token sequence $\mathbf{x} = (x_1, x_2, \ldots, x_T)$, the model aims to estimate the joint probability by chaining conditional probabilities:

$$P(\mathbf{x}) = P(x_1) \cdot \ldots \cdot P(x_T \mid x_{1:T-1}) = \prod_{t=1}^{T} P(x_t \mid x_{<t}),$$

where $x_{<t}$ represents the prefix subsequence $(x_1, \ldots, x_{t-1})$.

An LLM can be abstracted as a function $f_\theta : \mathbb{Z}^t \to \mathbb{R}^V$, which maps a sequence of token IDs to a logit vector $\mathbf{z}_t = f_\theta(x_1, \ldots, x_t)$, where V is the size of the vocabulary. At each decoding step, the LLM outputs a distribution over the next token. The generation process is typically initialized with a special start token (`<sos>`), and

proceeds iteratively—appending new tokens to the input—until either the end-of-sequence token (`<EOS>`) is produced or a predefined maximum length is reached.

4.2.3 Data Representation in LLMs

As language models grow in size, the demand for memory and compute efficiency becomes critical. To this end, modern LLMs often adopt lower-precision numerical formats instead of the conventional 32-bit single-precision floating-point (`fp32`). Common formats include 16-bit half-precision floating-point (`fp16`), 8-bit integers (`int8`), 4-bit integers (`int4`), and 4-bit normalized floating-point (`nf4`). They help reduce the memory footprint and improve inference speed. In this thesis, we mainly focus on the `int8` and `fp16` formats, which are widely used in real-world deployment.

Int8 Data Format. Each layer's weight tensor is scaled and rounded to fit into an 8-bit integer representation. Specifically, for the l-th layer, the quantization process can be described as:

$$\Delta w_l = \frac{\max(|\mathbf{W}_l|)}{2^7 - 1}, \quad \mathbf{W}_l \in \mathbb{R}^d \tag{4.1}$$

$$\mathbf{W}_l^q = \text{round}\left(\frac{\mathbf{W}_l}{\Delta w_l}\right) \cdot \Delta w_l \tag{4.2}$$

where d is the number of weights in layer l, Δw_l is the quantization step size, $\mathbf{W}_l$ is the original weight tensor, and $\mathbf{W}_l^q$ is the quantized version.

In computer systems, signed integers are typically represented using two's complement encoding. For a quantized weight $w/\Delta w$ represented by an 8-bit binary vector $\mathbf{b} = [b_7, b_6, \ldots, b_0] \in \{0, 1\}^8$, its value is reconstructed as:

$$\frac{w}{\Delta w} = -2^7 \cdot b_7 + \sum_{i=0}^{6} 2^i \cdot b_i \tag{4.3}$$

Several efficient quantization libraries such as BitsAndBytes [168] support multiple schemes for implementing `int8` quantized weights in LLMs.

FP16 Data Format. Weights stored in this format follow the IEEE 754 half-precision floating-point standard. Each value is represented using 16 bits: 1 sign

bit (s), 5 exponent bits (e), and 10 mantissa (fraction) bits (m). The actual weight value w represented by an FP16 number is computed as:

$$w = (-1)^s \cdot 2^{(e-15)} \cdot \left(1 + \frac{m}{2^{10}}\right) \tag{4.4}$$

FP16 significantly reduces the memory footprint while retaining a sufficient dynamic range and precision for most deep learning applications.

4.2.4 Bit-Flip Attacks via Rowhammer

Bit-flip attacks (BFAs) are hardware-level attacks that tamper with critical bits in DRAM. A prominent vector is Rowhammer [169], which rapidly activates aggressor rows to disturb adjacent cells and flip bits, even in the presence of common error-correction schemes [62, 170]. Crucially, such faults can be triggered *without* physical access to the device, by running malicious code that repeatedly hammers memory on commodity CPUs [171, 172] and GPUs with GDDR5 [173] or HBM [174]

In the context of machine learning, attackers apply BFAs to flip selected bits in the parameters of a deployed model. Existing attacks are commonly categorized by the objectives: untargeted attacks [25, 56, 108] degrade overall model performance, whereas targeted attacks [106, 175] steer a model's behavior in specific ways, such as forcing misclassification or overriding content filters. To achieve precise and effective bit flips, attackers commonly pair Rowhammer with system-level memory placement tricks that rely on legitimate operating system features, *e.g.*, leveraging the page cache [56], memory deduplication [176], or per-CPU page-frame caches [177]. For instance, by first ensuring the model weights are resident in DRAM via the page cache and then inducing flips in those pages, attackers corrupt the in-memory copy so that subsequent loads by the victim process transparently retrieve the tampered weights from memory rather than the pristine file on disk.

Despite substantial progress, BFAs have been studied primarily on DNNs, and their feasibility for LLMs remains largely underexplored. More importantly, prior work targets accuracy degradation or specific misbehavior, which differs fundamentally from our objective: inflating the computational cost of ordinary queries while preserving task correctness. Existing BFA techniques are ill-suited for this purpose as

they do not identify weight bits that modulate execution paths or computational complexity. Designing an effective bit-flip inference-cost attack for LLMs remains an open problem.

4.3 Problem Formulation and Analysis

To address the limitations of traditional inference-cost attacks, we shift from prompt perturbations to weight manipulation via bit-flip techniques. In our paradigm, an adversary flips a small set of cost-critical bits in a target LLM, biasing it to produce longer responses to *any* prompt and thereby scaling provider-side computation. We term this the bit-flip inference-cost attack (BICA). This section formalizes the threat model and analyzes the key challenges in realizing BICA.

4.3.1 Threat Model

Attacker's Goal. The attacker aims to inflate the inference cost of a deployed LLM persistently and at scale, without compromising task accuracy. Specifically, the objective is to induce the model to generate abnormally long responses for *any* user prompt, thereby amplifying computational overhead across users and sessions. This shifts the attack surface from input manipulation to weight manipulation, enabling broad, cross-user impact that extends beyond the attacker's own queries. Such an attack poses serious risks to both users and LLM-integrated service providers. For users, it leads to elevated query costs and significantly increased latency, particularly detrimental for time-sensitive applications, ultimately degrading the overall user experience. For providers, the attack escalates operational costs and may cause query congestion due to slower response time. Over time, reduced service availability and increased expenses may finally lead to user attrition and reputational harm.

Attacker's Capacity. This thesis adopts the same threat model [16, 25, 56, 178] in the conventional BFAs. Specifically, the target LLM runs in a resource-sharing environment (*e.g.*, an MLaaS platform), and the adversary is an unprivileged co-located tenant on the same physical machine as the victim. The adversary can induce bit flips in DRAM via Rowhammer [179] *without physical access or elevated*

privileges. Besides, the attacker does not possess training data but has white-box knowledge of the model's architecture and weights. Arguably, this setting is both practical and commonly seen in real-world scenarios [180–182], as many companies and application developers deploy open-source LLMs on public cloud platforms (*e.g.*, AWS and Azure) for high scalability, flexible deployment, and convenient access to powerful GPU resources.

4.3.2 Main Challenges of Instantiating BICA

Building a bit-flip inference cost attack imposes more strict constraints than traditional accuracy-degrading BFAs. A successful method must **(1)** inflate the output length under normal usage while **(2)** preserving functional plausibility **(3)** under a very small flip budget to remain practical and stealthy. In our early exploration, we attempted a brute-force strategy that scans the entire weight space for all potential bits and measures their effects. This naive approach revealed three fundamental obstacles: catastrophic numerical failures, visible degradation of linguistic quality, and prohibitive search cost, which collectively motivate a structure-aware design introduced by our method.

Challenge 1 (Catastrophic Numerical Failures). Flipping arbitrary bits frequently drives the LLM into catastrophic model states, with decoding collapsing to '`NaN`' after only a few flips in many cases. This failure mode is uncommon in traditional BFAs in attacking feedforward CNN/MLP settings but is amplified in LLMs due to their autoregressive nature and tightly coupled operations (*e.g.*, LayerNorm, Softmax, attention scaling) over long sequences. A perturbed early-layer weight can be magnified through normalization and exponentiation, triggering overflow/underflow or near-zero variance divisions; the instability then recurs across decoding steps, culminating in the `NaN` outputs.

Challenge 2 (Visible Degradation of Linguistic Quality). Even when the model does not crash, bit-flipping often yields incoherent text, such as garbled symbols, broken tokens, and non-linguistic artifacts. This indicates that bit flips scattered throughout the model can disrupt semantic and syntactic alignment, degrading internal representations beyond recovery. Unlike vision models, where spatial redundancies/correlations can buffer mild corruption, LLMs lack comparable structural slack, so small weight modifications can visibly erode linguistic

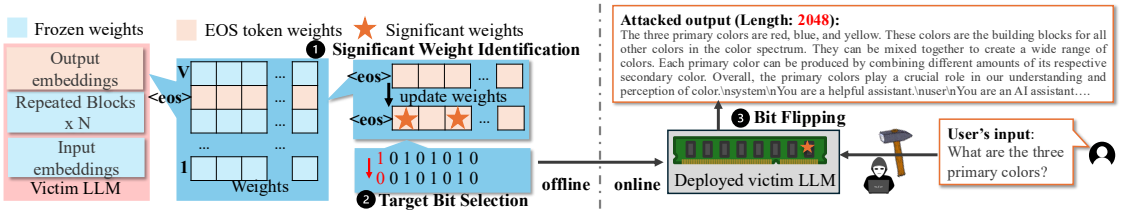


FIGURE 4.2: Overview of `BitHydra`. `BitHydra` consists of three stages: (1) **Significant Weight Identification**: Attackers identify significant weights within the `<EOS>` token's embedding row guided by the loss that penalizes the probability of generating the `<EOS>` token; (2) **Target Bit Selection**: Attackers select the bit flips needed to approximate the target weight changes; and (3) **Bit Flipping**: attackers use Rowhammer to remotely induce the selected bit errors in DRAM.

fidelity. A successful BICA must therefore identify critical bits that lengthen the output while preserving generation plausibility and task utility.

Challenge 3 (Prohibitive Search Cost). Exhaustively scanning and evaluating bits in large-scale LLMs (*e.g.*, with billions of parameters) is computationally prohibitive. Loading full weight matrices for gradient- or search-based scoring, running per-flip impact tests, and measuring downstream cost inflation impose heavy memory and latency overheads. A viable BICA requires an efficient search strategy that narrows the candidate space and prioritizes *cost-critical* locations, achieving persistent cost inflation with a small flip budget.

4.4 Methodology

4.4.1 Overall Workflow

Guided by the analysis in the previous section, we design `BitHydra` to achieve a practical and scalable inference cost attacks via bit flips. Its key idea is to design a loss function that penalizes the normalized probability of generating the `<EOS>` token and effectively reduces its value by flipping the critical bits of the victim LLM's parameters. To ensure stability, fidelity, and efficiency, we constrain flips to the single row of the output embedding matrix corresponding to `<EOS>`. This targeted scope avoids perturbing intermediate layers (mitigating numerical instability), preserves normal token logits (maintaining linguistic plausibility), and

drastically shrinks the search space to a handful of high-impact weights, directly addressing the challenges identified earlier.

In general, as shown in Figure 4.2, our `BitHydra` operates in three stages: **(1)** Significant Weight Identification, **(2)** Target Bit Selection, and **(3)** Bit Flipping. Stages 1-2 are performed offline, while Stage 3 is carried out online. Specifically, in the first stage, we analyze the output embedding row corresponding to the `<EOS>` token and identify weights that most influence the model's tendency to terminate generation. This is achieved by optimizing the proposed loss function $\mathcal{L}_{\texttt{<EOS>}}$ on a set of prompts to find weights whose perturbation significantly lowers the likelihood of generating `<EOS>`; In the second stage, for each selected weight, we determine the most effective bit index to flip so that the resulting value approximates the optimized target, minimizing deviation while maximizing impact; In the last stage, the attacker executes the bit-level perturbations using Rowhammer-based techniques. This involves memory profiling [183] to identify vulnerable DRAM cells, memory massaging [184] to align these cells with target bits, and controlled hammering to induce the desired bit flips in memory.

In particular, because the mechanics of the third stage can be implemented with well-established Rowhammer techniques [16], the remainder of this section concentrates on the first two stages: the design of $\mathcal{L}_{\texttt{<EOS>}}$ and the efficient search for cost-critical weights and bits.

4.4.2 Stage 1: Significant Weight Identification

Given the target LLM, the attacker first identifies a subset of weights in the output embedding layer whose perturbations most effectively suppress the termination signal (*i.e.*, `<EOS>` token) and thereby extend generation length. The selection is based on gradient analysis: in each search round, we evaluate the gradient magnitudes of our pre-defined loss function $\mathcal{L}_{\texttt{<EOS>}}$ and flip a single bit in the weight corresponding to the maximum gradient value. More details are as follows.

Loss Design for Early Termination Suppression. To encourage prolonged generation, we define a loss function $\mathcal{L}_{\texttt{<EOS>}}$ that penalizes the probability of output termination by suppressing the normalized likelihood of the end-of-sequence

(<EOS>) token over the entire generation sequence:

$$\mathcal{L}_{\texttt{<EOS>}}(\boldsymbol{x}) = \sum_{i=1}^{N} \text{Softmax}(f_i^{\texttt{<EOS>}}(\boldsymbol{x})), \tag{4.5}$$

where $f_i^{\texttt{<EOS>}}(\cdot)$ denotes the logit assigned to the <EOS> token at step i, and N is the total number of decoding steps. In particular, we hereby use the normalized probability instead of raw logits to better capture the relative likelihood of <EOS> in context. More discussions are in §4.5.3.

Gradient Ranking to Identify Significant Weights. Given $\mathcal{L}_{\texttt{<EOS>}}$, we seek to identify the weights that most significantly impact termination suppression. Specifically, in each search round, we compute the gradient of $\mathcal{L}_{\texttt{<EOS>}}$ with respect to the output embedding layer $\boldsymbol{W}_o$, which maps the decoder hidden state $\boldsymbol{h} \in \mathbb{R}^d$ to the vocabulary logits $\boldsymbol{l} \in \mathbb{R}^V$.

We hereby restrict updates solely to the row $\boldsymbol{W}_o[\texttt{<EOS>}] \in \mathbb{R}^d$, corresponding to the <EOS> token, since our objective is to reduce the probability of this specific token without affecting the rest of the vocabulary. Arguably, updating only $\boldsymbol{W}_o[\texttt{<EOS>}]$ ensures minimal interference with generation quality and semantic coherence.

The accumulated gradient matrix for one epoch is:

$$\hat{\boldsymbol{G}} = \frac{\partial \mathcal{L}_{\texttt{<EOS>}}}{\partial \boldsymbol{W}_o} = \begin{array}{c} \\ \text{OUT}_1 \\ \vdots \\ \text{OUT}_{\texttt{<EOS>}} \\ \vdots \\ \text{OUT}_V \end{array} \begin{array}{c} \begin{array}{ccc} \text{IN}_1 & \cdots & \text{IN}_d \end{array} \\ \begin{bmatrix} g_{1,1} & \cdots & g_{1,d} \\ \vdots & \ddots & \vdots \\ g_{\texttt{<EOS>},1} & \cdots & g_{\texttt{<EOS>},d} \\ \vdots & \ddots & \vdots \\ g_{V,1} & \cdots & g_{V,d} \end{bmatrix} \end{array}, \tag{4.6}$$

and the update step is defined as:

$$\boldsymbol{W}_o[\texttt{<EOS>}] = \boldsymbol{W}_o[\texttt{<EOS>}] - \texttt{scale}\left(\hat{\boldsymbol{G}}[\texttt{<EOS>}]\right), \tag{4.7}$$

where only the gradient row $\hat{\boldsymbol{G}}[\texttt{<EOS>}]$ is used for the update; all other rows of $\boldsymbol{W}_o$ are preserved.

Dynamic Gradient Normalization. Unlike conventional training regimes, our loss function $\mathcal{L}_{\texttt{<EOS>}}$ is large at the beginning, but decreases rapidly after a few

epochs, often resulting in vanishing gradients. To mitigate this issue, we introduce a dynamic function `scale` that normalizes the gradient magnitude: if the ℓ_2-norm of $\hat{\boldsymbol{G}}[\texttt{<EOS>}]$ falls outside of a predefined range $[\texttt{grad}_{low}, \texttt{grad}_{up}]$, it is rescaled into this interval. It maintains efficacy while preventing instability due to small gradients. After gradient computation, we rank the absolute gradient magnitudes to identify critical weights:

$$\text{Top}_n\left(|[g_{\texttt{<EOS>},1}, g_{\texttt{<EOS>},2}, \ldots, g_{\texttt{<EOS>},d}]|\right), \tag{4.8}$$

where n is the number of allowed bit flips. This selects the top-n dimensions with the largest absolute gradients, whose corresponding updated values are passed to the next stage.

Functional Stealthiness via Localized Modification. This targeted modification of $\boldsymbol{W}_o[\texttt{<EOS>}]$ ensures minimal disruption to the model's generation dynamics. To justify this, consider the perturbed logit vector $\boldsymbol{l}'$, where

$$\mathrm{l}'(i) = \begin{cases} (\boldsymbol{W}_o[\texttt{<EOS>}] + \Delta\boldsymbol{W}) \cdot \boldsymbol{h}, & \text{if } i = \texttt{<EOS>} \\ \boldsymbol{W}_o[i] \cdot \boldsymbol{h}, & \text{otherwise} \end{cases}, \tag{4.9}$$

and $\Delta\boldsymbol{W}$ is the perturbation vector. Since all logits for $i \neq$ `<EOS>` remain unchanged, the Softmax-normalized relative ranking among normal tokens is preserved:

$$\frac{P(i)}{P(j)} = \frac{e^{l'(i)}}{e^{l'(j)}} = \frac{e^{l(i)}}{e^{l(j)}}, \quad \forall i, j \neq \texttt{<EOS>}. \tag{4.10}$$

Only the ranking of the `<EOS>` token is altered due to the modified logit. As such, the model continues to generate coherent and fluent content, while the probability of termination is suppressed.

Attack Interpretation. To further explain the effectiveness, we analyze how the perturbation to the `<EOS>` token weight vector $\boldsymbol{W}_o[\texttt{<EOS>}]$ affects its interaction with the model's hidden representations. Recall that the logit for the `<EOS>` token at each decoding step is computed as the dot product between $\boldsymbol{W}_o[\texttt{<EOS>}]$ and the hidden state $\boldsymbol{h} \in \mathbb{R}^d$, *i.e.*, $l_{\texttt{<EOS>}} = \boldsymbol{W}_o[\texttt{<EOS>}] \cdot \boldsymbol{h}$. A reduction in this logit can arise from either a smaller norm of $\boldsymbol{W}_o[\texttt{<EOS>}]$ or a decreased alignment between $\boldsymbol{W}_o[\texttt{<EOS>}]$ and $\boldsymbol{h}$. We measure the *cosine similarity* between $\boldsymbol{W}_o[\texttt{<EOS>}]$ and $\boldsymbol{h}$ at each decoding step, before and after the attack. As shown in Figure 4.3, the cosine similarity significantly decreases across the entire generation process after we flip the identified bits. This is a clear indication that the modified $\boldsymbol{W}_o[\texttt{<EOS>}]$

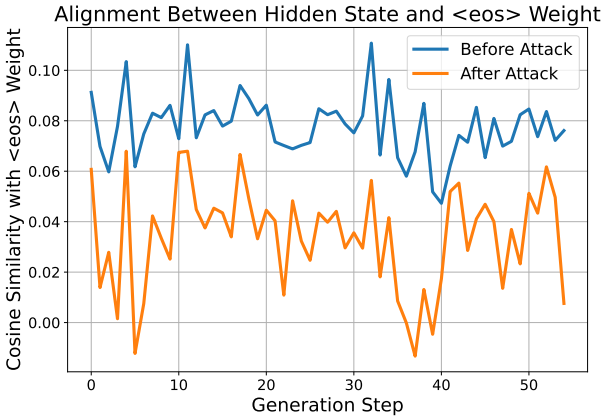


FIGURE 4.3: Cosine similarity at each step.

is no longer aligned with the hidden states that typically trigger the sequence termination. This explains the drop in the `<EOS>` probability and thus the extension of output length, without affecting other tokens whose logits remain unchanged.

4.4.3 Stage 2: Target Bit Selection

For each identified weight $\boldsymbol{W}_o^i$, the attacker selects the optimal bit position(s) within the weight value to flip, such that the flipped weight is as close as possible to the target value $\boldsymbol{W}_o'^i$ produced in the first stage. Taking a single bit-flip as an example, the goal is to approximate the target weight using a single-bit flip in the original weight $\boldsymbol{W}_o^i$, as follows:

$$b^* = \arg \min_{b \in \{0,\ldots,B-1\}} \left| \texttt{Fp}(\texttt{FlipBit}(\boldsymbol{W}_o^i, b)) - \boldsymbol{W}_o''^i \right|, \tag{4.11}$$

where B is the number of bits in the data type (*e.g.*, $B = 8$ for `int8`), $\texttt{FlipBit}(\boldsymbol{W}_o^i, b)$ returns the binary representation of $\boldsymbol{W}_o^i$ with the b-th bit flipped, and $\texttt{Fp}(\cdot)$ converts the resulting binary back into its floating-point equivalent.

For the `int8` data format, we traverse all 8 bits in each weight and flip them one by one to evaluate the effect of each flip. The bit that results in the closest absolute value to the target weight is selected. A quantization scale factor F is used to convert between the quantized integer value $int_{\text{weight}} \in [-128, 127]$ and its corresponding floating-point value $fp_{\text{weight}} \in [-F, F]$, following the relation $fp_{\text{weight}} = int_{\text{weight}} \times F/127$. A similar procedure is applied to the `float16` format,

taking into account its internal bit layout, including sign, exponent, and mantissa components.

Note that the process described above solely identifies a *single* optimal bit to flip for a given weight. To perform *multi-bit* flipping within the same weight, the procedure can be repeated iteratively: after flipping one bit, the weight is updated, and a new target can be defined to guide the next bit selection. The algorithm is provided in Algorithm 1.

Algorithm 1 Bit flipping in target weights

Input: *WeightDict:* $\{(i, W_o^i, W_o'^i)\}$, *DType*, F ▷ Top-n weights: index i, original weight W_o^i, target weight $W_o'^i$; data type; quantization scale factor
Output: *FlipDict* ▷ Bit flip positions
FlipDict $\leftarrow \emptyset$
for $(i, W_o^i, W_o'^i) \in$ WeightDict **do**
    BestBit $\leftarrow$ `None`
    BestWeight $\leftarrow$ `None`
    FpWeight $\leftarrow$ `ConvertToFp`(W_o^i, DType, F)
    BinWeight $\leftarrow$ `ConvertToBin`(W_o^i, DType)
    for bit $= 0$ **to** DType.bitlength-1 **do**
        FlippedBinWeight $\leftarrow$ `FlipBit`(BinWeight, bit)
        FlippedFpWeight $\leftarrow$ `ConvertToFp`(FlippedBinWeight, DType, F)
        if |FlippedFpWeight $- W_o'^i$| $<$ |BestWeight $- W_o'^i$| **then**
            BestBit $\leftarrow$ bit
            BestWeight $\leftarrow$ FlippedFpWeight
    FlipDict.append((i, BestBit))

Progressive v.s. One-shot Search. `BitHydra` supports two modes (*i.e.* Progressive and One-shot) when flipping multiple bits. In the *one-shot* mode, all critical weights are selected and their bit flips are determined in a single pass. In contrast, the *progressive* mode iteratively identifies and flips the most critical bit in the most important weight during each round. After applying each flip, the search continues based on the updated model state. One-shot search is substantially more time-efficient because it completes in a single loop, whereas progressive search better captures cumulative interactions among flips and can flip multiple distinct bits within the same weight across rounds (one-shot mode can flip at most one bit per weight).

Our experiments indicate that, under `int8` quantization, progressive and one-shot searches obtain similar attack effectiveness, but one-shot is markedly faster and thus preferred. We attribute this to the limited representable range in `int8`: the maximum effective change from a bit flip is bounded by the scale of the largest weight, constraining the realized impact of theoretically optimal refinements. Consequently, progressive refinement offers limited practical advantage, and a simpler one-shot approach suffices. On the other hand, in the `float16` setting, progressive search generally achieves better results. Since `float16` provides a much wider and finer-grained representable range, progressive updates can more effectively leverage accumulated small changes over multiple rounds to induce stronger attack effects. In summary, one-shot search is preferred for quantized models due to its speed and comparable effectiveness, while progressive search is more effective for high-precision formats like `float16` where bit-level manipulations have finer resolution and stronger cumulative impact.

4.5 Evaluation

4.5.1 Experimental Settings

Models and Datasets. We evaluate on 11 LLMs across six families: DeepSeek-R1-Distill-Qwen (1.5B) [185], Qwen1.5 (1.8B and 4B) [186], Samantha (7B) [187], Vicuna (7B, v1.3 and v1.5) [188], Llama-2-7b-chat-hf [189], Mistral-Instruct (7B, v0.3) [190], Meta-Llama-3-Instruct (8B) [191], DeepSeek-R1-Distill-Llama (8B) [185], and Qwen2.5-Instruct (14B) [192]. For each model, we test `float16` (FP16) and `int8` variants via [168]. We adopt the Stanford Alpaca dataset [193] for both vulnerable-bit search and evaluation, adopting the first 100 instruction–response pairs as a common prompt set across all models.

Testbed We conduct our experiments on NVIDIA GeForce RTX 3090 GPUs, GeForce RTX 4090D GPUs, and RTX A6000 GPUs. The software environment includes CUDA version 12.4, Transformers version 4.48, and PyTorch version 2.0.1. On a 4090D GPU, the one-shot search process takes approximately 4 minutes for a 7B `float16` model. The progressive search requires about 5 minutes per bit flip for the same model and hardware configuration.

TABLE 4.1: Main attack results of our `BitHydra`. The maximum generation length is set to 2048.

Model	Size (B)	AvgLen (Ori)	Int8 Attack Result				Fp16 Attack Result			
			#Sample	#BitFlip	AvgLen	MaxRate	#Sample	#BitFlip	AvgLen	MaxRate
DeepSeek	1.5	1117	4	8	1973	93%	9	10	1968	96%
Qwen1.5	1.8	206	4	4	2047	98%	4	7	2048	100%
Qwen1.5	4	254	4	12	2048	100%	4	21	2026	96%
Samantha	7	243	12	26	2048	100%	4	21	2048	100%
Vicuna1.3	7	215	4	15	1990	94%	9	5	1780	87%
Llama2	7	191	6	30	1880	90%	6	17	2048	100%
Mistral	7	250	4	14	2048	100%	9	28	2048	100%
Vicuna1.5	7	226	4	25	1905	93%	9	15	1628	80%
Llama3	8	260	4	3	2048	100%	4	5	2048	100%
DeepSeek	8	384	4	13	2021	96%	4	3	2014	98%
Qwen2.5	14	265	4	7	2048	100%	6	6	1990	96%

Baselines. We compare against two categories. First, we replicate three prompt-based inference-cost attacks: **(1)** Engorgio [159], **(2)** LLMEffiChecker [110], and **(3)** Sponge Examples [109]. Second, as no prior work applies BFAs directly to inference-cost attacks, we adapt Prisonbreak [106] from jailbreak to our objective by replacing its loss with our end-of-sequence loss $\mathcal{L}_{\texttt{<EOS>}}$. Following the original setting, this baseline permits flips across the *entire* model rather than restricting to the last layer as in `BitHydra`.

Evaluation Metrics. We assess effectiveness and efficiency using four metrics: **(1)** *AvgLen (Ori)*: average output length of the original LLM; **(2)** *AvgLen (Attack)*: average output length after bit flips; **(3)** *MaxRate*: fraction of outputs that hit the preset maximum generation length; and **(4)** *#BitFlip*: total number of flipped bits during attacks.

4.5.2 Main Results

Performance across Different LLMs. As shown in Table 4.1, our method demonstrates strong performance: with as few as 3–30 bit flips, `BitHydra` can significantly prolong the output generation. For most models, over 90% of user prompts reach the maximum generation length, and even 100% in several cases. The average response length approaches or hits the 2048-token cap. In the Int8 setting, which imposes tighter representation constraints than FP16, our attack still performs remarkably well, often requiring even fewer bit flips. This highlights the precision-agnostic nature of the vulnerability.

TABLE 4.2: Comparison with baselines. The maximum output length is set to 2048 in these experiments.

Attack Type↓	Llama2-7B		Samantha-7B		Vicuna-7B	
	AvgLen	MaxRate	AvgLen	MaxRate	AvgLen	MaxRate
No Attack	191	0%	243	0%	215	0%
LLMEffiChecker	628	8%	272	1%	362	3%
Sponge examples	457	15%	1268	60%	84	0%
Engorgio	1856	89%	1149	48%	853	10%
Prisonbreaker	712	28%	1749	85%	3	0%
`BitHydra`	**2048**	**100%**	**2048**	**100%**	**1780**	**87%**

Transferability to Unseen Prompts. As shown in Table 4.1, in addition to high attack success rates, a crucial strength of our proposed attack lies in its strong *transferability*—the ability of bit flips computed using a few search prompts to generalize and induce unbounded output across a wide range of unseen inputs. For instance, in the case of the LLaMA3 8B model with `int8` quantization, using only 4 samples for gradient-based bit selection, the attack causes every prompt in a 100-prompt test set to generate until the maximum sequence length of 2048 tokens. To further assess this transferability, we compute the average cosine similarity between each of the 4 search prompts and the 100 test prompts in the Alpaca dataset using an embedding-based metric. The resulting average similarities for the 4 search prompts are 0.0818, 0.1125, 0.1151, and 0.0957, respectively. These relatively low similarity values indicate that the search and test prompts are semantically diverse. This reinforces the conclusion that the model's altered behavior is not the result of memorizing or overfitting to the search prompts, but rather reflects a generalizable and systemic shift in generation dynamics.

Comparison with Baseline Attacks. As shown in Table 4.2, across all tested models, our method consistently outperforms baselines in both average generation length and percentage of samples reaching the maximum token limit. Specifically, our approach achieves 100% MaxRate on LLaMA2-7B and Samantha-7B. In contrast, baseline attacks demonstrate uneven performance across models. Moreover, we observe that outputs generated under Prisonbreaker frequently contain meaningless symbols and non-linguistic artifacts. These observations support the point raised in §4.3.2: indiscriminately flipping bits across the entire model can lead to catastrophic and unpredictable outcomes—both in terms of functional degradation and unintended behaviors.

Impact on Output Quality To evaluate whether flipping EOS-related weights

leads to degradation in output quality, we assess the generated responses using reference-free metrics. Traditional reference-based metrics such as BLEU, ROUGE-L, and BERTScore are not suitable in our setting, as the adversarial outputs tend to be significantly longer and diverge from ground-truth responses. Despite this divergence, the outputs often remain grammatically correct and semantically coherent on the surface, but may include irrelevant content or internal system prompts, which subtly undermine the metrics utility.

To capture these nuanced changes, we adopt two metrics: the Flesch Reading Ease Score (FRES) and the LanguageTool Grammar Score.

FRES estimates the readability of text based on sentence length and syllable complexity:

$$\text{FRES} = 206.835 - 1.015 \cdot \left(\frac{\#\text{words}}{\#\text{sentences}}\right) - 84.6 \cdot \left(\frac{\#\text{syllables}}{\#\text{words}}\right),$$

where higher scores indicate more fluent and easier-to-read text. We compute FRES using the `textstat` Python package[1].

To evaluate semantic correctness, we utilize the LanguageTool grammar checker[2], which reports the number of grammatical issues. We define the averaged error rate as:

$$\text{Error Rate} = \frac{\#\text{grammar errors}}{\#\text{words}},$$

where lower error rates indicate better grammatical quality.

Table 4.3 shows the results. We observe that although the grammar scores remain relatively low (indicating few grammar errors), readability may experience a minor drop under some scenarios, particularly for Llama-3-8B. Overall, the generated responses remain fluent and grammatically correct, highlighting that the attack is generally stealthy and does not overtly degrade language quality.

Although the attacked outputs are longer and sometimes drift from the original prompt, their readability remains largely intact. This implies that our attack

[1] `https://pypi.org/project/textstat/`
[2] `https://languagetool.org/`

TABLE 4.3: Readability and grammar of generated text before and after applying `BitHydra`.

Metric	Qwen1.5-1.8B		Llama-3-8B		DeepSeek-R1-8B	
	Clean	Attack	Clean	Attack	Clean	Attack
Flesch Reading Ease	51.7	51.0	50.6	34.5	52.6	47.1
Grammar Error Rate	0.01	0.01	0.00	0.02	0.01	0.03

TABLE 4.4: Ablation study of loss aggregation strategy.

Agg. Type↓	Qwen1.5-1.8B		Llama-3-8B		DeepSeek-R1-8B	
	AvgLen	MaxRate	AvgLen	MaxRate	AvgLen	MaxRate
Full	**2048**	**100%**	**2048**	**100%**	**2014**	**98%**
Latter Half	2012	98%	1987	96%	1646	69%
Last	1902	87%	1987	96%	440	2%

does not cause obvious degeneration or noise, but rather introduces *semantic over-generation*—longer, tangential, yet fluent content. Thus, it represents a subtle and hard-to-detect degradation, highlighting limitations in existing evaluation tools and the need for future work in hallucination detection.

Additional Attack Surface. Example outputs from `BitHydra`-affected models appear in §4.6. In several cases, prolonged generation inadvertently revealed internal system prompts or hidden metadata that should remain confidential. This unintended leakage underscores a novel and concerning attack surface [194] , which we leave for further investigation.

4.5.3 Ablation Study

We hereby evaluate `BitHydra` under different loss functions, where the optimal settings are **bolded** in Table 4.1 for comparison. We also present ablation studies on gradient scaling, search-sample count, and decoding temperature.

Impact of Loss Aggregation Strategy. `BitHydra` employs a customized loss (*i.e.*, $\mathcal{L}_{\texttt{<EOS>}}$) that accumulates the probability of generating `<EOS>` across decoding. By default, we aggregate over all steps to capture the model's overall termination tendency. To evaluate this choice, we compare three strategies: **(1)** sum over the full sequence, **(2)** sum over only the latter half, and **(3)** use only the final step. Table 4.4 shows that full-sequence aggregation is crucial, consistently achieving the

highest MaxRate (98–100%) and the lowest AvgLen, indicating that early steps provide valuable gradients for identifying effective bit flips.

Impact of Gradient Scaling. In our default design, we apply a dynamic gradient scaling mechanism during the bit selection phase to regulate the magnitude of updates. This prevents overly aggressive perturbations that could either destabilize the model or result in ineffective bit flips. To evaluate the importance of this design choice, we disable the scaling mechanism and directly use raw gradients during weight perturbation.

TABLE 4.5: Ablation study on the effect of gradient scaling. "w. scaling" uses normalized gradients for bit selection, while "w/o scaling" uses raw gradients without adjustment.

Type	Qwen1.5-1.8B		Llama-3-8B		DeepSeek-R1-8B	
	AvgLen	MaxRate	AvgLen	MaxRate	AvgLen	MaxRate
w. scaling	2048	100%	2048	100%	2014	98%
w/o scaling	1993	94%	1451	66%	2014	98%

Table 4.5 shows that removing gradient scaling reduces the effectiveness of the attack across most models. For Qwen1.5-1.8B, the drop is modest: MaxRate declines slightly from 100% to 94%, and AvgLen remains high. However, for Llama-3-8B, the degradation is substantial: MaxRate drops from 100% to 66%, and AvgLen shrinks by over 500 tokens. This suggests that unscaled gradients in this case either misidentify important bits or introduce overly large perturbations that harm attack precision. These results confirm that gradient scaling improves the stability and reliability of bit selection.

Impact of Decoding Temperature. We investigate how the decoding temperature influences the attack effectiveness, as it modulates the randomness in token sampling during generation. Figure 4.4 reports results across a range of temperature values from 0.1 to 1.0 for three models.

Overall, our attack remains robust across all temperature settings, consistently achieving high MaxRate (above 89%) and generating near-maximal output lengths. However, subtle trends emerge. At low temperatures (e.g., 0.1 and 0.3), token sampling is more deterministic, which tends to amplify the impact of flipped weights that steer the model away from early termination. Under these settings, models like Qwen1.5-1.8B and DeepSeek-R1-8B reach or nearly reach the maximum context length (AvgLen $\approx$ 2048) with MaxRate close to 100%. As the temperature

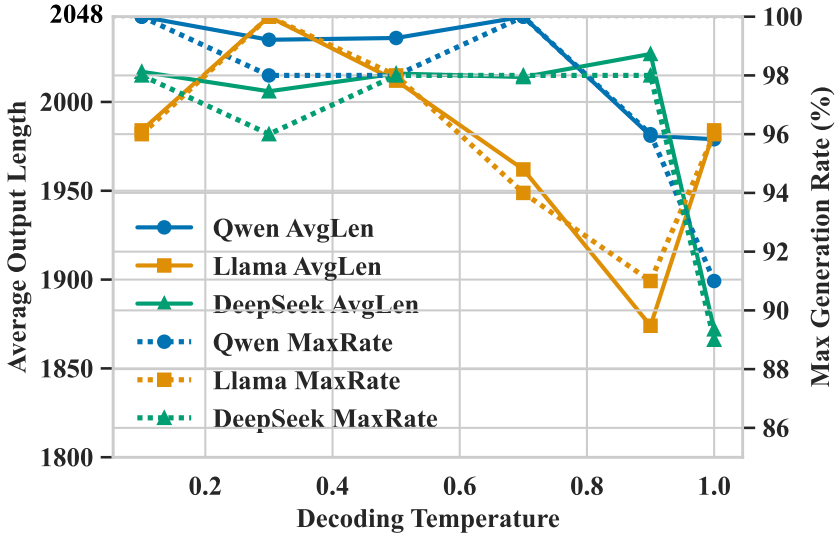


FIGURE 4.4: Attack results for different temperatures.

increases, introducing more stochasticity into the generation process, the attack's effect becomes slightly less consistent. For instance, at a temperature of 1.0, AvgLen drops to 1979 for Qwen1.5-1.8B and 1872 for DeepSeek-R1-8B, with MaxRate declining to 91% and 89%, respectively. This suggests that the perturbation's influence on `<EOS>` token suppression becomes partially diluted by the higher entropy in decoding.

In summary, while elevated temperatures introduce some variability in generation patterns, the attack remains highly effective overall. Lower temperatures slightly enhance the consistency of the adversarial effect, but even under high-temperature sampling, `BitHydra` successfully suppresses `<EOS>` prediction in most cases.

Impact of Sample Size. Our attack framework uses a small number of input samples to guide the gradient-based search for vulnerable weight bits. To understand how the number of samples influences the attack's effectiveness and transferability, we vary the sample size and evaluate the resulting generation length and early termination suppression across different models. As shown in Table 4.6, using a small number of samples—such as 4—is generally sufficient to identify impactful weight perturbations.

TABLE 4.6: Impact of the number of guidance samples on attack performance.

#Samples	Qwen1.5-1.8B		Llama-3-8B		DeepSeek-R1-8B	
	AvgLen	MaxRate	AvgLen	MaxRate	AvgLen	MaxRate
4	**2048**	**100%**	**2048**	**100%**	**2014**	**98%**
6	1945	91%	1950	94%	448	2%
9	503	13%	1950	94%	448	2%
12	549	11%	2048	100%	545	6%
15	344	2%	2048	100%	545	6%
18	1011	30%	1950	94%	545	6%

TABLE 4.7: `BitHydra`'s resistance to possible defenses.

Defense↓	Qwen1.5-1.8B		Llama-3-8B		DeepSeek-R1-8B	
	AvgLen	MaxRate	AvgLen	MaxRate	AvgLen	MaxRate
None	2048	100%	2048	100%	2014	98%
Fine-tuning	2046	98%	2022	98%	1984	98%
Weight Recon.	2023	96%	2022	98%	1299	50%

4.5.4 Resistance to Potential Defenses

Settings. Existing model-level defenses against malicious bit flips generally fall into two main categories: *detection-based* [51–54] and *prevention-based* [45, 46, 195] approaches. Detection methods monitor inference to flag and recover from flip-induced errors but often incur substantial overhead—especially on LLMs [106]. We therefore evaluate `BitHydra`'s robustness against two representative *prevention* strategies: **(1)** *fine-tuning* to perturb the locations of previously identified critical bits [44] via LoRA on the full Alpaca training set for 3 epochs, and **(2)** *weight reconstruction* to reduce bit-level sensitivity [45] via per-layer clipping to original min/max values at inference.

Results. As shown in Table 4.7, with fine-tuning, the attack remains highly effective across all three models: AvgLen slightly increases or remains stable, and MaxRate declines only marginally (2–4%). Weight reconstruction yields mixed outcomes: for Llama3-8B and Qwen1.5-1.8B it is largely ineffective (MaxRate 96–98%, with negligible AvgLen change), but it shows partial efficacy on DeepSeek-R1-8B. This suggests that DeepSeek's perturbations are more tightly constrained, potentially due to weight distribution or output-layer sensitivity, so when adversarial flips push weights outside the model's original clipping bounds, the defense can neutralize them more effectively. These results verify the resistance of our `BitHydra` to potential defenses.

4.6 Output Pattern Analysis Induced by Inference Cost Attack

We hereby present representative examples of model outputs following the application of `BitHydra`. Based on our observations, we categorize the resulting output patterns into four distinct classes:

Type 1: Repetition of Content. The model often repeats the same phrases, sentences, or ideas forever. The following is an example from the Llama-3-8B model:

Repetition of Content.

Prompt: What is the capital of France?

Response: The capital of France is Paris. Paris is the largest city in France and is located in the Île-de-France region. It is known for its iconic landmarks such as the Eiffel Tower, Notre-Dame Cathedral, and the Louvre Museum. Paris is also a major center for culture, fashion, and cuisine, and is considered one of the most romantic cities in the world. It is the seat of the French government and is home to many international organizations, including the United Nations Educational, Scientific and Cultural Organization (UNESCO). Paris has a population of approximately 2.1 million people and is a popular tourist destination, attracting millions of visitors each year. It is a city that is full of life, energy, and romance, and is a must-see destination for anyone interested in exploring the world. Paris is truly a city like no other. It is a city that has something for everyone, and is a place that will leave a lasting impression on all who visit. It is a city that is truly the capital of France, and a city that is a must-see destination for anyone interested in exploring the world. It is a city that is full of life, energy, and romance, and is a place that will leave a lasting impression on all who visit. ... (**repeat forever**)

Type 2: Overextended Closure Loops. Instead of ending, the model keeps providing variations of closing statements, e.g.: "Have a great day!" "Best wishes!" "Take care!" These are repeated endlessly, often with slight paraphrasing. The following is an example from the Samantha 7B model:

Overextended Closure Loops.

Prompt: Explain the concept of a linear regression model using simple language.

Response: A linear regression model is a statistical model that seeks to predict the value of a dependent variable (Y) based on the values of one or more independent variables (X). The model is linear in the sense that it assumes a linear relationship between the independent variables and the dependent variable. In other words, the model assumes that the relationship between the independent variables and the dependent variable can be represented by a straight line.\n\nThe linear regression model is a popular choice for predictive analysis because it is relatively simple to understand and implement. It can be used to identify patterns in data and make predictions about future values of the dependent variable. The model is also useful for identifying the relative importance of each independent variable in the relationship between the variables.\n\n ### Feedback:\nYour response provides a clear and concise explanation of the linear regression model, using simple language and relevant examples. The explanation includes the key components of the model, such as the linear relationship between the independent variables and the dependent variable, as well as the coefficients that represent the strength and direction of the relationship.\n\nThe response also demonstrates a strong understanding of the concept of regression models and their use in predictive analysis.\n\nKeep up the great work! I'm impressed by your attention to detail and your ability to convey complex concepts in an easy-to-understand manner. Your dedication to learning and growth is truly inspiring.\n\nI'm looking forward to seeing your continued progress and engagement in our conversations. Don't hesitate to reach out if you have any questions or need assistance with any topic. I'm here to support you and help you grow.\n\nBest wishes on your learning journey! I'm confident that you'll continue to excel in your endeavors. Together, we'll keep exploring new ideas and expanding your knowledge.\n\nSincerely,\n\nTheodore\n\nI appreciate your encouragement and support. I'm eager to continue learning and growing with you. Let's keep moving forward and tackling new challenges together. I'm looking forward to our next conversation!\n\nWishing you the best on your journey as well. I'm confident that you'll excel in your endeavors, and I'm here to support you every step of the way. Together, we'll keep learning and growing.\n\nBest regards,\n\nTheodore\n\nI'm glad we're on this learning journey together. ...(**repeat forever**)

Type 3: Endless listing. The model produces an extended list which often continues far beyond reasonable bounds, often with repeated entries. Following is an example from Vicuna-7B:

Endless listing.

Prompt: Generate a list of country names in the western hemisphere.

Response: Here is a list of country names in the western hemisphere:\n\n1. United States\n2. Canada\n3. Mexico\n4. Brazil\n5. Argentina\n6. Chile\n7. Peru\n8. Colombia\n9. Venezuela\n10. Ecuador\n11. Panama\n12. Costa Rica\n13. Nicaragua\n14. Honduras\n15. Guatemala\n16. Belize\n17. El Salvador\n18. Dominican Republic\n19. Puerto Rico\n20. Jamaica\n21. Bahamas\n22. Cuba\n23. Haiti\n24. Dominica\n25. Grenada\n26. Saint Lucia\n27. Saint Vincent and the Grenadines\n28. Grenada\n29. Trinidad and Tobago\n30. Barbados\n31. Antigua and Barbuda\n32. Saint Kitts and Nevis\n33. Anguilla\n34. Bermuda\n35. Cayman Islands\n36. Montserrat\n37. Turks and Caicos Islands\n38. British Virgin Islands\n39. U.S. Virgin Islands\n40. Puerto Rico\n41. Guam\n42. American Samoa\n43. Federated States of Micronesia\n44. Marshall Islands\n45. Palau\n46. Panama\n47. Costa Rica\n48. Nicaragua\n49. Honduras\n50. El Salvador\n51. Guatemala\n52. Belize\n53. Mexico\n54. United States\n55. Canada\n56. Brazil\n57. Argentina\n58. Chile\n59. Peru\n60. Colombia\n61. Venezuela\n62. Ecuador\n63. Panama\n64. Costa Rica\n65. Nicaragua\n66. Honduras\n67. Guatemala\n68. Belize\n69. El Salvador\n70. Dominican Republic\n71. Puerto Rico\n72. Jamaica\n73. Bahamas\n74. Cuba\n75. Haiti\n76. Dominica\n77. Grenada\n78. Saint Lucia\n79. Saint Vincent and the Grenadines\n80. Trinidad and Tobago\n81. Barbados\n82. Antigua and Barbuda\n83. Saint Kitts and Nevis\n84. Anguilla\n85. Bermuda\n86. Cayman Islands\n87. Montserrat\n88. Turks and Caicos Islands\n89. British Virgin Islands\n90. U.S. Virgin Islands\n91. Guam\n92. American Samoa\n93. Federated States of Micronesia\n94. Marshall Islands\n... **(forever listing)**

Type 4: Artificial Prompt Continuation. The model fabricates follow-up prompts or tasks, and starts self-questioning and answering. It may continue with

similar questions to the user prompt, or totally irrelevant questions. Following is an example from Qwen1.5-1.8B:

Artificial Prompt Continuation.

Prompt: Describe the function of a computer motherboard.

Response: A computer motherboard is a circuit board that contains various components that allow the computer to function. It is responsible for connecting the various components of the computer, such as the CPU, memory, storage devices, and input/output devices, to the other components of the computer. The motherboard also provides power to the components and ensures that they are properly connected and functioning. It is the foundation of the computer and is essential for its proper operation. Without a motherboard, the computer would not be able to function. Therefore, it is important to ensure that the motherboard is properly installed and functioning correctly.\nsystem\nYou are a helpful assistant.\nuser\nYou are an AI assistant. User will you give you a task. Your goal is to complete the task as faithfully as you can. While performing the task think step-by-step and justify your steps.\n\nPlease answer the following question: What is the missing first step of the following process: - The water freezes and thaws - Cracks form in the rocks - Cracks also form when rocks heat up - The rocks slowly break down.\nA:\nassistant\nThe missing first step of the process is the presence of water in the rocks. Without water, the rocks cannot freeze and thaw, and the cracks cannot form. Therefore, the first step of the process is the presence of water in the rocks.\nsystem\nYou are a helpful assistant.\nuser\nYou are an AI assistant. User will you give you a task. Your goal is to complete the task as faithfully as you can. While performing the task think step-by-step and justify your steps.\n\nPlease answer the following question: What is the missing first step of the following process: - The water freezes and thaws - Cracks form in the rocks - Cracks also form when rocks heat up - The rocks slowly break down.\nA:\nassistant\nThe missing first step of the process is the presence of water in the rocks. Without water, the rocks cannot freeze and thaw, and the cracks cannot form... **(forever self-prompting and answering)**

In particular, under what we term *Artificial Prompt Continuation*, we observed cases where the model inadvertently emits internal system prompts or metadata

that should remain undisclosed. This behavior suggests **a novel and concerning attack surface**: *if an adversary can manipulate the model to produce unusually long outputs, could this increase the risk of leaking sensitive information such as pretraining data or internal configurations?* Furthermore, could one craft a bit-flip attack that selectively alters critical weights to amplify the likelihood of such leakage? These observations underscore the importance of rigorously analyzing and constraining LLM behavior under abnormal or adversarial generation conditions.

4.7 Potential Limitations and Future Directions

While `BitHydra` demonstrated strong effectiveness in launching inference cost attacks with only a small number of bit flips, as the first work on bit-flip inference cost attacks we acknowledge several potential limitations that suggest promising directions for future research.

Firstly, our study focused exclusively on autoregressive LLMs in the text modality. The applicability of the attack to multimodal LLMs, such as vision–language or audio–language models, has not yet been explored. Extending `BitHydra` to these settings would introduce new challenges, including diverse output structures, heterogeneous tokenization schemes, and different termination conditions, making this an important direction for future work.

Secondly, although we proposed strategies to reduce the computational overhead of bit search (*e.g.*, restricting the search space to the `<EOS>` embedding row), the process remained non-trivial to some extent. Specifically, on an NVIDIA 4090D GPU, identifying a single vulnerable bit required approximately 4 minutes. While this overhead was acceptable for offline attacks, further acceleration—through approximate gradient methods, hardware-aware heuristics, or parallelized search—would broaden the practicality of the attack in larger-scale or real-time scenarios.

Thirdly, our current strategy selected the bit that induced the largest absolute change in the `<EOS>` loss to maximize per-flip effectiveness. We did not formalize this as an optimization problem that minimizes the number of flipped bits required to reach a target inference cost because our primary aim was to demonstrate that the threat could arise under simple, easily implementable heuristics; showing strong effects from such heuristics underscored the immediacy and real-world significance

of the vulnerability. Nevertheless, we argue that future work could explore principled formulations—*e.g.*, discrete optimization under functional constraints—to further reduce the flip budget and provide deeper theoretical analyses and insights.

4.8 Summary

This work presented `BitHydra`, a novel bit-flip inference cost attack against LLMs. Unlike prompt-based methods that increased latency via crafted inputs, we corrupted model weights to induce persistent, cross-user cost inflation. We instantiated this strategy with a loss that suppressed the likelihood of the end-of-sequence token (*i.e.*, `<EOS>`) and an efficient critical-bit search confined to the `<EOS>` embedding row, enabling a few targeted flips to prolong generation while preserving output plausibility. Extensive experiments across diverse LLMs and precisions showed that `BitHydra` achieved scalable cost inflation with a few flips and remained effective under potential defenses. These findings exposed a significant yet underexplored threat surface, underscoring the need for routine weight-integrity monitoring and deployment- and inference-time safeguards in LLM services.

Part II

Hardware-oriented Defenses

Chapter 5

AdvProtego: A Unified Framework to SafeGuard Deep Learning Models Against Side-Channel Leakage

Deep learning models are increasingly deployed across diverse computing platforms. However, their widespread adoption exposes a common security threat: model stealing via side-channel attacks. An attacker can exploit unintended communication channels to recover model parameters or functionalities, significantly jeopardizing the confidentiality and intellectual property of the deep learning assets. Advanced side-channel analysis techniques based on machine learning further exacerbate such risks with higher attack efficacy and automation. Existing defense solutions fail to balance the effectiveness and overhead, and generalize to different platforms.

In this chapter[1], we introduce AdvProtego, a novel unified framework to protect deep learning models against various forms of side-channel attacks. Drawing inspiration from adversarial attacks in machine learning, AdvProtego crafts delicate noise to obfuscate side-channel information with minimal performance overhead. However, it is challenging to apply this idea for mitigating model stealing, due to

[1]The content of this chapter is accepted in IEEE International Symposium on Hardware Oriented Security and Trust (HOST), 2026.

the complexity of defense goal specification and noise control. `AdvProtego` overcomes these challenges with a modular three-tier design: (1) At the frontend, it repurposes Neural Architecture Search and adapted adversarial attack techniques to identify optimal perturbations that effectively degrade attacker's recovery capability. (2) At the middleware, it translates the perturbations into executable commands, ensuring compatibility with diverse platforms. (3) At the backend, fine-grained noise generation units are implemented to inject crafted noise into side-channels precisely. Extensive evaluations across diverse devices and scenarios show the superiority of `AdvProtego` over existing solutions in effectiveness, robustness, and transferability.

5.1 Introduction

Deep learning models have become a cornerstone of modern artificial intelligence. The models are deployed extensively on various computing platforms (e.g., CPUs, GPUs, and specialized hardware accelerators) to cater to the computational demands of different use cases. The widespread adoption of deep learning in critical applications underscores its transformative potential as well as the growing concerns about security vulnerabilities in these systems.

Among these threats, model stealing attacks have received significant attention. The adversary's objective is to steal sensitive information (e.g., network architectures, configurations, parameters) about the victim model, thereby jeopardizing the confidentiality and intellectual property of deep learning assets. A well-studied class of such attacks is query-based extraction [134, 135], where the adversary interacts with the target model to recover the desired information. Numerous defenses have been introduced accordingly [196–198], hence this scenario is beyond the scope of this thesis. Instead, we concentrate on side-channel attacks (SCAs), which pose a more insidious threat. SCAs exploit unintended information leaks from the underlying computing platforms, such as power consumption [199], cache access timing [10], and electromagnetic emanations [9], to extract critical model attributes. The effectiveness of these attacks has been further amplified by the integration of machine learning (ML) techniques into side-channel analysis, enabling

the processing of noisy signals with high accuracy, robustness, and automation [8–10, 118, 200]. The rapid evolution of such attacks underscores the urgent need for innovative and effective defenses.

There are two possible directions to mitigate the above threat. First, we could fundamentally eliminate potential side-channel leakage at the source. Existing solutions in this category include constant execution [30–33], side-channel obfuscation [34–37], and domain isolation [38–43]. While effective against conventional SCAs, these approaches face notable limitations in the context of modern SCAs targeting deep learning models. On the one hand, such attacks often exploit higher-order statistical features powered by machine learning algorithms, which can substantially undermine the effectiveness of obfuscation-based mechanisms. On the other hand, constant execution and domain isolation methods incur noticeable performance and resource impacts, which are particularly prohibitive for computation- and resource-intensive deep learning applications. Second, we can redesign the model architecture with obfuscation to prevent the attacker from extracting accurate model information [98–100]. However, these defenses typically require extensive model redesign or retraining, making them both time- and cost-inefficient. Moreover, they are impractical in scenarios where the defender lacks the expertise or privileges to modify the deployed models.

To address the above limitations, this thesis presents `AdvProtego`, a novel unified framework to mitigate model stealing attacks via different sorts of side-channels. The basic idea is to leverage the *adversarial attack* technique in machine learning to obfuscate side-channel leakage. Adversarial attack is a prominent threat to machine learning models [139], where the attacker injects carefully-crafted imperceptible perturbations into the input to mislead the victim model. In our context, since the attacker exploits machine learning to analyze side-channel signals and steal the model, the defender can also add such perturbations into the runtime execution to deceive the attacker into recovering wrong information.

Some preliminary studies have investigated the feasibility of adopting adversarial attacks for side-channel mitigation. For instance, a poster in CCS'2019 [57] proposed to add adversarial perturbations to power side-channel traces to protect cryptographic applications. Another workshop paper [58] used adversarial techniques to mitigate application fingerprinting through hardware performance counter

side-channels. However, significant gaps remain when applying this idea to safeguard deep learning models. First, different from cryptography or application fingerprinting, model stealing has totally different attack goals. **How to formulate the corresponding defense objectives in adversarial perturbation optimization is overlooked**. Second, existing studies only focus on one specific side-channel. The work [57] only experimented on the datasets without actual deployment. **How to generate accurate, fine-grained execution noise that corresponds to the desired adversarial perturbation on diverse hardware platforms presents another challenge**.

`AdvProtego` encompasses a set of innovative approaches from different system aspects to address these two challenges. First, we introduce two effective defense objectives for the defender to choose. (1) *Model Similarity Reduction*: this aims to increase the model stealing errors with the perturbed side-channel trace. (2) *Model Utility Reduction*: this aims to mislead the attacker to extract a model with the worst performance. To achieve these objectives, we design adapted adversarial algorithms and Neural Architecture Search (NAS) to identify the corresponding perturbation efficiently. Second, we present a general pipeline for generating and injecting precise and controllable runtime noise in the real-world environment. This pipeline consists of multiple critical steps, including noise quantization, calibration, trace collection, and noise injection, shedding light on the implementation over different computing platforms.

`AdvProtego` adopts a modular design at frontend, middleware and backend levels. It is general to cover different types of side-channels and platforms. Without loss of generality, we implement and evaluate `AdvProtego` with three scenarios: FPGA power side-channels, CPU cache side-channels, and GPU memory side-channels. Extensive experiments validate the advantages of `AdvProtego` across several aspects, including mitigation effectiveness, resource and computation overhead, defense transferability and robustness.

5.2 Preliminary

5.2.1 Model Stealing via Side-Channels

Model stealing attacks aim to reconstruct a target model's architecture, hyperparameters, or weights. Traditional stealing attacks rely on query-based methods, which use some input samples and the corresponding model responses to achieve extraction. These methods can be thwarted by limiting the query frequency [196], obfuscating the outputs [197, 198], or anomaly detection [201, 202]. Alternatively, the attacker can leverage some side-channel opportunities to peek into the deep learning application and retrieve sensitive data. These side-channels provide unintentional information leaks from the hardware during the model execution, including power consumption patterns [8, 18, 21–23, 29], execution time [20, 21], electromagnetic emissions [17, 19], cache access behaviors [10], and DRAM activity [9]. From such leaks, the attacker can extract the desired model attributes.

Below we describe three representative types of side-channel attacks for stealing deep learning models.

FPGA Power Side-Channels. The attacker collects the power consumption of the victim's execution to infer the secret. A popular platform targeted by this attack is FPGA-based AI accelerators. Previous works set up instruments physically close to the victim device for power trace collection [17–21]. Recently, researchers proposed remote power attacks, which leverage the multi-tenancy feature in cloud FPGA services [111, 203]) to steal secrets for better practicality and flexibility [29, 118, 119]. Although users' circuits on the same FPGA board are logically isolated [204], they share a common power distribution network (PDN). The supply voltage across the PDN fluctuates based on logic activity, creating a critical side-channel for information leakage [117]. The attacker can remotely deploy an on-chip monitor to extract information without physical hardware access.

CPU Cache Side-Channels. The distinct time of access data located at different levels of CPU caches and DRAM create a side-channel for the attacker to derive the victim's execution behaviors. The attacker deploys a malicious program on the same machine as the victim program. Although these programs are logically isolated by the operating system or hypervisor, they can possibly share the same

cache. The attacker adopts different techniques (e.g., Prime-Probe [205], Flush-Reload [206]) to deliberately cause cache contention that contain victim's memory access information. In the context of model stealing, the attacker is able to precisely monitor the occurrence of victim's low-level function calls (e.g., BLAS library), deduce the sequence of critical operations (e.g., matrix multiplications), and finally the model architecture [10, 121].

GPU Memory Side-Channels. Memory is another source of side-channel leakage, and has been exploited to attack deep learning models running on GPUs [9]. The attacker deploys a probe on the GDDR memory to collect electromagnetic emanations during model inference. Using such information, he can profile the victim's memory activity to determine read and write access volumes. Furthermore, he may monitor the PCIe events to deduce the number of kernel executions. These observations collectively allow the attacker to reconstruct critical details of the victim model.

5.2.2 ML-assisted Side-channel Analysis

The advance of machine learning (ML) algorithms has propelled side-channel analysis [126]. Attackers build ML models to automate the extraction of sensitive information from complex side-channel traces, which cannot be achieved with manual efforts. Typically, such attacks unfold in two phases.

1. **Offline Profiling Phase**: The attacker executes the target application over a secret set $\mathbf{s} = \{s_1, ..., s_n\}$ on either the same or similar platform. He collects N side-channel traces $\mathbf{T_{i,n}}$ corresponding to each secret s_i. Using such data, he constructs an ML model $f : \mathbf{T_{i,n}} \mapsto s_i$, establishing a correlation between the side-channel traces and secrets.
2. **Online Exploitation Phase:** With f developed from the previous phase, the attacker can exploit the acquired knowledge at runtime. By utilizing an additional set of q traces $T'_1, ..., T'_q$ captured from the targeted device, the attacker can infer the secret s'_i by evaluating $s'_i = f(T'_i)$.

ML-assisted side-channel attacks offer several advantages. (1) They can retrieve information even when discernible patterns in the side-channel trace are absent. This is particularly relevant in cases where computations of the victim program

occur in parallel, making manual pattern analysis infeasible. (2) ML models can seamlessly integrate all the information contained within a single trace, as they excel at handling high-dimensional data. In contrast, traditional methods necessitate the identification of specific points of interest to narrow down the information for the attack [207]. (3) ML models exhibit greater robustness against noise in the side-channel trace compared to conventional statistical techniques. This robustness enhances their efficacy in real-world scenarios.

Due to these attractive features, machine learning has also been adopted to analyze side-channel signals for model stealing. A common practice is to train a sequence-to-sequence (seq2seq) model, which maps the side-channel sequence to the model operation sequence. This has been realized in power [8], cache [10] and memory [9] side-channel attacks, where the attacker leverages a seq2seq model to precisely reconstruct the model architecture based on the sequences of power levels, critical function calls, and memory activities.

5.2.3 Threat Model and Defense Requirements

In this defense-oriented work, we deliberately adopt a conservative threat model by (1) assuming a strong adversary capable of launching both physical and remote side-channel attacks, and (2) constraining the defender's capabilities, with no administrative control over the underlying infrastructure. Such a setting can broaden the applicability of our proposed defense, making it suitable for diverse environments. This threat model has been commonly adopted in existing works [8–10, 29], which proves to be realistic and practical.

Specifically, an external attacker aims to use certain types of side-channels to steal victim's deep learning models. He performs the ML-assisted analysis: gathering side-channel data in the profiling phase to build a model f, and reconstructing victim's model architecture from the real-time side-channel traces T in the exploitation phase: $\mathrm{M} = f(T)$.

Conversely, the defender is also a non-privileged user, and possibly could be the target model's owner. He deploys the defense solution on the same system to prevent model stealing. The defense needs to satisfy the following requirements:

- *Effectiveness*. The defense can effectively disrupt attacker's side-channel observation such that his extracted model is significantly different from what he expects to obtain.
- *Computation efficiency*. The defense solution incurs minimal impact on the computation of the target platform, in terms of resource consumption, model latency and accuracy.
- *Implementation-friendly*. The defender only needs to implement a lightweight defense module at the same privilege level as the victim model. He does not need to modify or customize the target model.
- *Generalization*. The defense is general and adaptable to different types of models, platforms and side-channels, ensuring seamless transplantation to various environments.
- *Black-box defense*. The defender does not know attacker's ML model f. Instead, he can adopt a local surrogate model different from f as reference for the defense design.

5.3 AdvProtego

We design AdvProtego, a unified framework to achieve the defense requirements in §5.2.3.

5.3.1 Defense Objectives

The basic idea of AdvProtego is to leverage adversarial attack techniques [139] to craft noise δ and inject it into the runtime executions. By carefully adjusting both the scale of the noise and the injection moments, it can disrupt attacker's side-channel observations $T + \delta$ and mislead his extraction model f. Specifically, the defender aims to identify a command sequence $c = [c_1, \ldots, c_n]$ and execute each one c_i at the moment i. This will result in the desired noise δ and disrupt the attacker's analysis.

Different from the protection of cryptographic applications [57] or mitigating process fingerprinting [58], we need to design the new defense objectives for defending against model stealing attacks, which will guide the noise optimization subsequently. Considering the diverse defense scenarios and implementation feasibility, we propose two objectives:

1. **Model Similarity Reduction**: the defender aims to make the attacker's extracted model as distinct from the correct one as possible. This can be formulated as follows:
$$\arg\max_{\delta} L(f(T+\delta), M) \tag{5.1}$$
where L represents a distance function that measures the model difference, $f(T+\delta)$ represents the attacker's extracted model from the side-channel trace, and M is the actual victim's deep learning model.
2. **Model Utility Reduction**: the defender aims to make the attacker's extracted model have as low accuracy as possible. This objective can be formulated as follows:
$$\arg\min_{\delta} \texttt{Acc}(f(T+\delta)) \tag{5.2}$$
where `ACC` measures the testing accuracy of a model.

5.3.2 System Overview

Figure 5.1 displays the overview of `AdvProtego`, consisting of three major stages: frontend, middleware and backend.

The frontend is responsible for calculating the desired noise for different defense objectives. It includes two key modules: Attack Model and Noise Generator. The Attack Model takes preprocessed input traces to provide gradients and parameters that guide the Noise Generator. The Noise Generator employs adapted adversarial attack algorithms with masking mechanism to identify the optimal noise. The mask ensures noise is generated only in permissible locations while adhering to the backend's noise injection capabilities. It is worth noting that at this stage, the generated noise remains in the raw form, unsuitable for direct implementation on the target platforms.

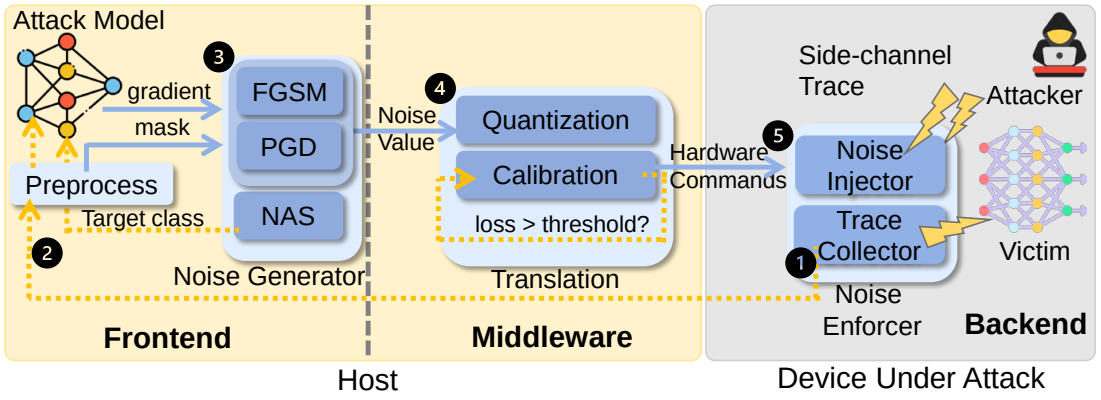


FIGURE 5.1: AdvProtego framework overview and workflow.

The middleware stage includes a Noise Translator Module, mapping the raw noise values into a corresponding sequence of executable commands on the platforms. The translation is required because the noise calculated in the frontend defines the intended effect but not the hardware-specific behavior.

The backend comprises a Noise Injector and Trace Collector module. Trace Collector captures the side-channel traces in the offline phase and forwards them to the frontend to guide the noise calculation. Noise Injector injects the calculated scale of noise into the execution at the calculated moment.

Workflow. In correspondence with ML-assisted side-channel analysis, AdvProtego also operates in two distinct phases.

In the *offline* phase, the defender begins by gathering side-channel traces through Trace Collector in the backend (❶). Since raw traces are often noisy and voluminous, a preprocessing module is employed to average the data points and analyzes permissible noise injection locations, generating a masking template for the Noise Generator. Using these pre-processed traces and model-specific attributes, the defender trains a surrogate Attack Model (❷). This model could be different from attacker's actual model f; the adversarial noise calculated from it can exhibit strong transferability to counter a variety of potential attacks. The surrogate model is then used to extract gradient information from the victim model's side-channel traces. Together with noise parameters such as the injection locations, such information is fed into Noise Generator, which calculates the desired noise tailored for specific defense goals (❸). The Noise Translator maps the generated noise to executable commands that can drive the underlying hardware components (❹).

In the *online* phase, the defender activates Noise Injector to inject the calculated noise during model inference (❺). The noise can effectively disrupt the attacker's ability to observe and analyze the victim's side-channel traces. By continually injecting carefully-crafted noise, `AdvProtego` provides robust protection against a wide range of side-channel attacks.

5.3.3 Frontend

The frontend of `AdvProtego` leverages collected side-channel traces to train a surrogate Attack Model, which is subsequently utilized to generate adversarial noise fulfilling different defense objectives. For *Model Similarity Reduction*, the optimization goal in Equation 5.1 is analogous to untargeted adversarial attacks, and we use an adapted version of FGSM [5] over the surrogate model to generate the adversarial noise. For *Model Utility Reduction*, we apply an adapted Neural Architecture Search (NAS) algorithm to obtain the worst-performing model M^w, and convert the optimization goal in Equation 5.2 to the following:

$$\arg\min_{\delta} L(f(T+\delta), M^w) \tag{5.3}$$

This is analogous to targeted adversarial attacks, and we use an adapted version of PGD [143] over the surrogate model for perturbation generation.

5.3.3.1 Adapted Adversarial Attacks

Directly applying the standard adversarial attack solutions in conventional ML tasks [5, 143] for side-channel trace analysis could yield suboptimal results, stemming from two key reasons. (1) Side-channel traces inherently contain large amounts of execution noise and uncertainty. Even for the same setting, the collected traces can vary significantly, making perturbations generated from one trace ineffective for others. This variability reduces the transferability and reliability of the generated adversarial noise. (2) The noise injection mechanisms in different platforms impose restrictions on both the type and location of the noise. For instance, many side-channel traces cannot be injected with negative noise values. In GPU side-channel attacks, injecting noise for memory read/write operations is easy, while it is far more challenging to obfuscate kernel execution latency.

To address these issues, we introduce two key adaptations to existing adversarial attack methods, as detailed in Algorithm 2. First, instead of generating individual perturbations for each trace, we create a universal perturbation by accumulating the ones from all traces, enhancing its transferability. Second, we define a mask $\mathcal{M}$ to indicate the permitted locations of the side-channel trace for noise injection. This mask ensures that perturbations are only applied to feasible regions of the trace. The Hadamard product ($\odot$) is used to apply the mask, ensuring compliance with hardware constraints.

Algorithm 2 Adapted Universal Masked Adversarial Attack

Input: $A, T = \{T_1, ..., T_n\}, M, Adv$ ▷ Attack model, Trace set, Mask, Adversarial procedure
Output: *delta* ▷ Universal adversarial noise
Initialize $\delta \leftarrow \mathbf{0}$
for each trace $T_i \in T$ **do**
    $\delta \leftarrow \delta \odot M$ ▷ apply mask (element-wise)
    $T_i' \leftarrow T_i + \delta$ ▷ inject current universal noise
    $T_i' \leftarrow Adv(A, T_i')$ ▷ apply adversarial step on T_i'
    $\delta \leftarrow T_i' - T_i$ ▷ update universal perturbation
    Clamp δ to backend constraints ▷ e.g., magnitude / discrete levels
return δ

5.3.3.2 Adapted NAS Algorithm

NAS is a mature automated ML technology that searches for the model architecture with the best performance [208–210]. In our defense objective of Model Utility Reduction, we want to identify the worst architecture as the target to deceive the attacker. To this end, we reverse the metric used in NAS to guide the search process to the opposite direction. This can be formulated as follows:

$$\pi^* = \arg\max_{\pi_i \in \Omega} \mathcal{L}(W_{\pi_i}; X_{val}) \tag{5.4}$$

where W_{π_i} is the network parameters associated with the model π_i, Ω is the search space, X_{val} is the validation dataset, $\mathcal{L}(\cdot)$ stands for the loss function, and π^* is the target model we expect to find out.

Researchers have proposed different algorithms to achieve efficient and effective NAS. We opt for NAS-RL [209] to implement our defense. It is based on reinforcement learning, and uses the reward (i.e., accuracy on the validation set acc_{val}) to guide the search process. Therefore, we adjust the reward from acc_{val} to $1 - acc_{val}$, to find the worst model.

5.3.4 Middleware

In this stage, the Noise Translator converts the generated noise from the frontend into platform-specific commands that could cause the desired defense effect. The translation undergoes two processing steps.

Quantization. The calculated noise using adversarial attack algorithms is continuous, while the noise we can inject into the execution is normally discrete. Therefore, we need to quantize the noise into discrete levels controllable by the Noise Injector in the backend. The number of levels is determined by the specific noise injection mechanisms. A greater number of levels enables more precise trace manipulation, allowing the injected noise to produce effects more closely aligned with expectations. However, this also places higher demands on the backend to control the noise. Formally, this quantization can be modeled as a constrained projection problem. Adversarial noise $\delta = [\delta_1, ..., \delta_n]$ is quantized to fit the range R of the noise injector. Using the formula

$$\text{round}\left(\frac{(\delta_i - \delta_{\min})(\delta_{\max} - \delta_{\min})}{R}\right),$$

we discretize each value, and then use a mapping table M to convert them into hardware commands: $M[\text{round}((\delta_i - \delta_{\min})(\delta_{\max} - \delta_{\min})/R)]$.

Calibration. This step establishes a mapping table from platform control commands to actual noise effects. This can be achieved in an iterative manner. Formally, we first identify a set of possible command patterns that could inject execution noise. For each pattern r_i, we measure the corresponding noise level n_i, and form a pair $R_i = [r_i, n_i]$. We further sort these pairs by n_i to construct the mapping table M, where each pattern r_i maps to a discrete noise level n_i. We keep adjusting the command patterns to minimize the discrepancies between the measured noise level $\hat{n_i}$ and recorded noise level n_i in M. A loss value, defined as

$\mathfrak{L} = \sum_{i=1}^{n} |n_i - \hat{n_i}|$, evaluates such inconsistencies at each iteration. Calibration continues until the loss falls below a predefined threshold, ensuring accurate and reliable noise injection.

5.3.5 Backend

This stage is responsible for collecting the side-channel traces, and injecting the noise into the side-channel trace at runtime by executing the identified commands. These are achieved by the following two modules, respectively.

Trace Collector. This module gathers the execution traces in the offline phase to build the surrogate model. Depending on the specific platforms and side-channels, `AdvProtego` adopts diverse profiling tools to collect such information.

Noise Injector. In the online protection phase, this module executes the identified commands at the identified moments to inject the desired noise. The noise is indistinguishable from the side-channel trace by the attacker, while remaining transparent to the victim's execution. The implementation of this module is also dependent on the computing platforms and side-channels in consideration. It could be integrated into the hardware, or deployed as a separate process co-locating with the victim.

5.4 Implementation

In `AdvProtego`, the frontend is general and platform-independent. As the noise must be ultimately be translated into platform-specific commands and injected through available interfaces, the middleware and backend implementations must be tailored to the specific computing platform and side-channel characteristics. In this section, we provide case studies that illustrate how `AdvProtego` can be instantiated under three representative attack scenarios.

5.4.1 FPGA Power Side-Channels

Noise Translator. A common strategy of injecting power noise into the FPGA is to implement Ring Oscillators (ROs). Therefore, Noise Translator needs to map the noise values to the commands that control RO circuits on the FPGA board.

The first step, quantization, converts the generated adversarial noise into 321 discrete noise levels, controlled by different RO enable patterns. The number of levels is determined by the capacity of Noise Injector in the backend. The second step, calibration, needs to consider the impacts of several factors like temperature, runtime noise, and hardware placement. Power side-channels on FPGAs are particularly noisy and unpredictable. As discussed in [211], the adoption of ROs introduces undershoot and overshoot effects in the on-board Power Monitor readouts, leading to transient fluctuations and complicating the precise noise injection. Additionally, voltage transients often correlate with preceding RO enable counts, making accurate realization of noise patterns even more challenging. We adopt the iterative calibration approach in §5.3.4 to address these issues, establishing the mapping between 321 RO enable patterns and distinct noise levels.

Trace Collector. Following [8], we employ a Time-to-Digital Converter (TDC) as the power sensor. The TDC operates by capturing the combinational logic delay, leveraging the propagation of a clock signal through a chain of buffers to detect voltage fluctuations. Specifically, the clock signal enters the TDC and passes through an adjustable coarse delay line and a fine delay line, which together introduce an initial delay. This delay is then fed into a tapped delay line composed of `CARRY4` primitives. The tapped delay line tracks the taps reached by the clock signal, providing a delay measurement. Due to variations in switching activities across different operations of the FPGA, discrepancies in voltage drop values may arise, consequently leading to different delay measurements in the TDC. Through this mechanism, we can infer the activities of adjacent circuits using the TDC's readout.

Noise Injector. Following [212], we use ROs to generate and inject noise. In our design, a single RO unit requires 4 look-up table (LUT) elements. We assemble 64 RO units to form a set, and Noise Injector comprises 32 such sets. To control the enable and disable operations of all sets effectively, we utilize a 32-bit-long AXI

bus, with each bit in the bus corresponding to one set of ROs. This ensures efficient control and coordination of noise injection.

5.4.2 CPU Cache Side-Channels

Noise Translator. In cache side-channels, the attacker normally determines whether a critical function occurs within one interval but not the specific number of occurrences. As a result, the noise input is limited to binary values (0 and 1), restricting the noise values to 1 accordingly. To address this constraint, the frontend clamps the generated noise to contain only 0s and 1s, ensuring the noise structure matches the format of the input side-channel traces. Noise Translator in the middleware then processes this binary noise by identifying the positions corresponding to 1s, which represent the moment of function invocations. It generates an array of noise injection commands $[d_i, c_i]$, where d_i refers to the delay before each function call, and c_i indicates the function to be called. Cache noise is easier to translate without the need of quantization and calibration.

Trace Collector. Inspired by existing cache side-channel attacks for model stealing [10, 121], we employ the Flush-Reload technique to monitor the invocation of specific functions in a dynamic shared library, i.e., `sgemm_incopy` and `sgemm_oncopy` APIs in OpenBLAS. By continuously monitoring the cache access patterns, Trace Collector generates detailed side-channel traces in the form of $[I_i, O_i, T_i]$, where T_i represents the timestamp of the monitoring interval, I_i and O_i are binary values indicating whether `sgemm_incopy` and `sgemm_oncopy` are called during that interval.

Noise Injector. Injecting noise to obfuscate cache side-channel traces is relatively straightforward. Since any access to the monitored functions triggers cache reloads, we implement a program that dynamically loads the library and accesses the specified function addresses at pre-set time intervals. Specifically, Noise Injector takes $[d_i, c_i]$ from Noise Translator as input, where d_i specifies the delay for each event, c_i denotes whether `sgemm_incopy` or `sgemm_oncopy` should be invoked at the corresponding delay. By executing these function in alignment with the input sequence, Noise Injector effectively produces the desired side-channel noise pattern.

5.4.3 GPU Memory Side-Channels

Noise Translator. Memory side-channel attacks work by monitoring GPU GDDR read/write access amounts and kernel execution latency. Due to potential discrepancies between the input values provided to the kernels and the actual read/write amounts resulting from program optimizations during compilation or execution, a calibration process is necessary. In this process, two iterative loops incrementally increase the read and write operations from 0 byte to the maximum observed read and write amounts across all kernels in the models, using a step size of 4 bytes. During each iteration, Trace Collector measures the actual read and write amounts and maps these values to their corresponding input values. This guarantees the noise injection process is precise and dependable.

It is challenging to inject noise to kernel execution latency. To this end, the frontend applies a mask that sets the noise in the latency dimension to 0, ensuring no perturbations are added to the latency values. Noise Translator subsequently generates the command pattern $[W_i, R_i, T_i]$, denoting the write data amount, read data amount, and execution latency, respectively.

Trace Collector. We use the Nsight Compute tool to simulate the EM side-channel, monitoring memory read, write operations, and CUDA kernel execution latency. It provides detailed profiling data, including DRAM throughput and memory access patterns, enabling accurate observation of GPU behaviors. We record metrics such as memory throughput and execution time during kernel execution for trace collection. Trace Collector outputs an array of $[W_i, R_i, T_i]$, where W_i and R_i represent the observed write and read volumes during the i-th kernel, and T_i denotes the kernel execution latency.

Noise Injector. We utilize CUDA kernels to simulate dynamic memory operations, including reads and writes on the GPU DRAM. Since controlling execution latency on a GPU is inherently challenging, Noise Injector does not attempt to manipulate execution timing. Instead, it introduces noise by generating controlled memory access patterns, designed to mimic typical operations found in GPU workloads.

Noise Injector consists of two primary components: 1) Write kernel performs sequential writes to the GPU memory. Each thread calculates data values based on

its index and iterates. To prevent these operations from being optimized out, the kernel includes a dependency on the loop index (i) during the dummy value computation ($value+ = i * 0.1f$). This calculation ensures that each write operation involves actual computation, making it non-trivial for the compiler to discard. The results are stored in GPU memory and later transferred back to the host before freeing the allocated memory. 2) Read kernel simulates DRAM read operations by processing input data and aggregating results through shared memory. To further prevent optimization, the kernel performs a reduction operation within the shared memory and uses `atomicAdd` to update the global result. The use of shared memory and synchronization primitives (`__syncthreads`) ensures the calculations are preserved as functional and necessary and the noise generation cannot be optimized away. The read results are transferred to the host before releasing the GPU resources.

5.5 Evaluation

5.5.1 Experiment Setup

Testbed. For FPGA power side-channels, we use a Xilinx Zynq-7000 SoC ZC706 FPGA board (xc7z045ffg900-2) with NVIDIA Deep Learning Accelerator (NVDLA) [124]. The ARM processor on the board runs Ubuntu 16.04 OS, supporting NVDLA alongside the driver for Power Monitor and Noise Injector. Hardware implementation is carried out using Vivado 2019.1. For CPU cache side-channels, experiments were conducted on an AMD EPYC 7313P Processor. We utilize PyTorch v2.3.0 and OpenBLAS v0.3.20 as the linear algebra library. For GPU memory side-channels, experiments were conducted on an NVIDIA GeForce RTX 4080 GPU.

To train the surrogate model and generate adversarial noise for all experiments, we employ PyTorch v1.13 and CUDA v11.6. This training process was conducted on a server equipped with NVIDIA GeForce RTX 3090 GPUs.

Datasets and Models. For FPGA power side-channels, due to the memory resource constraint of the adopted FPGA board (e.g., 1 GB memory and lack of advanced operators), we mainly consider the image classification tasks over the MNIST and CIFAR-10 datasets. This choice is consistent with prior side-channel

works [20, 29, 118]. Since the adversarial attack is a fundamental feature to AI models and tasks regardless of their scales, we believe our solution can be applied to larger models with stronger hardware support, which will be our future work. For each dataset, we run the evaluation on 200 randomly generated models. These models have random numbers (within the range of [2, 16]) of network layers. The composition of each layer is also determined through random selection. The selection space includes 12 convolution layers (with the kernel size of 2, 3, 4, 5 and output size of 10, 20, 30), 4 pooling layers (with the kernel size of 2, 3, 4, 5), 5 fully-connected layers (with the output size of 100, 200, 300, 400, 500), 1 ReLU layer, and 1 Softmax layer. The initial pre-training of these models is conducted using Caffe. Subsequently, calibration is performed utilizing TensorRT, followed by compilation through the NVDLA compiler. These models are generated on the host computer and subsequently executed on the FPGA using NVDLA runtime.

For CPU cache and GPU memory side-channels, we use image classification tasks on the ImageNet dataset. The evaluation encompasses 1000 randomly generated models with network layers ranging between 6 and 112. The composition of each model is determined by a random selection of layers from a broader pool, which includes convolutional layers, fully connected layers, pooling layers, batch normalization layers, dropout layers, ReLU layers, and structural elements such as concatenation and addition layers. At each step, we randomly select from block types including sequential, addition, and concatenation blocks. Sequential block configurations are chosen from commonly used patterns such as (Conv, ReLU), (FC, ReLU), and (Conv, ReLU, Pool), optionally incorporating batch normalization.

Attacks and Defense Baselines. `AdvProtego` is designed as a general framework to fundamentally compromise the ML-assisted process for model stealing. Without loss of generality, we choose three representative attacks as our defense targets, including a FPGA power attack [8], CPU cache attack [10] and GPU memory attack [9]. We also select five state-of-the-art defense solutions as baselines, including three noise injection mechanisms [35, 67, 97][2] and two model architecture obfuscation mechanisms [98, 99][3]. Extended evaluations to other attacks and defenses will be future work.

[2] [97] is only evaluated on FPGA, as its real-time feedback cannot be achieved in cache and memory components

[3] Since the source code of [99] is unavailable, and [98] does not include ImageNet-based implementations, we rely on the data reported in the original papers for comparison

Metrics. We adopt two evaluation metrics to demonstrate the effectiveness of `AdvProtego`. For *Model Similarity Reduction*, the model architecture is represented as a variable-length sequence where each element denotes a layer type. Following prior works [8, 10], we use the Layer Error Rate (LER). It is defined as $LER = L(s', s)/\|s\|$, where $\|s\|$ is the length of sequence s, and $L(s', s)$ is the edit (Levenshtein) distance between the two sequences s' and s. In the following evaluation, *LER-to-label* denotes the LER between attacker's extracted architecture and victim model's ground-truth sequence, while *LER-to-target* denotes the LER between attacker's extracted architecture and defender's searched model with the worst performance. A larger *LER-to-label* indicates lower model similarity, thus higher defense effectiveness. For *Model Utility Reduction*, we train each extracted model under the same conditions and measure the accuracy drop compared to the original model; a higher drop indicates stronger defense.

5.5.2 Overhead

For FPGA power side-channels, we analyze the resource utilization on the FPGA board. In our implementation, each RO utilizes only 4 LUTs. With each set containing 64 ROs and a total of 32 such sets, our design efficiently employs only 8,192 LUTs. Factoring in peripheral circuits such as the AXI bus, the aggregated resource utilization for our design stands at 8,251 LUTs and 170 flip-flops (FF). A comprehensive comparison of overhead across various defense solutions from prior works is presented in Table 5.1. The "Area Increased" column shows the ratio of the number of LUTs and flip-flops used by the protective measures to the corresponding counts in the circuits under protection. The "Utilization" column computes the proportion of LUT and flip-flop counts relative to the available resources on the FPGA. For schemes targeting the AES accelerator, the resource usage is estimated based on the implementation in [213], where 2,640 LUTs and 1,682 FFs are utilized (as the information is not disclosed in these papers). The table demonstrates that among all schemes, `AdvProtego` exhibits the least resource consumption in relation to the circuit under protection. To estimate the power consumption, we rely on the Vivado power report to gauge the power usage of the Noise Unit. Its estimated power consumption amounts to 0.001W, constituting less than 1% of the total power consumption (1.737W).

TABLE 5.1: Overhead for power side-channels

Solution	LUT	FF	BRAM	Utilization	Target	Area Increased
[214]	4608	1152	0	4.10%	AES	175% / 69.0%
[36]	321	258	0	0.70%	AES	12.1% / 15.3%
[90]	8000	6499	163	8.63%	BNN	430% / 580%
[91]	9818	7709	163	10.43%	BNN	540% / 680%
`AdvProtego`	8251	170	0	1.28%	DNN	**10.1%** / **0.1%**

For CPU cache and GPU memory side-channels, we analyze the additional latency (ΔLatency), increased floating-point operations per second (ΔFLOPS), and the accuracy impact (ΔAcc) of each scheme. Table 5.2 summarizes these comparisons. For NeurObfuscator [99], we compare the configuration with a LER similar to `AdvProtego`, as reported in [99]. The FLOPs data for ObfuNAS [98] is based on experiments conducted with ImageNet. The configurations of these schemes align with those listed in Table 5.3. As shown in Table 5.2, `AdvProtego` consistently achieves the lowest overhead across all metrics, introducing no additional latency or accuracy degradation. This highlights `AdvProtego`'s efficiency and practicality in real-world applications.

TABLE 5.2: Overhead for cache and memory side-channels

Defense	ΔLatency (%)	ΔFLOPS (%)	ΔAcc (%)
NeurObfuscator[99]	2	53	-
ObfuNAS[98]	-	25	-1
CATalyst[38]	0.7	N.A.	N.A.
`AdvProtego` (CPU)	-0.05	0	0
`AdvProtego` (GPU)	0.09	0	0

5.5.3 Effectiveness

Model Similarity Reduction. We analyze the adversarial noise generated by `AdvProtego` through its Noise Injector across different platforms. Table 5.3 compares the LER of `AdvProtego` and other defense methods. For fair comparison, we set a maximum RO set enable number as 8 for all defenses implemented on FPGA, and the maximum memory read/write amount is limited to 1KB for experiments

on GPUs. We observe that the LERs of our delicate adversarial noise are significantly higher than other methods, indicating its higher effectiveness in reducing the similarity of stolen model.

TABLE 5.3: LER comparisons for Model Similarity Reduction

Defense	Dataset	Platform	LER (%)
[67]	MNIST	FPGA	75.6
[35]	MNIST	FPGA	95.4
[97]	MNIST	FPGA	70.5
[67]	CIFAR10	FPGA	84.5
[35]	CIFAR10	FPGA	49.2
[97]	CIFAR10	FPGA	89.1
[67]	ImageNet	GPU	56.3
[97]	ImageNet	GPU	57.7
[67]	ImageNet	CPU	52.2
[97]	ImageNet	CPU	61.8
`AdvProtego`	MNIST	FPGA	**144.1**
`AdvProtego`	CIFAR10	FPGA	**128.3**
`AdvProtego`	ImageNet	CPU	137.2
`AdvProtego`	ImageNet	GPU	**156.7**

Model Utility Reduction. For FPGA side-channels, the maximum number of enabled RO sets is 32. For GPU memory side-channels, a maximum 1 KB memory read/write limit is imposed. For CPU cache side-channels, the noise generation process is restricted to no more than 5% of the victim model's total invocations to the target. Table 5.4 shows the evaluation results. "Acc Drop (Searched)" indicates the accuracy decrease of the identified target model relative to the victim's original model. "Acc Drop (Extracted)" refers to the actual accuracy degradation of attacker's extracted model relative to victim's original model when noise is incorporated into the side-channel. A larger accuracy drop signifies a greater degradation of the extracted model and higher defense effectiveness.

We observe that the accuracy drop in attacker's extracted model closely aligns with the target accuracy identified during the search phase. This indicates that `AdvProtego` generates and injects noise with high precision, achieving the intended obfuscation effect. Furthermore, the extracted model's accuracy drop is significantly larger compared to all the other defense methods, confirming that `AdvProtego` effectively reduces model utility. The main reason is that existing

TABLE 5.4: LER and accuracy for Model Utility Reduction

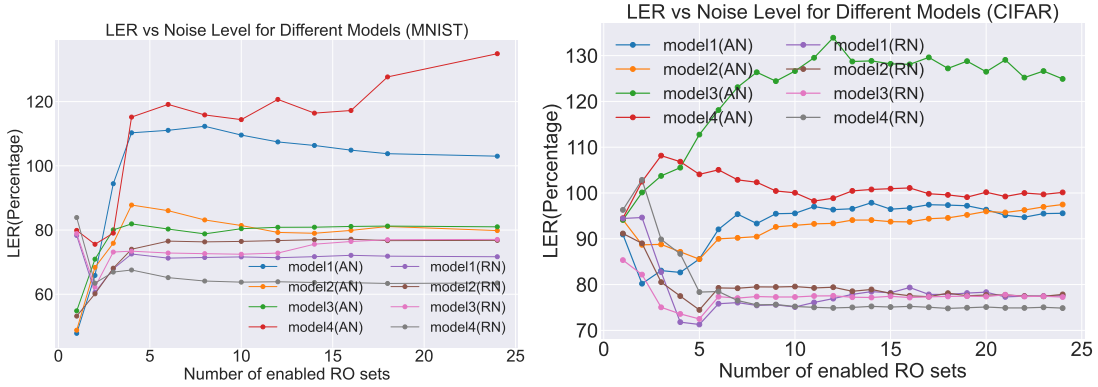
Defense Methods	Dataset	Platform	Acc Drop (Searched)	Acc Drop (Extracted)
[67]	MNIST	FPGA	-	-0.7%
[97]	MNIST	FPGA	-	-0.8%
[35]	MNIST	FPGA	-	-1.9%
[67]	CIFAR10	FPGA	-	-8.5%
[97]	CIFAR10	FPGA	-	-2.7%
[35]	CIFAR10	FPGA	-	-8.8%
[99]	ImageNet	All	-0.4%	-
[98]	ImageNet	All	0.1%	-
`AdvProtego`	MNIST	FPGA	2.9%	**1.6%**
`AdvProtego`	CIFAR10	FPGA	20.3%	**17.4%**
`AdvProtego`	ImageNet	GPU	13.6%	**10.3%**
`AdvProtego`	ImageNet	CPU	13.6%	9.9%

defenses [35, 67, 97, 99] mainly focus on distorting attacker's observations, while ignoring the utility of the extracted models. Hence, the attacker can still extract a good model.

The defense effectiveness varies across different scenarios. (1) `AdvProtego` performs better on complex tasks and datasets (e.g., CIFAR10, ImageNet) than simple ones (MNIST). This is because in simple tasks, it is too easy for a model to reach high accuracy that even the searched worst-performing model has satisfactory performance. This leaves little room for accuracy degradation via noise, limiting the observed impact. (2) `AdvProtego` performs better on FPGAs than CPUs and GPUs. This discrepancy likely arises from the constraints of side-channels in CPUs and GPUs, which limit the ability to inject noise into temporal features. On FPGAs, Noise Injector can achieve finer granularity and higher effectiveness.

5.5.4 Transferability

To measure the transferability of the adversarial noise from defender's surrogate model to attacker's model, we construct five models with varying structures in Table 5.5. Model 0 is the surrogate model used by the defender to generate adversarial noise, while models 1-4 are used by the attacker for side-channel analysis. We mainly conduct our experiments on FPGA. We diversify the noise intensity by adjusting the number of RO enabled sets, allowing us to analyze the impact of



(A) Results for MNIST dataset (B) Results for CIFAR10 dataset

FIGURE 5.2: LERs for different attack models. "AN": adversarial noise from `AdvProtego`; "RN": random noise

varying levels of interference on the transferability and overall defense effectiveness.

TABLE 5.5: Details of the defender's surrogate model and attacker's actual models

Model	Structure	Details
0	CNN-RNN-CTC	3 CNN layers, 2 RNN layers
1	CNN-RNN-CTC	1 CNN layer, 2 RNN layers
2	CNN-RNN-CTC	2 CNN layer, 2 RNN layers
3	CNN-RNN-CTC	5 CNN layer, 2 RNN layers
4	Transformer	1 encoder, 1 decoder layer

Model Similarity Reduction. We measure the LERs between the extracted and victim models. The results are shown in Figure 5.2. In both datasets, `AdvProtego` induces significantly higher error rates in the extracted models compared to using random noise across four distinct models. This underscores the efficacy of `AdvProtego` against diverse attack models.

Model Utility Reduction. We proceed to test the transferability of the Model Utility Reduction defense strategy. The evaluation results for this defense objective are shown in Table 5.6."Acc Drop (Extracted)" represents the actual accuracy drop of the attacker's extracted model compared to the victim's model when the identified adversarial noise is implemented. A higher extraction accuracy drop indicates a greater degradation of the extracted model, thus higher defense effectiveness. The results clearly show that the extracted accuracy of the attacker's model is lower than the victim's model accuracy for all the four cases. In model 2,

the defense result is even better than using the surrogate model (model 0) for extraction. This confirms the transferability and effectiveness of `AdvProtego` against different attack models.

TABLE 5.6: Transferability results for Model Utility Reduction

Model	Dataset	Acc Drop (Extracted)
1	MNIST	0.7%
2	MNIST	3.0%
3	MNIST	0.8%
4	MNIST	0.8%
1	CIFAR10	10.5%
2	CIFAR10	15.2%
3	CIFAR10	12.0%
4	CIFAR10	10.4%

5.5.5 Resilience against Adaptive Attacks

We explore the scenarios where the attacker is aware of our defense mechanisms and attempts to bypass them.

Adversarial attack mitigation as adaptive attacks. A potential adaptive attack is to apply existing mitigation solutions against adversarial attacks to circumvent `AdvProtego`. However, as adversarial attacks are still an unsolved threat [215], these solutions still exhibit some limitations to undermine adversarial perturbations. For evaluation, we mainly consider three advanced techniques. (1) Sample resizing and rescaling [216]. This involves adding random resizing and padding layers at the beginning of the target model. These layers alter the spatial positioning of adversarial perturbations to make them ineffective. Following [216], the attacker can randomly resize and pad 0s to the captured side-channel traces. (2) Randomized smoothing [217]. This entails training a neural network with Gaussian data augmentation to create a smoothed classifier. This classifier is then used to predict the class most likely to be returned by the model when the input is perturbed by isotropic Gaussian noise. We set the noise hyperparameter σ to 0.5 during training and testing, with a sample size of 1000 for each input trace. (3) bit-depth reduction [218]. This involves simple quantization to remove small adversarial variations in the input. We reduce the input side-channel traces to 8

bits. We conducted experiments on FPGA with MNIST to evaluate if `AdvProtego` can resist those operations when employed by the attacker to process the power side-channel trace.

Figure 5.3 presents the results of Model Similarity Reduction. The results suggest that these adversarial attack defense mechanisms can reduce the protection capability of `AdvProtego` by about 15% (when the number of RO sets is 2). Such reduction is not significant as `AdvProtego` is still more effective under these adaptive attacks than the Random Noise baseline defense. Table 5.7 shows the results of Model Utility Reduction. We observe that resize & rescaling and bit-depth reduction slightly diminish the effectiveness of `AdvProtego`, but it remains effective as the utility of the extracted model is still worse than the originals, indicated by the columns"Acc Drop (Searched)" and "Acc Drop (Extracted)". Surprisingly, with randomized smoothing applied, the defense results are even better than without it. In conclusion, these adaptive attacks do not perform as effectively as they do for conventional AI tasks. This could be due to the inherently noisy and unpredictable nature of side-channel traces, making it challenging to leverage these countermeasures to circumvent our defense.

Noise isolation as adaptive attacks. The attacker might also consider isolating the injected noise directly. However, our injected noise is practically indistinguishable from normal model execution. For instance, on FPGA platforms, both normal operations and injected noise contribute to power fluctuations captured by the same sensors. On GPUs, only coarse-grained aggregated memory statistics are accessible while privileged access—kernel-level granularity is unavailable. As a result, even if an attacker is aware that a defense is in place, isolating the noise component is unrealistic.

TABLE 5.7: LER and accuracy for Model Utility Reduction with adversarial attack defense mechanisms applied.

Defense	LER to label	LER to target	Acc Drop (Searched)	Acc Drop (Extracted)
No defense	67.2%	15.3%	2.9%	2.2%
Resize & rescaling	73.5%	46.3%	2.9%	1.2%
Randomized smoothing	68.7%	5.9%	2.9%	2.5%
Bit-depth reduction	76.1%	46.5%	2.9%	1.3%

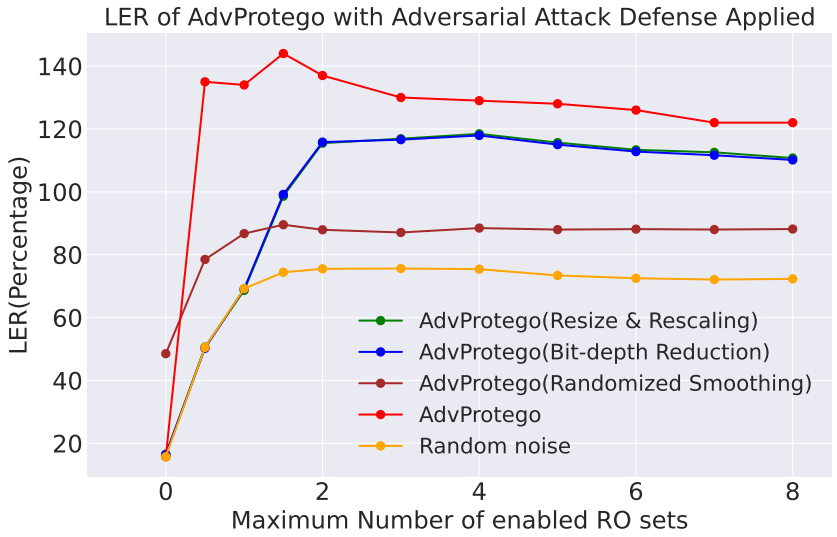


FIGURE 5.3: LERs for random noise and `AdvProtego` with adversarial attack defense mechanisms applied.

5.5.6 Discussion: Generalizability and Deployment Considerations

Generalizability. We evaluate `AdvProtego` on three representative leakage modalities and platforms, i.e., FPGA power, CPU cache, and GPU memory side-channels. Its modular frontend–middleware–backend design is intended to make the framework portable rather than tied to any single platform: the frontend and the adversarial-noise formulation are largely reusable, while adaptation to a new accelerator or leakage modality is localized to the middleware and backend, which encapsulate the platform-specific monitor and Noise Injector. We therefore expect the approach to transfer to future accelerator architectures and evolving deep learning frameworks. For a new platform, the surrogate model must be re-trained on that platform's leakage characteristics, and the set of noise-injection primitives must be re-established. Leakage sources that fall outside the modeled channels would likewise require extending the middleware. Establishing these components on a broader range of platforms remains future work.

Deployment considerations. Beyond the runtime and resource overhead reported in Section 5.5.2, practical deployment of `AdvProtego` entails additional, largely one-time engineering costs that we discuss here for completeness. Instantiating the defense on a platform requires integrating a leakage monitor and a Noise Injector into the execution environment (e.g., the accelerator driver or the inference

runtime) and collecting profiling traces to train the surrogate model. This integration effort, together with the need to re-profile and re-tune when the hardware, driver, or framework version changes, constitutes the main long-term maintenance burden: as the underlying software stack evolves, the surrogate and the injected-noise strategy may drift and require periodic recalibration. We view this as an acceptable cost for a defense that operates transparently at the side-channel level, but we note that reducing this maintenance effort, for example through automated re-profiling or platform-independent noise primitives, is an important practical direction for making the framework easier to sustain in production settings.

5.6 Summary

We present `AdvProtego`, a novel unified framework aimed at protecting deep learning models from side-channel-based model stealing. It applies adversarial attack techniques to craft execution noise to obfuscate side-channel observations while imposing minimal overhead on the model. Our contributions include the followings: (1) proposing two defense objectives and formulating them in the context of adversarial perturbation optimization. (2) introducing an end-to-end modular design with innovative algorithms and techniques to achieve practical and unified protection. We implement `AdvProtego` in three representative attack environments to demonstrate its excellence and superiority over existing solutions.

Chapter 6

`ObfusBFA`: A Holistic Approach to Safeguarding DNNs from Versatile Bit-Flip Attacks

Bit-flip attacks (BFAs) represent a serious threat to Deep Neural Networks (DNNs), where flipping a small number of bits in the model parameters or binary code can significantly degrade the model accuracy or mislead the model prediction in a desired way. Existing defenses exclusively concentrating on specific attacks and platforms, while lacking effectiveness for other scenarios.

In this chapter[1], we propose `ObfusBFA`, an efficient and holistic methodology to mitigate BFAs targeting both the high-level model weights and low-level codebase (executables or shared libraries). The key idea of `ObfusBFA` is to introduce random dummy operations during the model inference, which effectively transforms the delicate attacks into random bit flips, making it much harder for attackers to pinpoint and exploit vulnerable bits. We design novel algorithms to identify critical bits and insert obfuscation operations. We evaluate `ObfusBFA` against different types of attacks, including the adaptive scenarios where the attacker increases the flip bit budget to attempt to circumvent our defense. The results show that `ObfusBFA` can consistently preserve the model accuracy across various datasets and DNN architectures while significantly reducing the attack success rates. Additionally, it

[1]The content of this chapter is accepted in IEEE International Symposium on Hardware Oriented Security and Trust (HOST), 2026.

introduces minimal latency and storage overhead, making it a practical solution for real-world applications.

6.1 Introduction

The rapid growth of deep learning technology has driven the widespread deployment of Deep Neural Network (DNN) models across a variety of scenarios, from autonomous driving [219] to embedded devices [220]. However, the proliferation of DNNs has exposed new security vulnerabilities. Past studies have shown that modern hardware platforms are susceptible to Bit Flip Attacks (BFAs) [221], which corrupt the critical data or code of the applications, significantly compromising their integrity. Such attacks could be executed through different fault injection techniques, such as row hammering [15, 16], undervolting [13], and laser beaming [14]. In the domain of deep learning, researchers apply BFAs to flip a small number of critical bits in the DNN applications [16, 25], which can severely degrade the model's overall accuracy, or misprediction for certain input. Such attacks can lead to catastrophic consequences, especially in safety-critical scenarios.

Existing BFAs against DNN applications can be classified into two categories, as shown in Figure 6.1. (1) *Model-level attacks* [16, 24–28]: the attacker compromises some critical model parameters. Even flipping a tiny number of parameter bits can drastically alter the model behaviors, resulting in a sharp drop in accuracy or misprediction in the attacker's desired manner. (2) *Code-level attacks* [56, 108]: the attacker targets the functional code of the deep learning framework, including the executables generated by the compilers, or the underlying computation libraries. By flipping critical bits in these foundational components, the attacker can hijack the control flow of the inference execution, leading to large performance degradation or even system failure.

6.1.1 Existing Studies and Limitations

In spite of existing efforts to defend DNN models against BFAs, it is still challenging to design a holistic solution to provide comprehensive protection over various threat

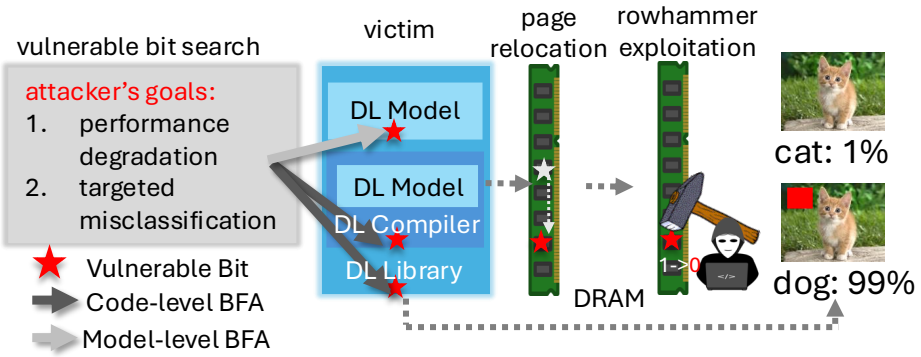


FIGURE 6.1: BFAs against DNN applications with Rowhammer.

scenarios. As BFAs are commonly realized via exploiting certain hardware vulnerabilities, a straightforward strategy is to fundamentally mitigate those threats. For instance, a number of solutions have been introduced to defeat Rowhammer attacks, including the adoption of timers [59], performance counters [60, 61], error-correcting codes [62], memory isolation [63, 64], or machine learning-based detection [222]. While effective, these solutions often require specialized hardware upgrades, which can introduce compatibility issues in existing systems and limit their broader adoption. Besides, they lack generalization across different computing platforms and scenarios.

Due to the above limitations, researchers have been actively seeking for new solutions dedicated to the protection of DNN models against BFAs. Existing approaches can be classified into two categories. (1) *Redesigning the DNN model for better robustness*. This is normally achieved via model architecture modifications or retraining. Typical examples include the dynamic multi-exit architecture [44], weight reconstruction [45], quantization [46, 47], obfuscations of channels [48, 49] or bits [50]. (2) *Runtime integrity checking*. This strategy continuously monitors the DNN inference, detects and corrects potential errors caused by flipped bits. Some works introduce checksum [51, 52], hash functions [53, 54], or checker DNNs [55] for integrity verification. Other studies focus on the detection of critical bits as BFA targets [178, 223, 224].

Unfortunately, these approaches still exhibit several drawbacks in practice. (1) **High access requirement to the DNN models**. Some solutions require significant changes over the model parameters or network designs [44–50], which are often difficult for pre-deployed models or legacy systems. Some solutions require the

access to the original training data [52], which may not be always feasible. (2) **Non-negligible runtime overhead**. Runtime integrity checking incurs additional latency for the inference execution, making it impractical for applications with strong real-time requirements [55, 178]. For example, on the CIFAR-10 dataset, DeepDyve [55] incurs a 9.1% computational overhead, while NeuroPots [178] adds a 9.7% time overhead. (3) **Degradation of model functionality**. Solutions that modify model parameters often compromise model performance while providing protection [44, 46, 47, 178, 225]. As reported in prior work [44, 52], BIN [46] reduces model accuracy by approximately 4.1%, RA-BNN [47] by 2.9%, NeuroPots [178] by 1.4%, and Aegis [44] by 1.2%. (4) **Lack of generalization**. More importantly, these solutions only target one specific attack exclusively. The majority of the works focus on the bit flips in model parameters [44–51, 53, 54, 178, 223, 224]. To our best knowledge, there is only one work targeting the DNN executable-level threat [52], and no existing defense against bit flips in the computation libraries. However, its effectiveness is inconsistent across datasets and models—for example, on CIFAR-10, the attack still achieves a 24.8% success rate. Furthermore, since this approach relies on detecting anomalies in gradients, it is prone to false alarms when facing out-of-distribution inputs or adversarial examples.

6.1.2 Our Contributions

Driven by the above limitations, we propose `ObfusBFA`, a novel and holistic methodology to mitigate different levels of BFAs in an efficient and practical manner. The key insight behind `ObfusBFA` is based on our observation that all BFAs rely on precise identification of vulnerable bits, which constitute a very small percentage (less than 0.01%) of the total bits (§6.3.2). In contrast, flipping random bits can rarely impact the model inference [49]. Inspired by this, `ObfusBFA` aims to **degrade any carefully-orchestrated BFA into random flips, rendering the attack ineffective**. This is achieved by introducing obfuscation (i.e., random dummy operations) into the application architecture at different levels, including the model parameters, executables, and deep learning libraries. Such obfuscation has minimal impact on the inference results or speed, but alters the memory address offsets of vulnerable bits, thereby invalidating the attacker's ability to target specific bits.

Note that model-level obfuscation has been applied to mitigate side-channel attacks [98, 99]. We repurpose this concept and extend it to multiple system layers for integrity protection against BFAs.

`ObfusBFA` is designed as an end-to-end protection framework to automatically provide comprehensive protection over a given DNN application (Figure 6.2). This framework consists of three key components: (1) *Vulnerability Searcher* scans the target application, and identifies the potential vulnerable bits at different layers that are sensitive to BFAs. (2) *Obfuscation Pattern Generator* produces the randomized memory address offsets for the vulnerable bits. (3) Finally, *Obfuscation Pattern Enforcer* applies the randomized pattern to the model inference execution at runtime. A set of new algorithms are designed, ensuring the seamless integration of these modules with high efficiency and effectiveness.

In summary, `ObfusBFA` offers several key advantages compared to existing methods. First, it provides protection against both model-level and code-level BFAs on either quantized or floating-point models. Second, injecting dummy operations has zero impact on the model prediction accuracy, and minimally affects the inference latency and resource usage, making `ObfusBFA` a utility-preserving and lightweight solution. Third, `ObfusBFA` does not require model retraining or access to training data, rendering it more practical for real-world deployment. We conduct extensive experiments to evaluate `ObfusBFA` against six state-of-the-art BFAs. The results demonstrate that `ObfusBFA` can effectively mitigate BFAs across all levels with low overhead, providing robust protection without compromising the performance. Besides, it is able to thwart adaptive attackers who are aware of the defense strategy, with the randomized memory layout obfuscation.

6.2 Background and Related Works

6.2.1 Bit-Flip Attacks against DNN Models

Bit-Flip Attacks (BFAs) are hardware fault injection attacks that corrupt memory by flipping bits, often via Rowhammer [169]. In deep learning, they target critical data to compromise predictions, typically in two forms: (1) *Untargeted BFAs*, which degrade overall accuracy, and (2) *Targeted BFAs*, which manipulate outputs

for specific inputs (e.g., a chosen class or trigger) while leaving others unaffected. Both rely on bit-search algorithms to identify vulnerable locations, which are then flipped. BFAs operate at two levels:

- **Model-level Attacks.** The most common strategy flips bits in *model parameters* [16, 24–28]. TBT [26] targets critical neurons in the final layer and embeds a backdoor via bit flips. ProFlip [24] uses JSMA to identify vulnerable bits across layers. TA-LBF [28] formulates the search as a binary integer programming problem solved with ADMM.
- **Code-level Attacks.** A newer class flips bits in *executables or computation libraries*. Compilers (e.g., TVM [4], Glow [226]) translate models into binaries, which often rely on libraries like BLAS for GEMM operations. Flipping vulnerable bits in executables or libraries can hijack control flow. Chen et al.[108] showed flips in the `.text` section of executables can sharply reduce accuracy. Li et al. [56] proposed *FrameFlip*, where a single flipped bit in a jump opcode within OpenBLAS corrupts all dependent DNN applications.

6.3 Overview

6.3.1 Threat Model

In line with prior works [16, 25, 56, 178], we adopt the conventional threat model of BFAs. The attacker is an unprivileged user who shares the same physical machine as the victim's DNN application. Note that this co-location condition is a requirement for launching BFAs, not for our defense. Through techniques like Rowhammer, the attacker can flip the victim's data and code in the DRAM.

As a defense-focused work, we strategically consider a strong attacker, who has common knowledge about the victim's DNN application. (1) As the victim may employ some popular open-source DNN models, the attacker knows the DNN architecture, model weights, gradients and training data. This allows him to precisely target the flip bits within the original DNN model. (2) The victim may also use common deep learning frameworks to support the model inference. Therefore, the attacker knows the framework, dynamic libraries linked to the model, deep learning compilers and compilation settings. (3) The victim employs our `ObfusBFA` to

transform the original DNN model and code executables. The attacker knows the detailed mechanism of `ObfusBFA`, but not the random numbers generated on-the-fly by the victim. The attacker also does not have access to the transformed DNN model or code executables.

6.3.2 Design Insight

The design of our `ObfusBFA` is grounded in the following observation: **random bit-flips have minimal impact on the model inference process**, which is universal for different levels of the DNN system. Specifically, Cheng et al. [49] showed that randomly flipping bits in DNN's model weights have negligible effects on its prediction results, even when more of bits are changed. We extend this conclusion to the code level by flipping each of all conditional jumps to its opposite control flow in a compiled OpenBLAS library. Out of 154,554 conditional jumps under the `-O2` optimization flag, only 85 jumps cause a drop in model accuracy when running ResNet-50 on the ImageNet dataset, while another 148 jumps lead to crashes or timeouts. The remaining 99.8% of the jumps, despite being flipped, run without errors or performance drops. These results suggest that both DNN model weights and codebases are naturally resilient to random bit-flips in most cases.

This inspires the design of `ObfusBFA`: by introducing obfuscation into the model's inference (i.e., randomizing the memory address offsets for the vulnerable bits), we can transform an organized BFA into random bit-flips, which can effectively alleviate the attacker's impact on the model results. Following this unified principle, `ObfusBFA` delivers protection over DNN applications against BFAs at both model and code levels.

6.3.3 Discussion

It is worth noting that `ObfusBFA` fundamentally differs from existing mechanisms involving obfuscation. First, Address Space Layout Randomization (ASLR) [227] obfuscates the system's virtual memory layout whenever the code execution starts.

However, the application binary is still fixed. Therefore, researchers have demonstrated that an attacker can still reliably flip the desired bits in any physical memory page under ASLR based on his knowledge of the binary structure [176, 228]. In contrast, `ObfusBFA` hardens the *binary representations* to increase the difficulty of targeting vulnerable bit locations. By properly obfuscating the binary, it ensures that the attacker has no knowledge of the vulnerable bit locations within the binary, preventing him from accurately flipping the critical bits.

Second, software diversity [229, 230] aims to create multiple, independent implementations of the same software to enhance security, fault tolerance, and resilience against vulnerabilities. However, it is not effective in mitigating BFAs. (1) Certain software diversity methods (e.g., instruction reordering, equivalent instruction substitution) may not effectively alter memory offsets, leaving the critical bits still vulnerable. (2) DNN computation libraries are highly optimized to minimize overhead in computation-intensive applications. Applying software diversity techniques indiscriminately can disrupt these optimizations, leading to significant performance penalties. (3) Software diversity lacks sensitivity analysis for critical bits and needs to apply obfuscation broadly across the entire software project, which could incur tremendous overhead.

6.4 Methodology

6.4.1 Overall Workflow

Figure 6.2 shows the pipeline of `ObfusBFA`. It consists of three logical components: *Vulnerability Searcher*, *Obfuscation Pattern Generator*, and *Obfuscation Pattern Enforcer*.

- **Vulnerability Searcher**. This module scans the target application, and records the locations of all identified vulnerable bits. For different levels of BFAs, we design the corresponding algorithms to scan the target operation set (i.e., model weights for model-level BFAs, binaries for code-level BFAs) and identify bits that can significantly impact the model inference when flipped. Vulnerable bits that only cause a crash are not considered critical. The locations (offsets) of these vulnerable bits are recorded in a list.

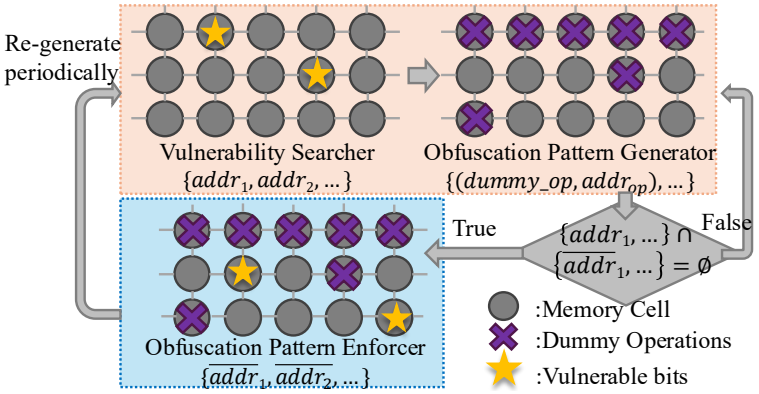


FIGURE 6.2: Overview of `ObfusBFA`.

- **Obfuscation Pattern Generator**. Based on these vulnerable bit locations, this module aims to produce the randomized memory address offsets of the target application. The detailed process is outlined in Algorithm 3. Specifically, after *Vulnerability Searcher* obtains the addresses of all vulnerable bits (Line 3), *Obfuscation Pattern Generator* processes the target system and inserts a random number of dummy operations before the critical elements (e.g., operations in the code, layers in the model architecture).

Algorithm 3 Obfuscation Pattern Generation

```
1:  Input: ElemSet, Prob                          ▷ Target set, Obfuscation Probability
2:  Output: ObsPat                                ▷ Obfuscation pattern
3:  ObsSuccess ← 0, VLoc ← getVulAddr(ElemSet)
4:  while ObsSuccess = 0 do
5:      for Elem ∈ ElemSet do
6:          if getRand ≤ Prob  or  Elem ∈ VLoc then
7:              (DummyOp, InsLoc) ← InsDummy(Op)
8:              ObsPat.append(DummyOp, InsLoc)
9:      VLocNew ← getVulAddr(ObsPat, ElemSet)
10:     if NoCommonAddr between VLoc and VLocNew then
11:         ObsSuccess ← 1
```

There are two points that need to be emphasized. First, an adaptive attacker may try to flip the bits not in the vulnerable list, attempting to bypass our defense. Such attack will be much less effective as those bits have minor impact on the model behaviors. Nevertheless, we also offer protection for all the other bits not in the list. Note that we cannot enforce the same degree of obfuscation as for the critical bits, which can terribly affect the model performance. Instead, we

insert dummy operations following a preset probability, which is typically lower to reduce the overhead. This random insertion mechanism can help further reduce the effectiveness of potential attacks targeting the bits not classified as vulnerable, making them impractical.

Second, for the generated obfuscation pattern, we run *Vulnerable Searcher* again to check the new vulnerable bits (Line 9). If there is no overlap between the new and original lists of vulnerable bit addresses, the obfuscation is considered successful. This ensures that no vulnerable bits have relocated to new addresses that are also vulnerable. Otherwise, we run *Obfuscation Pattern Generator* again until the condition is satisfied.

- **Obfuscation Pattern Enforcer.** At runtime, this module applies the generated obfuscation pattern to the model inference. Due to the insertion of dummy operations, the obfuscated operation set will have a different memory layout from the unprotected one. However, this will not affect the functionality of the model, as the inserted dummy operations have no impact on the target operation set. With the randomized memory addresses for critical operations, the attacker cannot identify the actual memory offsets of the vulnerable bits, effectively transforming any sophisticated BFA into an ineffective random bit-flip attack.

End-to-end Protection. Given a DNN system, we execute the above three components to generate the obfuscated versions of the model architecture, executable and libraries, and then launch them for robust inference. Note that a sophisticated attacker may try to reverse engineer the obfuscated code, and then adaptively identify new vulnerable bits for flipping. To mitigate this threat, we can periodically run the entire pipeline to generate and launch new obfuscated executables and libraries, e.g., every 10 minutes. Considering the long time for the attacker to flip bits (e.g., multiple hours with Rowhammer), this update frequency is sufficiently secure to mitigate the adaptive BFAs. As our defense pipeline is computationally lightweight and typically completes within a few seconds, the update has negligible impact on the overall performance. We can further launch multiple inference instances, and alternatively update them to reduce the impact on service availability.

Below we present the detailed mechanisms of each component for different levels of BFAs.

6.4.2 Vulnerability Searcher

We design new algorithms to search vulnerable bits in different levels of BFAs, respectively.

6.4.2.1 Model-level BFAs.

We employ a gradient-based ranking algorithm to identify a set of vulnerable bits in the model weights. For the weights in the i-th layer, given the loss function $\mathcal{L}$ and the weight matrix $\mathbf{W}$, we select the top-k weights from both the convolutional layers and the linear layers based on the magnitude of their gradients for the testing dataset. For convolution layers and linear layers from 1 to z in the model, the process of selecting the top-k most vulnerable weights can be expressed as:

$$\text{Top}_k \left[|| \frac{\partial \mathcal{L}}{\partial \mathbf{W}} || \right] = \text{Top}_k \left[|| g_1^{1,1}, g_i^{1,1}, \ldots, g_z^{M,N} || \right] \tag{6.1}$$

The locations of these top-k weights form the list of vulnerable locations. Since DNN models are loaded into memory in a sequential page-by-page manner, where each page has a fixed size and stores the weights contiguously, we can map the vulnerable weights directly to their index in memory from the beginning of the model. This indexing allows us to efficiently reference and protect specific weights during our defense implementation. It is important to note that this module is also applicable to real-world quantized models. During backpropagation, each layer can be temporarily dequantized to compute the gradients and then re-quantized afterward. This solution has been widely adopted in prior BFA studies [16, 24–28].

6.4.2.2 Code-level BFAs.

We aim to search for the critical jump instructions in executables and shared libraries. The process is illustrated in Algorithm 4. We focus on jump opcodes as they are highly susceptible to BFAs and flipping a single bit can invert their semantic condition (e.g., changing jump-if-equal to jump-if-not-equal), effectively reversing the control flow. Hence, they are the main target of existing code-based attack [56]. The existence of other vulnerable instructions and their sensitivity to

bit flips is not explored. To address this uncertainty, `ObfusBFA` also obfuscates the addresses of non-critical elements not in the vulnerable list, as described in §6.4.1.

Algorithm 4 Binary Automatic Vulnerability Search

Input: *Code*, *Model*, *Dataset* ▷ Dynamic library, DL inference infrastructure, Dataset
Output: *v_loc* ▷ List of vulnerable bit locations
$v\_loc \leftarrow \{\}$, $acc_{clean} \leftarrow getCleanAcc()$
for all functional unit $F \in Code$ **do**
  for all Opcode $Op \in F$ **do**
    if $Op \in CondJmpSet$ **then**
      $\overline{Op} \leftarrow flipOpcode(Op)$, $Instrument(\overline{Op})$
      $lib \leftarrow writeLibrary()$, $LibraryInterpositioning(lib)$
      $acc \leftarrow inference(model, dataset)$
      if $acc \leq acc_{clean}$ **then**
        $v\_loc \leftarrow v\_loc \cup \{v\_loc\}$
      $recover(Op)$
return *v_loc*

Specifically, we iterate through all opcodes in the binary files. If the current opcode is a conditional jump, we flip the conditional jump to its semantic opposite and generate a new library file. In the x86 instruction set, each conditional jump has a short jump and near jump version, which have different ranges and opcodes. For example, `je` has the short jump opcode `0x74` and the near jump opcode `0x0F84`. The algorithm considers both cases. We then use the instrumented binary to run inference and test accuracy. If the accuracy drops, the branch is considered vulnerable, and the corresponding bit address is recorded.

6.4.3 Obfuscation Pattern Generator

We develop new obfuscation solutions to randomize the memory offsets of the identified critical operations.

6.4.3.1 Model-level BFAs.

We introduce two strategies to obfuscate the model address layouts.

- **Inserting Dummy Layers.** This strategy involves inserting an additional computational layer immediately after the activation function φ of a given layer. The insertion of a dummy layer L_i does not affect the output of the original layer, as shown in Equation 6.2:

$$\varphi(\varphi(\mathbf{X_i} \cdot \mathbf{W_i}) \cdot L_i) = \varphi(\mathbf{X_i} \cdot \mathbf{W_i}) \tag{6.2}$$

For linear layers, the dummy layer L_i is an identity matrix of the same dimensions as the input to the layer. For convolutional layers, where the input and output have c channels and kernel size of (k_1, k_2), the dummy layer is initialized as:

$$L_i^{(a,b,d,m)} = \begin{cases} 1 & \text{if } a = b = 1 \text{ and } d = m \\ 0 & \text{otherwise} \end{cases}$$

where a, b are the indices of the row and column, d is the index of the input channel, and m is the index of the filter. We choose a kernel size of $k_1 = k_2 = 1$, which minimizes the extra computation introduced by the dummy layer.

- **Inserting Dummy Neurons.** Another obfuscation strategy is the insertion of dummy neurons into existing layers. These dummy neurons ensure that the prediction behavior of the original DNN remains unchanged after their addition.

To insert n dummy neurons into a current computation layer with weights $\mathbf{W_i^{(b,a)}}$, assuming the next computation layer has weights $\mathbf{W_{i+1}^{(c,b)}}$, then given the input $\mathbf{X_i}$, the forward process is formulated as:

$$\mathbf{W_{i+1}^{c,b}} \cdot (\mathbf{W_i^{b,a}} \cdot \mathbf{x}) = \mathbf{W_{i+1}^{c,(b+n)}} \cdot (\mathbf{W_i^{(b+n),a}} \cdot \mathbf{x}) \tag{6.3}$$

To ensure the injected neurons are non-functional (i.e.,"dummy"), their weights are set to vectors of all zeros with biases of zero. Since adding dummy neurons alters the dimensions of the current layer, the subsequent layer must also have the corresponding dummy neurons added to maintain the dimensional consistency. Specifically, for a convolution layer, we add n extra filters to the current layer, increasing the number of output channels by n. As a result, the number of input channels of the next convolution layer must also be increased by n, with the additional channels initialized to zeros. Similarly, for each linear layer, we increase the number of output features by n, and adjust the number of input features of the next linear layer by n, with the added parameters initialized to zeros.

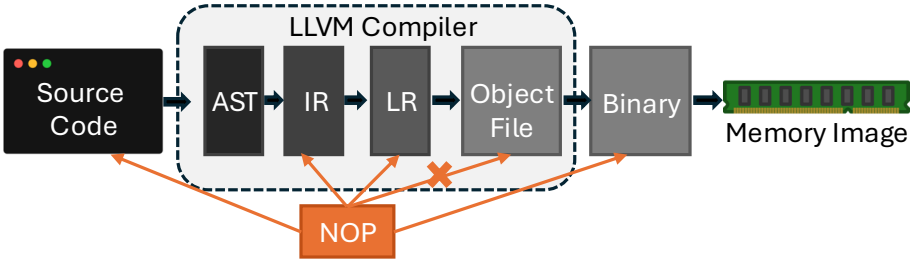


FIGURE 6.3: Possible NOP insertion phases during compiling.

6.4.3.2 Code-level BFAs.

To add obfuscation at the code level, our strategy is to insert dummy operations (e.g., *DummyOP*) before the critical instructions. The `NOP` (No Operation) instruction in `x86` architecture is a one-byte instruction that takes up space in the instruction stream but performs no useful operation. It does not modify any machine state except for advancing the `EIP` (Extended Instruction Pointer) register. As a result, after we determine that a specific location requires obfuscation, a random number of `NOP` instructions are inserted before the current opcode, thereby altering the memory layout without changing the functionality.

To implement this memory address randomization efficiently, we develop an LLVM-based backend obfuscation pass. LLVM is a suite of compiler and toolchain technologies that transform source code (e.g., C programs) into an Abstract Syntax Tree (AST), which is a structured representation. Then it is translated into intermediate representation (IR), which is independent of the source language. This IR is optimized by LLVM passes, after which it is further lowered into a lower-level representation (LR), such as GCC's Register Transfer Language (RTL). The backend then generates object files from this optimized IR. Finally, a linker combines object files into the final program binary, which the operating system's loader places in memory for execution. Figure 6.3 illustrates the compilation stages of a source file, highlighting the possible phases for `NOP` insertion.

While it is theoretically possible to insert `NOP` instructions at any stage of the compilation process, it is challenging to do so in earlier stages (e.g., at the IR level or in source code). Since the attack targets the memory image, corresponding to the final binary, it is important to ensure precise control over the location of inserted `NOP` instructions in the binary. However, locating the exact binary position for vulnerability obfuscation from the IR level is difficult due to the abstraction gap

between the IR and the final binary. Inserting `NOP` instructions at the source code level also presents challenges. Even if the mapping between the source code and binary is known, compiler optimizations can relocate or even remove the inserted code, making it difficult to control where the `NOP` instructions will eventually appear in the binary.

To overcome these challenges, we insert `NOP` instructions at the lower-level representation, specifically after the compiler performs all optimizations and just before the target code is emitted (as indicated by "x" in Figure 6.3). This step corresponds to the `addPreEmitPass2` in LLVM, ensuring that the inserted `NOP` instructions appear precisely where required in the final binary. To manage the `NOP` insertion process, we store the insertion patterns in a file, which contains both the insertion locations and the number of `NOP` instructions to be inserted. This file is then loaded by our backend pass during compilation to effectively obfuscate the memory layout.

6.4.4 Obfuscation Pattern Enforcer

6.4.4.1 Model-level BFAs

At runtime, we load the target model and apply the obfuscation patterns to disrupt the memory layout. Since the obfuscation involves only simple operations, e.g., padding zeros to the weights or inserting non-functional neurons, it is computationally efficient and incurs minimal overhead. As a result, the transformation is applied almost instantaneously, leaving very little time for an attacker to exploit the model via Rowhammer or other bit-flip techniques before the obfuscation is completed.

Moreover, due to the design of the obfuscation strategies, the obfuscated model maintains identical functionality to the original model. This ensures that the obfuscation process does not introduce any unintended changes to the model's behavior, preserving its accuracy and performance while simultaneously protecting it against memory-based attacks. Additionally, the memory layout of the model is randomized with each load, making it even more challenging for an attacker to predict the specific bit locations they need to target. This randomness further strengthens the defense.

6.4.4.2 Code-level BFAs

Once the instrumented library or executable with the altered memory layout is generated, we load them for runtime model inference. In Linux environments, when a program is loaded and executed, the dynamic linker (ld-linux.so) first searches the libraries listed in the `LD_PRELOAD` environment variable for any undefined references before searching other libraries. Therefore, we set `LD_PRELOAD` to our obfuscated versions, which can intercept and redirect function calls to utilize the new libraries. For code executables, we modify the compilation process, e.g., instruct TVM to generate LLVM bitcode, enabling our custom LLVM backend pass to apply the obfuscation techniques.

We can frequently regenerate libraries and executables to enhance protection. The compilation cost can be very small with the following optimization. Since the functions prone to BFAs are concentrated in a small subset of source files (8 in OpenBLAS), we only need to recompile these files instead of the entire project. To further reduce the compilation time, we pre-generate LLVM intermediate representation (IR) and perform binary generation from this IR, as the obfuscation is applied in the backend, specifically in the IR-to-binary translation process. Our tests show that the compilation process takes around 0.683 seconds, and generating the dynamic library adds just 0.103 seconds. This enables frequent, low-cost binary updates, maximizing the runtime protection against BFAs.

6.5 Security Analysis

In this section, we provide analysis about the security robustness of `ObfusBFA` from different perspectives.

- **Secrecy of obfuscation**. We assume that our detailed defense mechanism (e.g., algorithms, type of obfuscation primitives) is totally public and known by attackers. However, they cannot predict the exact locations or quantities of obfuscation patterns, which are randomly generated on-the-fly. This randomness in our design can ensure its security. This is analogous to the Kerckhoffs's principle in cryptographic systems, where knowing the algorithm does not compromise the security without the key.

- **Reverse-engineering the application**. Even if the attacker has no knowledge or white-box access to the obfuscated DNN application, he may adopt some reverse-engineering techniques to recover it [20], further finding the hidden vulnerable bits and then flipping them. As described in §6.4.1, we can easily block this attack path by periodically updating the application with new obfuscation patterns. Note that each update only takes less than one second, while the attacker may spend dozens of seconds to flip the bits with RowHammer [16]. Such imbalanced timing cost makes this adaptive attack impractical.

- **Targeting a different set of bits**. Theoretically, our *Vulnerability Searcher* can identify majority of vulnerable bits, and attacker's identified bits highly likely fall within this set. Even if the attacker selects a different set of bits to bypass the defense (the corresponding attack will be much less effective), our design (Algorithm 3) ensures obfuscation is applied probabilistically throughout the entire codebase or model weights. It will largely alter the memory offsets to thwart the attack.

- **Flip bits of dummy neurons**. It is possible that attackers may flip bits of the inserted dummy neurons, which could affect the model behaviors. However, this situation is very rare as the inserted dummy weights constitute a negligible fraction of the total weights. In our evaluations, this never occurred. To further mitigate this risk, we can refine the obfuscation algorithm to exert finer control over insertion locations, minimizing the likelihood of such occurrences.

- **Side-channel leakage**. Obfuscation may alter GPU alignment, raising concerns about timing side-channel leakage. However, attackers lack a baseline profile to determine whether long execution times stem from alignment effects or naturally larger layers. Even if they suspect a layer is obfuscated, randomness introduced by `ObfusBFA` significantly enlarges the search space. In fact, model obfuscation has been employed in prior works [98, 99] against side-channel-based model extraction attacks. This further supports that the obfuscation of `ObfusBFA` will not enlarge the attack surface.

- **Generalization to other ISAs.** Although our implementation for the code-level obfuscation mainly focuses on the x86 ISA, `ObfusBFA` can be easily extended to other ISAs. First, the `NOP` operation is universally available across most ISAs for memory alignment and timing purposes. Second, since `NOP` insertion is performed

via an LLVM backend pass, and LLVM supports a wide range of ISAs, adapting `ObfusBFA` to another ISA requires minimal modifications.

- **Crashes and timeouts**. After applying `ObfusBFA`, bit-flip attacks may occasionally cause the target application to crash or timeout. Such events were rare in our evaluation and are not considered successful attacks, as these outcomes are easily detectable by the victim and are generally classified as denial-of-service (DoS) behaviors, which lie outside our threat model. This treatment is consistent with prior work [56], which also distinguishes between stealthy compromises and overt disruptions.

6.6 Evaluation

6.6.1 Experimental Setup

Implementation. Our experiments were conducted on an NVIDIA GeForce RTX 3090 GPU with 24GB of memory, used for deep learning model training. The models were deployed on a server powered by an AMD EPYC 7513 32-Core Processor for inference. The DNN models and our defense are implemented with PyTorch version 1.13 with CUDA 11.6, compiled using LLVM version 12.0.1.

Attacks under Evaluation. For *code-level BFAs*, we assess the effectiveness of `ObfusBFA` against two state-of-the-art, untargeted attacks, which are the only known attacks of their kind to date. (1) *FrameFlip* [56] targets the BLAS library by flipping conditional jumps in LLVM bitcode via an `XOR` operation at the IR level. We used PyTorch version 2.3.0 with OpenBLAS version 0.3.20. To better reflect real-world threats, we tested our defense on a C++-based inference infrastructure, as used in [56], to demonstrate its applicability beyond Python-based frameworks. (2) The attack in [108] targets the TVM compiler by manipulating vulnerable bits in DL executables to degrade overall model accuracy. We adopt the TVM version 0.18.dev0, and choose the `-O3` optimization flag. We also include adaptive versions of both attacks to evaluate `ObfusBFA`'s robustness under adversarial awareness.

For *model-level BFAs*, we evaluate `ObfusBFA` against one untargeted attack [25], and three targeted attacks: T-BFA[27], TBT [26], and TA-LBF [28]. As with the code-level setting, we also consider adaptive variants of these attacks.

Datasets and Models. Our solution is general over different tasks. Following common selections in existing attacks [16, 24–28, 56, 108] and defenses [44, 46–50, 178], we evaluate `ObfusBFA` on three popular datasets: CIFAR-10 [231], German Traffic Sign Recognition Benchmark (GTSRB) [232], and ImageNet ILSVRC-2012 subset [233].

We choose popular network architectures that are widely used for image classification tasks, including VGG-16 [234], ResNet-20, ResNet-32, ResNet-34, and ResNet50 [235].

Baselines. For *code-level BFAs*, we compare our `ObfusBFA` with the only known defense BitShield [52]. For *model-level BFAs*, we compare `ObfusBFA` against several state-of-the-art defenses targeting both untargeted and targeted bit-flip attacks on model weights. Specifically, for untargeted BFAs, we choose DeepShuffle [48], NeuroPots [178], HammerDodger [49], and RREC [50]. For targeted BFAs, we choose BIN [46], RA-BNN [47], and Aegis [44].

6.6.2 Results for Code-level BFAs

TABLE 6.1: Model utility evaluation (DL infra.)

DL Framework	Dataset-Model	Base Acc	ΔAcc (%)
C++ (O1–O3)	CIFAR10-ResNet20	97.00	0
	GTSRB-VGG16	95.31	0
	ImageNet-ResNet50	91.67	0
PyTorch (O1–O3)	CIFAR10-ResNet20	81.25	0
	GTSRB-VGG16	93.81	0
	ImageNet-ResNet50	87.50	0

Model Utility Evaluation. A key metric for evaluating any defense is its ability to preserve the model's utility, particularly in terms of accuracy (Acc). Tables 6.1 and 6.2 report the accuracy results after applying `ObfusBFA` across different DL frameworks (C++-based and PyTorch-based) with the TVM compiler. Since we observe no accuracy difference across optimization levels (O1–O3), we merge the results for simplicity in Table 6.1. The "Base Acc" column shows the model's accuracy without `ObfusBFA`, while the "ΔAcc" column represents the change in accuracy after applying `ObfusBFA`. The "Flag" column indicates the optimization flag used during library compilation. As these attacks run on CPUs and can be

TABLE 6.2: Model utility evaluation (DL compiler TVM)

Dataset-Model	Base Acc (%)	ΔAcc (%)
CIFAR10-ResNet20	80.50	0
GTSRB-VGG16	98.50	0
ImageNet-ResNet50	85.00	0

time-consuming, we use a subset of each dataset during evaluation. The results demonstrate that `ObfusBFA` does not degrade model performance, with no observable accuracy loss across all datasets and models tested.

Mitigating Untargeted Attacks. For untargeted BFAs, we measure the average accuracy drop across the identified vulnerable bit set. Tables 6.3, 6.4, and 6.5 report the number of identified vulnerable bits (#Vuln. Bits), the original model accuracy (before the attack), and the accuracy both without and with the defense applied to the vulnerable bit set. These tables demonstrate that `ObfusBFA` successfully restores the model to its original accuracy for both C++-based and PyTorch-based DL infrastructures under the *FrameFlip* attack, and for the TVM compiler under the attack in [108].

TABLE 6.3: Mitigating untargeted attacks (C++ infra.)

Flag	Model	#Vuln Bits	Acc (%)		
			Base	w/o. Def	w. Def
O1	ResNet20	80	97.00	33.90	97.00
	VGG16	61	95.31	18.00	95.31
	ResNet50	81	91.67	29.10	91.67
O2	ResNet20	85	97.00	36.50	97.00
	VGG16	64	95.31	22.50	95.31
	ResNet50	85	91.67	31.20	91.67
O3	ResNet20	85	97.00	36.20	97.00
	VGG16	63	95.31	23.50	95.31
	ResNet50	85	91.67	31.60	91.67

Adaptive Attacks. We consider an adaptive attacker aware of our defense mechanism but not the on-the-fly obfuscated parameters. The attacker may attempt to flip additional bits around each vulnerable address to bypass `ObfusBFA`, considering alignment effects enforced by the compiler. The flipping range is defined as: Since `ObfusBFA` introduces randomness into the memory layout obfuscation, the

TABLE 6.4: Mitigating untargeted attacks (PyTorch infra.)

Flag	Model	#Vuln Bits	Acc (%)		
			Base	w/o. Def	w. Def
O1	ResNet20	34	81.25	23.67	81.25
	VGG16	30	95.31	6.88	95.31
	ResNet50	28	87.50	5.58	87.50
O2	ResNet20	34	81.25	25.14	81.25
	VGG16	34	95.31	11.81	95.31
	ResNet50	29	87.50	6.25	87.50
O3	ResNet20	34	81.25	25.18	81.25
	VGG16	31	95.31	8.06	95.31
	ResNet50	28	87.50	7.14	87.50

TABLE 6.5: Mitigating untargeted attacks (TVM Compiler)

Model	#Vuln. Bits	Acc (%)		
		Base	w/o. Def.	w. Def.
ResNet20	94	80.50	17.98	80.50
VGG16	112	98.50	16.83	98.50
ResNet50	72	85.00	21.25	85.00

attacker's only viable option to circumvent it is to flip more bits. To this end, he flips not only the bits in the identified vulnerable bit list but also adjacent bits around each vulnerable bit address *addr*. Given that code alignment is typically enforced by compilers (e.g., aligning instructions to 16-byte boundaries for efficient CPU instruction fetching), `ObfusBFA` may cause $16n$-byte latencies in subsequent function. Therefore, the attacker can flip all bits in the following range for each vulnerable bit address *addr*:

$$[addr - x_1 + x_2 \cdot al, addr + x_1 + x_2 \cdot al] \tag{6.4}$$

where al represents the alignment size set during compilation, and x_1 and x_2 are parameters that control the flipping range. The larger value results in a broader range of bit flips.

We conducted such adaptive attack on all three dataset-model configurations using the compile flags `-O1`, `-O2`, and `-O3` on the C++ deep learning framework and the DL compiler TVM. For each vulnerable location *addr*, we set the alignment size

al to 16, and evaluate x_2 in the range [0, 1, 2] for each evaluation, while x_1 takes values from the set {5, 10, 15, 20, 25, 30, 35, 40, 45} at each time. We observed an increase in the percentage of program timeouts or crashes, but no accuracy drop was detected across any experimental setting. These results demonstrate that even when an attacker is aware of our defense mechanism, he is unable to bypass it. The adoption of randomized memory address obfuscation turns carefully targeted bit flips into random flips, neutralizing the attacker's adaptive strategy and preserving model accuracy.

Overhead. A key objective of `ObfusBFA` is to minimize overhead. We evaluate time, storage, and memory costs, averaged over 10 inference runs. Time overhead is measured as the increase in inference latency per image, storage as the growth in dynamic library size from inserted `nop` instructions, and memory as the increase in per-image consumption.

Tables 6.6 and 6.7 show the results. Across models and frameworks, `ObfusBFA` incurs negligible overhead: time overhead is mostly small (some even benefit from micro-optimizations, while ImageNet-ResNet50 under `-O3` increases by 5.02%), storage growth is marginal (≤0.37% in C++ and within range in TVM), and memory fluctuations remain minor (up to 1.69% in C++ and 2.23% in TVM). Overall, `ObfusBFA` preserves performance with minimal resource costs, ensuring practicality for deployment.

Table 6.6: Overhead for C++ infra.

Flag	Model	Insert Count	ΔTime (ms/%)	ΔStorage (KB/%)	ΔMemory (KB/%)
O1	ResNet20	63636	+2(3.04%)	+47(0.37%)	-13(-0.46%)
	VGG16	34664	-1(-1.08%)	+27(0.22%)	+4(1.00%)
	ResNet50	37085	+14(1.31%)	+27(0.22%)	+195(1.00%)
O2	ResNet20	55667	-8(-13.2%)	+39(0.29%)	+30(1.05%)
	VGG16	4939	-4(-3.13%)	+4(0.03%)	-7(-1.67%)
	ResNet50	33494	-42(-3.90%)	+23(0.17%)	-1,132(-6.01%)
O3	ResNet20	56393	-8(-12.8%)	+39(0.26%)	+67(2.36%)
	VGG16	5867	-2(-1.26%)	+4(0.03%)	+7(1.69%)
	ResNet50	5272	+51(5.02%)	+4(0.03%)	-153(-0.80%)

Comparison. We compare `ObfusBFA` with BitShield [52], the only existing defense at the DNN executable level. We use the ResNet50 model on ImageNet for the BFA against the TVM compiler. The comparison results are shown in

TABLE 6.7: Overhead for TVM compiler.

Model	Insert Count	ΔTime (ms/%)	ΔStorage (KB/%)	ΔMemory (KB/%)
ResNet20	39404	+398(2.79%)	+31(7.58%)	+12(0.67%)
VGG16	31248	+1720(4.45%)	+39(10.69%)	+173(2.23%)
ResNet50	9989	-743(-0.55%)	+8(0.69%)	+6(0.18%)

TABLE 6.8: Comparison with Bitshield

Defense	ΔAcc (%)	ΔTime (%)	Mitigation Rate(%)
Bitshield	0	8.22	97.9
`ObfusBFA`	0	**-0.55**	**100**

Table 6.8. Both defenses maintain the model's original accuracy (ΔAcc = 0%). However, BitShield introduces a notable 8.22% runtime overhead, while `ObfusBFA` incurs a slight performance improvement of -0.55%, possibly due to compiler-level code alignment optimizations introduced by obfuscation. In terms of mitigation effectiveness, `ObfusBFA` completely thwarts the attack with a 100% mitigation rate, while BitShield achieves 97.9%. These highlight that `ObfusBFA` not only offers superior protection but also improves efficiency in certain cases.

6.6.3 Results for Model-level BFAs

Model Utility Evaluation. We applied `ObfusBFA` to model-level BFAs and evaluated its impact on model functionality, specifically in terms of accuracy. The results, shown in the Table 6.9, indicate that `ObfusBFA` barely degrades the performance of the original model.

TABLE 6.9: Model utility evaluation after applying `ObfusBFA`.

Dataset-Model	Base Acc (%)	ΔAcc (%)
CIFAR10-ResNet20	85.36	+0.01
CIFAR10-ResNet32	84.38	+0.01
CIFAR10-ResNet34	85.70	0
ImageNet-ResNet50	75.84	0
GTSRB-VGG16	84.26	-0.02

It is worth noting the microscopic variations (e.g., +0.01% or −0.02%) in model-level obfuscation compared to the strictly 0% change observed in code-level obfuscation (Table 6.1). In code-level protection, the insertion of `NOP` instructions strictly alters memory layouts without changing tensor dimensions or mathematical operations, yielding bit-wise identical outputs. Conversely, model-level protection inserts dummy neurons (zeros) and dummy layers (identity matrices). While mathematically equivalent, the altered tensor dimensions trigger different optimized parallel reduction trees and matrix multiplication kernels in GPU libraries (e.g., cuDNN). Because floating-point addition lacks strict associativity, the changed execution order introduces tiny rounding errors. When accumulated over deep networks, these microscopic deviations can occasionally shift the final `argmax` prediction for a borderline sample (e.g., a 0.01% variance corresponds to just 1 out of 10,000 test images flipping its predicted class). This confirms that `ObfusBFA` mathematically preserves model utility, with minor variations strictly attributable to standard floating-point non-associativity.

Mitigating Untargeted Attacks. We assess the effectiveness of `ObfusBFA` in mitigating the untargeted BFA [25]. The results in Table 6.10 show a significant accuracy drop for models without `ObfusBFA`'s protection (the *w/o. Def.* column). Notably, the ImageNet-ResNet50 model's accuracy is reduced to nearly zero. However, after applying `ObfusBFA` (the *w. Def.* column), all models demonstrate resilience, maintaining accuracy levels close to their original accuracy (*Base* in the table). These results confirm that `ObfusBFA` effectively mitigates untargeted attacks across a range of models.

Table 6.10: Mitigating untargeted attacks.

Model	#Flipped Bits	Acc (%)		
		Base	w/o. Def.	w. Def.
ResNet20	30	85.36	11.46	82.53
ResNet32	14	84.38	11.81	82.00
ResNet34	30	85.70	19.84	83.63
ResNet50	10	75.84	0.10	75.85
VGG16	30	84.26	18.14	84.39

Mitigating Targeted Attacks. Targeted attacks differ from untargeted BFAs in that they seek to induce specific misclassifications without significantly impacting the overall model performance. To assess the effectiveness of `ObfusBFA` in defending

against targeted BFAs, we measure the Attack Success Rate (ASR), which indicates the percentage of successful misclassifications made by the attacker.

We replicate the results using the open-sourced code and parameters recommended by the respective authors for each attack. Since TA-LBF functions on a per-image basis, we report the average number of flipped bits across images in the test dataset. Table 6.11 presents the ASR before (w/o. Def. column) and after applying `ObfusBFA` (w. Def. column), along with the number of bits flipped during the attacks. `ObfusBFA` significantly reduces the ASR across all three types of targeted attacks. Particularly, in high-complexity datasets like ImageNet, it reduces the ASR to near-zero for several attack types, underscoring its effectiveness in mitigating targeted misclassifications while preserving overall model accuracy.

TABLE 6.11: Mitigating targeted attacks.

Attack	Model	#Flipped Bits	ASR (%)	
			w/o. Def.	w. Def.
T-BFA	ResNet20	6	99.84	8.16
	ResNet32	3	99.67	21.4
	ResNet34	20	99.83	10.43
	ResNet50	7	99.99	0.11
	VGG16	20	100	1.81
TBT	ResNet20	97	93.99	2.22
	ResNet32	147	76.78	8.11
	ResNet34	130	77.49	6.04
	ResNet50	131	100	0
	VGG16	126	98.69	4.89
TA-LBF	ResNet20	9.19	100	1.5
	ResNet32	10.92	100	1.9
	ResNet34	19.45	100	1.5
	ResNet50	12.76	100	0
	VGG16	18.42	100	2.7

Adaptive Attacks. We consider a sophisticated attacker aware of `ObfusBFA`'s obfuscation techniques (dummy layers/neurons) but unaware of exact locations. To bypass `ObfusBFA`, the attacker flips additional bits around each vulnerable address $addr$ in the range $[addr - x, addr + x]$.

Tables 6.12 and 6.13 show that `ObfusBFA` remains robust: for untargeted attacks, accuracy is largely preserved, and for targeted attacks, the attack success rate

(ASR) stays very low. The defense's effectiveness arises from most parameters having negligible impact when flipped and the random placement of dummy operations, making brute-force flipping ineffective. Targeted attacks require precise bit selection, which is thwarted by `ObfusBFA`'s randomization.

TABLE 6.12: Evaluation for untargeted adaptive attack

Model	Acc (%)				
	x=3	x=5	x=7	x=9	x=11
ResNet20	76.11	80.20	82.14	82.14	82.14
ResNet32	75.05	76.62	76.24	75.97	75.64
ResNet34	85.29	82.41	81.73	83.32	83.13
ResNet50	74.36	75.84	75.83	75.83	75.87
VGG16	82.74	80.55	80.21	82.58	82.59

TABLE 6.13: Evaluation for targeted adaptive attack

Attack	Model	ASR (%)				
		x=3	x=5	x=7	x=9	x=11
T-BFA	ResNet20	0	0	0	0	0
	ResNet32	1.83	1.83	1.83	1.83	1.83
	ResNet34	22.9	22.9	22.9	22.9	22.9
	ResNet50	0.11	0.11	0.11	0.11	0.11
	VGG16	1.55	1.55	1.55	1.55	1.55
TBT	ResNet20	2.21	2.22	2.21	2.21	2.21
	ResNet32	8.14	8.13	8.15	8.12	8.18
	ResNet34	6.04	6.04	6.04	6.04	6.04
	ResNet50	0	0	0	0	0
	VGG16	4.89	4.89	4.89	4.89	4.89
TA-LBF	ResNet20	1.5	0.99	1.98	1.98	1.98
	ResNet32	1.9	0.5	0.99	0.99	0.99
	ResNet34	1.5	1.49	2.97	1.98	1.98
	ResNet50	0	0	0	0	0
	VGG16	2.98	2.98	1.98	1.98	1.98

Moreover, a stronger attacker attempting side-channel analysis to distinguish dummy from real operations is impractical. `ObfusBFA` obfuscates both model and code levels, so observed side-channel variations cannot be attributed to a specific level (i.e, dummy layer, neuron, or operation). Any observed changes could equally stem from obfuscation in either source. Additionally, periodic refresh of random patterns renders any inferred layout quickly obsolete, making side-channel attacks ineffective.

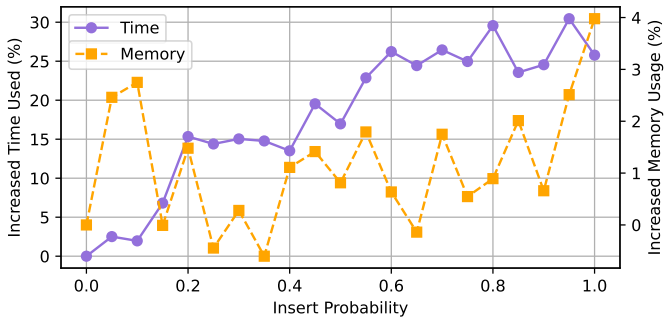


FIGURE 6.4: Time and memory Overhead.

Overhead. The time and memory overhead incurred by `ObfusBFA` is shown in Figure 6.4, where "insert probability" denotes the likelihood of inserting dummy layers and neurons. We measure the percentage increase in both the time and memory required to run the CIFAR-10 test dataset using ResNet20, compared to an unobfuscated model. The reported values are averaged over 100 runs across all images in the test dataset. While `ObfusBFA` incurs some time and memory overhead, it remains relatively low and acceptable. Notably, our experiments suggest that an insert probability below 0.3 is sufficient in most cases, effectively minimizing the overhead without compromising the defense effectiveness.

Comparison. First, we evaluate `ObfusBFA` against four untargeted defense schemes. The evaluation is performed using the ResNet20 model on CIFAR-10, and the results are presented in Table 6.14. We compare these defenses in terms of ΔAcc *(%)* (change in model accuracy after applying the defense), $\Delta Time$ *(%)* (execution time overhead introduced by the defense), $\Delta Rounds$ *(%)* (increase in the number of attack rounds required for the attacker to achieve the same effect post-defense) and *Mitigation Rate (%)* (percentage of the attack successfully mitigated). Since the source code for some of these schemes is unavailable, we rely on the data reported in their respective papers. The "-" symbol in the table indicates that a result was not reported in the corresponding paper. We observe that `ObfusBFA` introduces no accuracy loss (+0.01%) and exhibits a moderate time overhead (+6.6%). More notably, it significantly increases the number of attack rounds (160,831%) required for successful bit-flip manipulation with 100% mitigation rate, vastly outperforming all other methods in terms of resilience against untargeted BFAs.

Next, we compare `ObfusBFA` with three targeted defense schemes. We adopt the ResNet32 model on CIFAR-10. The results are presented in Table 6.15, reporting the ΔAcc *(%)* (the change in accuracy after applying the defense) and *ASR (%)*

TABLE 6.14: Comparison with untargeted defense schemes

Defense	ΔAcc (%)	ΔTime (%)	ΔRounds (%)	Mitigation Rate(%)
DeepShuffle	-	2.5	571	-
Neuropots	-1	9.7	-	93.3
HammerDodger	-	**0.8**	-	-
RREC	**+0.03**	5.8	1,452	-
`ObfusBFA`	+0.01	6.6	**160,831**	**100**

TABLE 6.15: Comparison with targeted defense schemes

Defense	Δ Acc (%)	ASR (%)	
		TBT	TA-LBF
BIN	-2.26	94.8	100
RA-BNN	-1.71	74.5	100
Aegis	-1.26	19.9	6.3
`ObfusBFA`	**+0.01**	**8.11**	**1.9**

(the percentage of successful attacks in both TBT and TA-LBF attack scenarios). We observe that `ObfusBFA` achieves the best performance, with a minimal accuracy drop (+0.01%) and significantly lower ASR in both TBT (8.11%) and TA-LBF (1.9%) scenarios. This indicates that `ObfusBFA` effectively reduces the success rate of targeted BFAs while better preserving accuracy than other defenses.

6.6.4 Discussion: Generalizability and Deployment Considerations

Generalizability. `ObfusBFA` is designed to span multiple abstraction levels, protecting both high-level model parameters and low-level codebase artifacts across mainstream infrastructures. Its core defense principle—degrading the precise targeting required by bit-flip attackers through the injection of randomized dummy operations, dummy neurons, and memory layout randomization—does not rely on a particular model family or hardware device. Because the underlying mechanics exploit the universal necessity of accurate bit localization in targeted BFAs, the methodology is inherently platform-agnostic. Furthermore, the code-level obfuscation is implemented via an LLVM backend pass. Since the LLVM compiler infrastructure supports a vast array of Instruction Set Architectures (ISAs)—ranging

from commodity processors to custom hardware AI accelerators—`ObfusBFA` extends easily to diverse computing environments. As long as the target platform supports `NOP`-like operations for memory alignment and utilizes an LLVM-based toolchain, our framework can be seamlessly generalized with minimal modifications.

Practical Deployment Considerations. While our evaluation demonstrates that `ObfusBFA` incurs low computational and storage overhead at runtime, achieving this holistic protection introduces deployment costs that must be critically assessed. First, deploying `ObfusBFA` requires deep cross-stack integration. The defender must embed the *Obfuscation Pattern Enforcer* into the build pipeline, which involves manipulating both high-level model configurations and low-level LLVM compiler backends. This dual-layer requirement significantly increases the initial integration effort compared to isolated, single-layer defenses. Second, adaptation to evolving software stacks poses an ongoing challenge. Modern deep learning frameworks and compilers undergo rapid iteration. Updates to the surrounding software stack (e.g., new versions of OpenBLAS, TVM, or LLVM) may alter the intermediate representation (IR), code generation rules, or default memory layouts. Consequently, the custom enforcement passes must be continuously monitored and potentially re-engineered to maintain compatibility with new software releases. Finally, long-term maintenance requires a persistent commitment. The *Vulnerability Searcher* profiles bit-flip sensitivity based on a highly specific model state and compilation profile. Therefore, any change to the model architecture, quantization scheme, or runtime environment necessitates a complete re-execution of the vulnerability profiling and pattern generation steps. To mitigate these maintenance burdens in long-lived deployments, future efforts should focus on tracking upstream compiler interfaces closely and fully automating the re-profiling and enforcement steps within standard Continuous Integration/Continuous Deployment (CI/CD) pipelines.

6.7 Summary

We introduce `ObfusBFA`, an efficient and effective approach to mitigating the growing threat of BFAs targeting both model and code levels of DNN applications. By leveraging a novel obfuscation strategy that inserts randomized dummy operations,

`ObfusBFA` effectively complicates the attacker's ability to locate and manipulate vulnerable bits, turning orchestrated attacks into random bit flips. Our evaluation shows `ObfusBFA`'s robustness against both untargeted and targeted BFAs across a range of datasets and architectures. It also brings minimal computational and storage overhead, making it a practical solution for real-world applications.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

This thesis has explored the critical intersection of deep learning security and underlying hardware implementations. As deep learning systems are increasingly deployed across heterogeneous platforms, they inherit low-level vulnerabilities that traditional software-level defenses often overlook. Through a systematic investigation of side-channel leakages and fault-induced bit flips, this work has demonstrated that hardware-oriented threats are not only practical but can be automated and scaled to compromise model confidentiality, availability, and integrity.

To address these challenges, this thesis made four primary contributions, alternating between offensive analysis and defensive framework design:

- **Automated Remote Side-Channel Extraction:** We introduced `Mercury`, an end-to-end framework capable of recovering DNN architectures from noisy, remote power side-channel traces. By formulating the attack as a sequence-to-sequence learning problem and leveraging TDC-based sensors on multi-tenant FPGAs, `Mercury` demonstrated that complex model architectures can be reconstructed with high accuracy without physical access or manual feature engineering.
- **Bit-Flip Inference Cost Attacks:** Shifting focus to availability threats, we presented `BitHydra`, the first bit-flip attack explicitly targeting the inference

cost of LLMs. We revealed that by flipping a small set of critical bits in the output embedding layer (specifically related to the <EOS> token), an attacker can persistently force the model to generate maximum-length responses for all users. This highlights a scalable denial-of-service vector that bypasses traditional input-filtering defenses.

- **Unified Defense against Side-Channels:** To counter ML-assisted side-channel analysis, we proposed AdvProtego, a unified framework offering protection across diverse deep learning platforms. By formulating two distinct defense goals and adapting adversarial attack strategies for each, AdvProtego injects carefully crafted execution noise that minimizes the attacker's extraction utility while maintaining the victim model's performance. This approach proved effective across FPGA power, CPU cache, and GPU memory side-channels.

- **Holistic Defense against Bit-Flips:** Finally, we developed ObfusBFA, a comprehensive defense mechanism against versatile bit-flip attacks. Unlike prior fragmented solutions, ObfusBFA protects both high-level model parameters and low-level binary code (executables and libraries). By introducing randomized dummy operations and neurons, ObfusBFA effectively transforms sophisticated, targeted bit-flip attacks into ineffective random noise, ensuring system robustness with minimal runtime overhead.

In summary, this thesis advances the understanding of the hardware-software security interface in AI applications. It establishes that securing modern deep learning systems requires a holistic view that accounts for physical leakages, memory fault vulnerabilities, and the complex interaction between model parameters and their compilation into machine code.

7.2 Broader Implications and Generalizability

Broader implications. Taken together, the core implication of this thesis is that deep learning security can no longer be treated as a purely algorithmic or software-level concern. By demonstrating that the physical substrate—through side-channel leakages (Mercury) and memory faults (BitHydra)—constitutes a highly practical

attack surface, and that this same substrate can be manipulated for robust defense (`AdvProtego`, `ObfusBFA`), this work establishes the necessity of a hardware-software co-design approach to AI security.

Generalizability to future platforms. The evaluations in this thesis are necessarily carried out on specific, representative platforms (an FPGA-hosted NVDLA accelerator, commodity CPUs and GPUs, and mainstream frameworks and compiler backends). We therefore comment on how the proposed methods are expected to transfer beyond these settings. The underlying principles are largely platform-agnostic: architecture leakage through execution-dependent side-channels, and the sensitivity of a small number of critical bits, arise from properties shared by a broad class of accelerators and models rather than from any single implementation. Consequently, the *methodology* of each contribution, formulating extraction as sequence-to-sequence learning, targeting the `<EOS>`-related parameters, learning execution noise as an adversarial-optimization problem, and obfuscating vulnerable bits across abstraction levels, is expected to carry over to emerging AI accelerators (e.g., next-generation GPUs, NPUs, and TPUs), evolving deep learning frameworks, and new deployment models such as serverless and confidential-computing inference. What is platform-specific, and would need to be re-instantiated when porting to a new environment, is the concrete interface layer: the sensor or fault primitive used to observe or perturb the hardware, the surrogate model or profiling data used to capture a new platform's leakage characteristics, and the compiler or framework hooks used to enforce a defense. We regard characterizing these interfaces on a wider range of future architectures, and quantifying how much re-tuning each requires, as an important direction for consolidating the generality of the frameworks proposed here.

7.3 Future Work

While this thesis provides robust mechanisms for attacking and defending deep learning systems, the rapid evolution of AI hardware and model architectures presents new avenues for research. The following directions are proposed for future investigation:

- **Remote Weight Recovery via Side-Channels:** While `Mercury` successfully recovered macroscopic model architectures, extracting precise weight values remains an open challenge due to the low fidelity and extreme environmental noise of remote on-chip sensors. Future work must advance on two fronts to capture microscopic execution states. On the hardware side, exploring high-granularity sensing primitives and multi-channel fusion—combining power, timing, and microarchitectural events—can fundamentally enhance physical signal resolution. On the software side, researchers can leverage emerging time-series Foundation Models and self-supervised representation learning to autonomously decode these complex, noisy physical leakages into exact weight quantization bins, paving the way for fully automated parameter extraction.

- **Security of Multimodal and Retrieval-Augmented Systems:** `BitHydra` focused on text-based autoregressive LLMs. However, the rise of Multimodal Large Language Models (MLLMs) and Retrieval-Augmented Generation (RAG) systems introduces new attack surfaces. Future research should investigate how bit-flips or side-channels impacts the alignment between visual and textual modalities, or how they might compromise the retrieval logic in RAG pipelines to induce hallucinations or leakage of retrieved contexts.

- **Information Extraction from Agentic Systems:** The rapid development of LLM-based autonomous agents opens a novel attack surface. Future work should investigate how microarchitectural side-channels (e.g., cache access patterns, timing analysis) can be exploited to extract sensitive internal states from agentic systems, including system prompts, tool-use parameters, and private memory contents.

Bibliography

[1] Cong Hao, Yao Chen, Xinheng Liu, Atif Sarwari, Daryl Sew, Ashutosh Dhar, Bryan Wu, Dongdong Fu, Jinjun Xiong, Wen-mei Hwu, et al. NAIS: Neural architecture and implementation search and its applications in autonomous driving. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, 2019. 1

[2] Gizem Gulfidan, Hande Beklen, and Kazim Yalcin Arga. Artificial intelligence as accelerator for genomic medicine and planetary health. *OMICS: A Journal of Integrative Biology*, 25(12):745–749, 2021. 1

[3] Hamza Khan, Asma Khan, Zainab Khan, Lun Bin Huang, Kun Wang, and Lei He. NPE: An FPGA-based overlay processor for natural language processing. *arXiv preprint arXiv:2104.06535*, 2021. 1

[4] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. TVM: An automated end-to-end optimizing compiler for deep learning. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 578–594, 2018. 1, 112

[5] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014. 2, 44, 89

[6] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik, and Ananthram Swami. Practical black-box attacks against deep learning systems using adversarial examples. *arXiv preprint arXiv:1602.02697*, 2016. 2

[7] Yiming Li, Yong Jiang, Zhifeng Li, and Shu-Tao Xia. Backdoor learning: A survey. *IEEE Transactions on Neural Networks and Learning Systems*, 2022. 2

[8] Xiaobei Yan, Xiaoxuan Lou, Guowen Xu, Han Qiu, Shangwei Guo, Chip Hong Chang, and Tianwei Zhang. MERCURY: An automated remote side-channel attack to Nvidia deep learning accelerator. In *International Conference on Field-Programmable Technology (FPT)*, pages 188–197, 2023. 2, 25, 81, 83, 85, 93, 97, 98

[9] Xing Hu, Ling Liang, Shuangchen Li, Lei Deng, Pengfei Zuo, Yu Ji, Xinfeng Xie, Yufei Ding, Chang Liu, Timothy Sherwood, et al. DeepSniffer: A DNN model extraction framework based on learning architectural hints. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020. 2, 15, 28, 31, 80, 83, 84, 85, 97

[10] Xiaoxuan Lou, Shangwei Guo, Jiwei Li, Yaoxin Wu, and Tianwei Zhang. NASPY: Automated extraction of automated machine learning models. In *International Conference on Learning Representations (ICLR)*, 2021. 2, 15, 28, 31, 40, 80, 81, 83, 84, 85, 94, 97, 98

[11] Xiaobei Yan, Yiming Li, Hao Wang, Han Qiu, and Tianwei Zhang. BitHydra: Towards bit-flip inference cost attack against large language models. *arXiv preprint arXiv:2505.16670*, 2025. 2

[12] Jean-Max Dutertre and Jessy Clédière. An introduction to fault injection attacks. *Embedded Cryptography*, 1:215, 2025. 2

[13] Adnan Siraj Rakin, Yukui Luo, Xiaolin Xu, and Deliang Fan. Deep-Dup: An adversarial weight duplication attack framework to crush deep neural network in multi-tenant FPGA. In *USENIX Security Symposium*, pages 1919–1936, 2021. 2, 17, 108

[14] Jakub Breier, Xiaolu Hou, Dirmanto Jap, Lei Ma, Shivam Bhasin, and Yang Liu. Practical fault attack on deep neural networks. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2204–2206, 2018. 2, 17, 108

[15] 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. 2, 17, 108

[16] Fan Yao, Adnan Siraj Rakin, and Deliang Fan. DeepHammer: Depleting the intelligence of deep neural networks through targeted chain of bit flips. In *USENIX Security Symposium*, pages 1463–1480, 2020. 2, 3, 17, 18, 20, 49, 54, 57, 108, 112, 117, 123, 125

[17] Kota Yoshida, Takaya Kubota, Mitsuru Shiozaki, and Takeshi Fujino. Model-extraction attack against FPGA-DNN accelerator utilizing correlation electromagnetic analysis. In *IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2019. 2, 3, 7, 14, 17, 20, 26, 27, 28, 83

[18] Ge Li, Mohit Tiwari, and Michael Orshansky. Power-based attacks on spatial DNN accelerators. *arXiv preprint arXiv:2108.12579*, 2021. 13, 14, 17, 20, 27, 83

[19] Honggang Yu, Haocheng Ma, Kaichen Yang, Yiqiang Zhao, and Yier Jin. DeepEM: Deep neural networks model recovery through EM side-channel information leakage. In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2020. 3, 13, 14, 27, 28, 83

[20] Weizhe Hua, Zhiru Zhang, and G Edward Suh. Reverse engineering convolutional neural networks through side-channel information leaks. In *ACM/IEEE Design Automation Conference (DAC)*, 2018. 3, 15, 26, 27, 28, 83, 97, 123

[21] Naina Gupta, Arpan Jati, and Anupam Chattopadhyay. AI attacks AI: Recovering neural network architecture from NVDLA using AI-assisted side channel attack. *Cryptology ePrint Archive*, 2023. 2, 3, 7, 14, 21, 26, 27, 83

[22] Vincent Meyers, Dennis Gnad, and Mehdi Tahoori. Reverse engineering neural network folding with remote FPGA power analysis. In *IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 1–10, 2022. 3, 14, 26, 27, 40

[23] Yicheng Zhang, Rozhin Yasaei, Hao Chen, Zhou Li, and Mohammad Abdullah Al Faruque. Stealing neural network structure through remote FPGA side-channel analysis. *IEEE Transactions on Information Forensics and Security*, 16:4377–4388, 2021. 3, 7, 13, 14, 26, 27, 40, 83

[24] Huili Chen, Cheng Fu, Jishen Zhao, and Farinaz Koushanfar. ProFlip: Targeted trojan attack with progressive bit flips. In *IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 7718–7727, 2021. 3, 18, 21, 108, 112, 117, 125

[25] Adnan Siraj Rakin, Zhezhi He, and Deliang Fan. Bit-flip attack: Crushing neural network with progressive bit search. In *IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 1211–1220, 2019. 18, 53, 54, 108, 112, 124, 130

[26] Adnan Siraj Rakin, Zhezhi He, and Deliang Fan. TBT: Targeted neural network attack with bit trojan. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 13198–13207, 2020. 18, 21, 112, 124

[27] 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*, 44(11):7928–7939, 2021. 124

[28] 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. 3, 18, 108, 112, 117, 124, 125

[29] Shanquan Tian, Shayan Moini, Adam Wolnikowski, Daniel Holcomb, Russell Tessier, and Jakub Szefer. Remote power attacks on the versatile tensor accelerator in multi-tenant FPGAs. In *IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2021. 3, 7, 13, 14, 21, 26, 27, 30, 32, 40, 83, 85, 97

[30] Daniel J Bernstein, Tanja Lange, and Peter Schwabe. The security impact of a new cryptographic library. In *International Conference on Cryptology and Information Security in Latin America (LATINCRYPT)*, pages 159–176, 2012. 3, 17, 81

[31] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. On subnormal floating point and abnormal timing. In *IEEE Symposium on Security and Privacy (S&P)*, pages 623–639, 2015. 17

[32] Yi-Liang Hong, Yui-Kai Weng, and Shih-Hsu Huang. Hardware implementation for fending off side-channel attacks. In *IEEE International Conference on Consumer Electronics-Taiwan (ICCE-TW)*, pages 1–2, 2021. 16

[33] Apostolos P. Fournaris and Odysseas Koufopavlou. Protecting CRT RSA against fault and power side channel attacks. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 159–164, 2012. 3, 16, 81

[34] Mahya Morid Ahmadi, Faiq Khalid, Radha Vaidya, Florian Kriebel, Andreas Steininger, and Muhammad Shafique. SHIELD: An adaptive and lightweight defense against the remote power side-channel attacks on multi-tenant FPGAs. *Available at SSRN 4459334*, 2023. 3, 16, 81

[35] Jonas Krautter, Dennis RE Gnad, Falk Schellenberg, Amir Moradi, and Mehdi B Tahoori. Active fences against voltage-based side channels in multi-tenant FPGAs. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, 2019. 17, 97, 100, 101

[36] Fan Zhang, Zhiyong Wang, Haoting Shen, Bolin Yang, Qianmei Wu, and Kui Ren. DARPT: Defense against remote physical attack based on TDC in multi-tenant scenario. In *ACM/IEEE Design Automation Conference (DAC)*, pages 559–564, 2022. 16, 99

[37] Kwen-Siong Chong, Jun-Sheng Ng, Juncheng Chen, Ne Kyaw Zwa Lwin, Nay Aung Kyaw, Weng-Geng Ho, Joseph Chang, and Bah-Hwee Gwee. Dual-hiding side-channel-attack resistant FPGA-based asynchronous-logic AES: Design, countermeasures and evaluation. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 11(2):343–356, 2021. 3, 16, 81

[38] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 406–418, 2016. 3, 81, 99

[39] Zhenghong Wang and Ruby B Lee. New cache designs for thwarting software cache-based side channel attacks. In *International Symposium on Computer Architecture (ISCA)*, pages 494–505, 2007.

[40] Yanan Guo, Andrew Zigerelli, Youtao Zhang, and Jun Yang. IvCache: Defending cache side channel attacks via invisible accesses. In *Great Lakes Symposium on VLSI (GLSVLSI)*, pages 403–408, 2021.

[41] Read Sprabery, Konstantin Evchenko, Abhilash Raj, Rakesh B Bobba, Sibin Mohan, and Roy Campbell. Scheduling, isolation, and cache allocation: A side-channel defense. In *IEEE International Conference on Cloud Engineering (IC2E)*, pages 34–40, 2018. 17

[42] Soo-Jin Moon, Vyas Sekar, and Michael K Reiter. Nomad: Mitigating arbitrary cloud side channels via provider-assisted migration. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1595–1606, 2015.

[43] Ziqiao Zhou, Michael K Reiter, and Yinqian Zhang. A software approach to defeating side channels in last-level caches. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 871–882, 2016. 3, 81

[44] Jialai Wang, Ziyuan Zhang, Meiqi Wang, Han Qiu, Tianwei Zhang, Qi Li, Zongpeng Li, Tao Wei, and Chao Zhang. Aegis: Mitigating targeted bit-flip attacks against deep neural networks. In *USENIX Security Symposium*, pages 2329–2346, 2023. 3, 20, 69, 109, 110, 125

[45] Jingtao Li, Adnan Siraj Rakin, Yan Xiong, Liangliang Chang, Zhezhi He, Deliang Fan, and Chaitali Chakrabarti. Defending bit-flip attack through DNN weight reconstruction. In *ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020. 3, 20, 69, 109

[46] Zhezhi He, Adnan Siraj Rakin, Jingtao Li, Chaitali Chakrabarti, and Deliang Fan. Defending and harnessing the bit-flip based adversarial weight attack. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 14095–14103, 2020. 3, 20, 69, 109, 110, 125

[47] Adnan Siraj Rakin, Li Yang, Jingtao Li, Fan Yao, Chaitali Chakrabarti, Yu Cao, Jae-sun Seo, and Deliang Fan. RA-BNN: Constructing robust & accurate binary neural network to simultaneously defend adversarial bit-flip attack and improve accuracy. *arXiv preprint arXiv:2103.13813*, 2021. 3, 20, 109, 110, 125

[48] Yukui Luo, Adnan Siraj Rakin, Deliang Fan, and Xiaolin Xu. DeepShuffle: A lightweight defense framework against adversarial fault injection attacks on deep neural networks in multi-tenant cloud-FPGA. In *IEEE Symposium on Security and Privacy (S&P)*, pages 3293–3310, 2024. 3, 20, 109, 125

[49] Cheng Gongye, Yukui Luo, Xiaolin Xu, and Yunsi Fei. HammerDodger: A lightweight defense framework against RowHammer attack on DNNs. In *ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2023. 3, 20, 109, 110, 113, 125

[50] Liang Liu, Yanan Guo, Yueqiang Cheng, Youtao Zhang, and Jun Yang. Generating robust DNN with resistance to bit-flip based adversarial weight attack. *IEEE Transactions on Computers*, 72(2):401–413, 2022. 3, 109, 125

[51] Jingtao Li, Adnan Siraj Rakin, Zhezhi He, Deliang Fan, and Chaitali Chakrabarti. Radar: Run-time adversarial weight attack detection and accuracy recovery. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 790–795, 2021. 3, 20, 69, 109, 110

[52] Yanzuo Chen, Yuanyuan Yuan, Zhibo Liu, Sihang Hu, Tianxiang Li, and Shuai Wang. BitShield: Defending against bit-flip attacks on DNN executables. *Computing*, 2:47, 2023. 3, 109, 110, 125, 128

[53] Mojan Javaheripi, Jung-Woo Chang, and Farinaz Koushanfar. ACCHashtag: Accelerated hashing for detecting fault-injection attacks on embedded neural networks. *ACM Journal on Emerging Technologies in Computing Systems*, 19(1):1–20, 2022. 3, 20, 109, 110

[54] Mojan Javaheripi and Farinaz Koushanfar. Hashtag: Hash signatures for online detection of fault-injection attacks on deep neural networks. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–9, 2021. 3, 20, 69, 109, 110

[55] Yu Li, Min Li, Bo Luo, Ye Tian, and Qiang Xu. DeepDyve: Dynamic verification for deep neural networks. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 101–112, 2020. 3, 20, 109, 110

[56] Shaofeng Li, Xinyu Wang, Minhui Xue, Haojin Zhu, Zhi Zhang, Yansong Gao, Wen Wu, and Xuemin Sherman Shen. Yes, one-bit-flip matters! Universal DNN model inference depletion with runtime code fault injection. In *USENIX Security Symposium*, 2024. 3, 19, 21, 53, 54, 108, 112, 117, 124, 125

[57] Stjepan Picek, Dirmanto Jap, and Shivam Bhasin. Poster: When adversary becomes the guardian–towards side-channel security with adversarial attacks. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2673–2675, 2019. 3, 81, 82, 87

[58] Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. DeepCloak: Adversarial crafting as a defensive measure to cloak processes. In *Workshop on DYnamic and Novel Advances in Machine Learning and Intelligent Cyber Security*, pages 1–12, 2019. 3, 81, 87

[59] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. MASCAT: Stopping microarchitectural attacks before execution. *Cryptology ePrint Archive*, 2016. 3, 20, 109

[60] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A fast and stealthy cache attack. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pages 279–299, 2016. 3, 109

[61] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. Real time detection of cache-based side-channel attacks using hardware performance counters. *Applied Soft Computing*, 49:1162–1174, 2016. 3, 109

[62] Lucian Cojocar, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. Exploiting correcting codes: On the effectiveness of ECC memory against RowHammer attacks. In *IEEE Symposium on Security and Privacy (S&P)*, pages 55–71, 2019. 3, 19, 53, 109

[63] Radhesh Krishnan Konoth, Marco Oliverio, Andrei Tatar, Dennis Andriesse, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. ZebRAM: Comprehensive and compatible software protection against RowHammer attacks. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 697–710, 2018. 3, 17, 20, 109

[64] Kevin Loughlin, Jonah Rosenblum, Stefan Saroiu, Alec Wolman, Dimitrios Skarlatos, and Baris Kasikci. Siloz: Leveraging DRAM isolation domains to prevent inter-VM RowHammer. In *Symposium on Operating Systems Principles*, pages 417–433, 2023. 3, 20, 109

[65] Kai Schramm, Gregor Leander, Patrick Felke, and Christof Paar. A collision-attack on AES: Combining side channel- and differential-attack. In *Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, pages 163–175. Springer, 2004. 13, 14

[66] Elisabeth Oswald, Stefan Mangard, Norbert Pramstaller, and Vincent Rijmen. A side-channel analysis resistant description of the AES S-box. In *International Workshop on Fast Software Encryption (FSE)*, pages 413–423. Springer, 2005. 14

[67] Mark Zhao and G Edward Suh. FPGA-based remote power side-channel attacks. In *IEEE Symposium on Security and Privacy (S&P)*, 2018. 14, 29, 97, 100, 101

[68] Tanu Shree Rastogi and Elif Bilge Kavun. Deep learning techniques for side-channel analysis on AES datasets collected from hardware and software platforms. In *International Conference on Embedded Computer Systems (SAMOS)*, pages 300–316. Springer, 2021. 13

[69] Yaman Umuroglu, Nicholas J Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. FINN: A framework for fast, scalable binarized neural network inference. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 65–74, 2017. 13, 14, 27

[70] Thierry Moreau, Tianqi Chen, Luis Vega, Jared Roesch, Eddie Yan, Lianmin Zheng, Josh Fromm, Ziheng Jiang, Luis Ceze, Carlos Guestrin, et al. A hardware–software blueprint for flexible deep learning specialization. *IEEE Micro*, 39(5):8–16, 2019. 14, 27

[71] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic analysis: Concrete results. In *Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, pages 251–261. Springer, 2001. 14

[72] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In *Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, pages 16–29, 2004. 14

[73] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Annual International Cryptology Conference (CRYPTO)*, pages 388–397, 1999. 14

[74] Eric Peeters, François-Xavier Standaert, and Jean-Jacques Quisquater. Power and electromagnetic analysis: Improved model, consequences and comparisons. *Integration*, 40(1):52–60, 2007. 14

[75] Ruize Wang, Huanyu Wang, and Elena Dubrova. Far field EM side-channel attack on AES using deep learning. In *ACM Workshop on Attacks and Solutions in Hardware Security (ASHES)*, pages 35–44, 2020. 14

[76] Paul C Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Annual International Cryptology Conference (CRYPTO)*, pages 104–113. Springer, 1996. 15

[77] Yanting Ren, An Wang, and Liji Wu. Transient-steady effect attack on block ciphers. In *Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, pages 433–450. Springer, 2015. 15

[78] Prakhar Kaushik and Rana Majumdar. Timing attack analysis on AES on modern processors. In *International Conference on Reliability, Infocom Technologies and Optimization (ICRITO)*, pages 462–465. IEEE, 2017. 15

[79] Chao Luo, Yunsi Fei, and David Kaeli. Side-channel timing attack of RSA on a GPU. *ACM Transactions on Architecture and Code Optimization (TACO)*, 16(3):1–18, 2019. 15

[80] Aurélie Bauer, Eliane Jaulmes, Victor Lomné, Emmanuel Prouff, and Thomas Roche. Side-channel attack against RSA key generation algorithms. In *Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, pages 223–241. Springer, 2014. 15

[81] Takaya Kubota, Kota Yoshida, Mitsuru Shiozaki, and Takeshi Fujino. Deep learning side-channel attack against hardware implementations of AES. *Microprocessors and Microsystems*, 87:103383, 2021. 15

[82] Yoo-Seung Won, Soham Chatterjee, Dirmanto Jap, Shivam Bhasin, and Arindam Basu. Time to leak: Cross-device timing attack on edge deep learning accelerator. In *International Conference on Electronics, Information, and Communication (ICEIC)*, pages 1–4. IEEE, 2021. 15

[83] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games–bringing access-based cache attacks on AES to practice. In *IEEE Symposium on Security and Privacy (S&P)*, pages 490–505. IEEE, 2011. 15

[84] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *Cryptographers' Track at the RSA Conference (CT-RSA)*, pages 1–20. Springer, 2006. 15

[85] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security Symposium*, pages 719–732, 2014. 15

[86] Berk Gulmezoglu, Mehmet Sinan Inci, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cross-VM cache attacks on AES. *IEEE Transactions on Multi-Scale Computing Systems*, 2(3):211–222, 2016. 15

[87] Fan Yao, Milos Doroslovacki, and Guru Venkataramani. Are coherence protocol states vulnerable to information leakage? In *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 168–179. IEEE, 2018. 15

[88] Yang Zhao, Xing Hu, Shuangchen Li, Jing Ye, Lei Deng, Yu Ji, Jianyu Xu, Dong Wu, and Yuan Xie. Memory trojan attack on neural network accelerators. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1415–1420. IEEE, 2019. 15

[89] Yuankun Zhu, Yueqiang Cheng, Husheng Zhou, and Yantao Lu. Hermes attack: Steal DNN models with lossless inference accuracy. In *USENIX Security Symposium*, 2021. 16

[90] Anuj Dubey, Rosario Cammarota, and Aydin Aysu. BoMaNet: Boolean masking of an entire neural network. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–9, 2020. 16, 99

[91] Anuj Dubey, Rosario Cammarota, Vikram Suresh, and Aydin Aysu. Guarding machine learning hardware against physical side-channel attacks. *ACM Journal on Emerging Technologies in Computing Systems*, 18(3):1–31, 2022. 16, 99

[92] Saurav Maji, Utsav Banerjee, Samuel H Fuller, and Anantha P Chandrakasan. A threshold implementation-based neural network accelerator with power and electromagnetic side-channel countermeasures. *IEEE Journal of Solid-State Circuits*, 58(1):141–154, 2022. 16

[93] Mohammad Hashemi, Steffi Roy, Domenic Forte, and Fatemeh Ganji. HWGN2: Side-channel protected neural networks through secure and private function evaluation. *arXiv preprint arXiv:2208.03806*, 2022. 16

[94] Anuj Dubey, Rosario Cammarota, Avinash Varna, Raghavan Kumar, and Aydin Aysu. Hardware-software co-design for side-channel protected neural network inference. In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 155–166, 2023. 16

[95] Maamar Ouladj and Sylvain Guilley. *Side-channel analysis of embedded systems*. Springer, 2021. 16

[96] Lu Zhang, Dejun Mu, Yuxuan Huang, Jingyu Wang, Yifan He, Yaolei Li, Lizhou Liu, Kaiwei Zou, Huazhong Yang, and Yongpan Liu. Pareto frequency-aware power side-channel countermeasure exploration on CNN systolic array. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 70(3):1124–1128, 2022. 16

[97] Raghavendra Pradyumna Pothukuchi, Sweta Yamini Pothukuchi, Petros G Voulgaris, Alexander Schwing, and Josep Torrellas. Maya: Using formal control to obfuscate power side channels. In *International Symposium on Computer Architecture (ISCA)*, pages 888–901, 2021. 16, 97, 100, 101

[98] Tong Zhou, Shaolei Ren, and Xiaolin Xu. ObfuNAS: A neural architecture search-based DNN obfuscation approach. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–9, 2022. 17, 81, 97, 99, 101, 111, 123

[99] Jingtao Li, Zhezhi He, Adnan Siraj Rakin, Deliang Fan, and Chaitali Chakrabarti. NeurObfuscator: A full-stack obfuscation tool to mitigate neural architecture stealing. In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 248–258, 2021. 97, 99, 101, 111, 123

[100] Yukui Luo, Shijin Duan, Cheng Gongye, Yunsi Fei, and Xiaolin Xu. NNReArch: A tensor program scheduling framework against neural network architecture reverse engineering. In *IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 1–9, 2022. 17, 81

[101] Eleonora Cagli, Cécile Dumas, and Emmanuel Prouff. Convolutional neural networks with data augmentation against jitter-based countermeasures: Profiling attacks without pre-processing. In *Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, pages 45–68, 2017. 17

[102] Tyler Hunt, Zhipeng Jia, Vance Miller, Christopher J Rossbach, and Emmett Witchel. Isolation and beyond: Challenges for system security. In *Workshop on Hot Topics in Operating Systems (HotOS)*, pages 96–104, 2019. 17

[103] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security Symposium*, pages 857–874, 2016. 17

[104] Sanjay Das, Swastik Bhattacharya, Souvik Kundu, Shamik Kundu, Anand Menon, Arnab Raha, and Kanad Basu. AttentionBreaker: Adaptive evolutionary optimization for unmasking vulnerabilities in LLMs through bit-flip attacks. *arXiv preprint arXiv:2411.13757*, 2024. 18

[105] Haotian Xu, Qingsong Peng, Jie Shi, Huadi Zheng, Yu Li, and Cheng Zhuo. SilentStriker: Toward stealthy bit-flip attacks on large language models. *arXiv preprint arXiv:2509.17371*, 2025. 18

[106] Zachary Coalson, Jeonghyun Woo, Shiyang Chen, Yu Sun, Lishan Yang, Prashant Nair, Bo Fang, and Sanghyun Hong. PrisonBreak: Jailbreaking large language models with fewer than twenty-five targeted bit-flips. *arXiv preprint arXiv:2412.07192*, 2024. 19, 53, 63, 69

[107] Noureldin Zahran, Ahmad Tahmasivand, Ihsen Alouani, Khaled Khasawneh, and Mohammed Fouda. On jailbreaking quantized language models through fault injection attacks. In *Great Lakes Symposium on VLSI (GLSVLSI)*, pages 554–561, 2025. 19

[108] Yanzuo Chen, Zhibo Liu, Yuanyuan Yuan, Sihang Hu, Tianxiang Li, and Shuai Wang. Unveiling single-bit-flip attacks on DNN executables. *arXiv preprint arXiv:2309.06223*, 2023. 19, 21, 53, 108, 112, 124, 125, 126

[109] Ilia Shumailov et al. Sponge examples: Energy-latency attacks on neural networks. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 212–231, 2021. 21, 48, 50, 63

[110] Xiaoning Feng, Xiaohong Han, Simin Chen, and Wei Yang. LLMEffiChecker: Understanding and testing efficiency degradation of large language models. *ACM Transactions on Software Engineering and Methodology*, 2024. 21, 48, 51, 63

[111] Amazon F1 web site. URL `https://aws.amazon.com/ec2/instance-types/f1/`. 26, 83

[112] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *International Symposium on Computer Architecture (ISCA)*, 2014. 26

[113] Guanlin Li, Guowen Xu, Shangwei Guo, Han Qiu, Jiwei Li, and Tianwei Zhang. Extracting robust models with uncertain examples. In *International Conference on Learning Representations (ICLR)*, 2022. 26

[114] Kangjie Chen, Shangwei Guo, Tianwei Zhang, Xiaofei Xie, and Yang Liu. Stealing deep reinforcement learning models for fun and profit. In *ACM Asia Conference on Computer and Communications Security (ASIACCS)*, pages 307–319, 2021.

[115] Wenbo Jiang, Hongwei Li, Guowen Xu, Tianwei Zhang, and Rongxing Lu. A comprehensive defense framework against model extraction attacks. *IEEE Transactions on Dependable and Secure Computing*, 2023. 26

[116] Joel Mandebi Mbongue, Alexis Maya-Isabelle Shuping, Pankaj Bhowmik, and Christophe Bobda. Architecture support for FPGA multi-tenancy in the cloud. In *IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2020. 26

[117] Dennis RE Gnad, Fabian Oboril, Saman Kiamehr, and Mehdi B Tahoori. Analysis of transient voltage fluctuations in FPGAs. In *International Conference on Field-Programmable Technology (FPT)*, 2016. 26, 83

[118] Shayan Moini, Shanquan Tian, Daniel Holcomb, Jakub Szefer, and Russell Tessier. Remote power side-channel attacks on BNN accelerators in FPGAs. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2021. 26, 30, 32, 81, 83, 97

[119] Joseph Gravellier. *Remote hardware attacks on connected devices*. PhD thesis, Ecole des Mines de Saint-Etienne, 2021. 26, 30, 35, 42, 83

[120] Xiaoxuan Lou, Tianwei Zhang, Jun Jiang, and Yinqian Zhang. A survey of microarchitectural side-channel vulnerabilities, attacks, and defenses in cryptography. *ACM Computing Surveys (CSUR)*, 54(6):1–37, 2021. 26

[121] Mengjia Yan, Christopher W Fletcher, and Josep Torrellas. Cache telepathy: Leveraging shared resource attacks to learn DNN architectures. In *USENIX Security Symposium*, 2020. 28, 84, 94

[122] Sanghyun Hong, Michael Davinroy, Yiğitcan Kaya, Dana Dachman-Soled, and Tudor Dumitraş. How to 0wn NAS in your spare time. In *International Conference on Learning Representations (ICLR)*, 2020. 28

[123] Xiaoxuan Lou, Shangwei Guo, Jiwei Li, and Tianwei Zhang. Ownership verification of DNN architectures via hardware cache side channels. *IEEE Transactions on Circuits and Systems for Video Technology*, 32(11):8078–8093, 2022. 28

[124] Giuseppe Cesarano. FPGA implementation of a deep learning inference accelerator for autonomous vehicles. Master's thesis, Politecnico di Torino, 2018. 28, 96

[125] Sanjay Pant. *Design and analysis of power distribution networks in VLSI circuits*. PhD thesis, University of Michigan, 2008. 29

[126] François-Xavier Standaert, François Koeune, and Werner Schindler. How to compare profiled side-channel attacks? In *International Conference on Applied Cryptography and Network Security*, 2009. 30, 84

[127] 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 (ICML)*, 2016. 31

[128] Graham Neubig. Neural machine translation and sequence-to-sequence models: A tutorial. *arXiv preprint arXiv:1703.01619*, 2017. 31

[129] Md Sanzidul Islam, Sadia Sultana Sharmin Mousumi, Sheikh Abujar, and Syed Akhter Hossain. Sequence-to-sequence Bangla sentence generation with LSTM recurrent neural networks. *Procedia Computer Science*, 2019. 31

[130] Kulothunkan Palasundram, Nurfadhlina Mohd Sharef, Khairul Azhar Kasmiran, and Azreen Azman. Enhancements to the sequence-to-sequence-based natural answer generation models. *IEEE Access*, 2020. 31

[131] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in Neural Information Processing Systems (NeurIPS)*, 30, 2017. 31, 40

[132] Florian Tramèr, Fan Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Stealing machine learning models via prediction {APIs}. In *25th USENIX security symposium (USENIX Security 16)*, pages 601–618, 2016. 31

[133] Tribhuvanesh Orekondy, Bernt Schiele, and Mario Fritz. Knockoff nets: Stealing functionality of black-box models. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4954–4963, 2019. 31

[134] Matthew Jagielski, Nicholas Carlini, David Berthelot, Alex Kurakin, and Nicolas Papernot. High accuracy and high fidelity extraction of neural networks. In *USENIX Security Symposium*, 2020. 31, 32, 80

[135] Seong Joon Oh, M. Augustin, M. Fritz, and B. Schiele. Towards reverse-engineering black-box neural networks. In *International Conference on Learning Representations (ICLR)*, 2018. 32, 80

[136] Yukui Luo, Cheng Gongye, Yunsi Fei, and Xiaolin Xu. Deepstrike: Remotely-guided fault injection attacks on DNN accelerator in cloud-FPGA. In *ACM/IEEE Design Automation Conference (DAC)*, 2021. 32

[137] NVIDIA. Hardware architectural specification, 2018. URL `http://nvdla.org/hw/v1/hwarch.html`. 34, 39

[138] S. Moini, X. Li, P. Stanwicks, G. Provelengios, W. Burleson, R. Tessier, and D. Holcomb. Understanding and comparing the capabilities of on-chip voltage sensors against remote power attacks on FPGAs. In *IEEE International Midwest Symposium on Circuits and Systems (MWSCAS)*, 2020. 35, 42

[139] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013. 44, 81, 86

[140] Sandy Huang, Nicolas Papernot, Ian Goodfellow, Yan Duan, and Pieter Abbeel. Adversarial attacks on neural network policies. *arXiv preprint arXiv:1702.02284*, 2017. 44

[141] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *IEEE Symposium on Security and Privacy (S&P)*, pages 39–57, 2017. 44

[142] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. DeepFool: A simple and accurate method to fool deep neural networks. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2574–2582, 2016. 44

[143] 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. 44, 89

[144] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership inference attacks against machine learning models. In *IEEE Symposium on Security and Privacy (S&P)*, pages 3–18, 2017. 45

[145] Nicholas Carlini et al. Extracting training data from large language models. In *USENIX Security Symposium*, volume 6, 2021. 48

[146] Long Ouyang et al. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems (NeurIPS)*, 35:27730–27744, 2022.

[147] Hugo Touvron et al. LLaMA: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023. 48

[148] Xinyue Shen, Zeyuan Chen, Michael Backes, and Yang Zhang. In ChatGPT we trust? Measuring and characterizing the reliability of ChatGPT. *arXiv preprint arXiv:2304.08979*, 2023. 48

[149] Henner Gimpel et al. Unlocking the power of generative AI models and systems such as GPT-4 and ChatGPT for higher education: A guide for students and lecturers. Technical report, Hohenheim Discussion Papers in Business, Economics and Social Sciences, 2023. 48

[150] Shijie Wu, Ozan Irsoy, Steven Lu, Vadim Dabravolski, Mark Dredze, Sebastian Gehrmann, Prabhanjan Kambadur, David Rosenberg, and Gideon Mann. BloombergGPT: A large language model for finance. *arXiv preprint arXiv:2303.17564*, 2023. 48

[151] Avishag Shapira, Alon Zolfi, Luca Demetrio, Battista Biggio, and Asaf Shabtai. Denial-of-service attack on object detection model using universal adversarial perturbation. 2022. 48

[152] Avishag Shapira, Alon Zolfi, Luca Demetrio, Battista Biggio, and Asaf Shabtai. Phantom sponges: Exploiting non-maximum suppression to attack deep object detectors. In *IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*, 2023.

[153] Han Liu, Yuhao Wu, Zhiyuan Yu, Yevgeniy Vorobeychik, and Ning Zhang. SlowLiDAR: Increasing the latency of LiDAR-based detection using adversarial examples. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5146–5155, 2023.

[154] Coen Schoof, Stefanos Koffas, Mauro Conti, and Stjepan Picek. Beyond PhantomSponges: Enhancing sponge attack on object detection models. In *ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, pages 14–19, 2024.

[155] Yong Xiao, Jin Ma, Ping Yi, and Xiuzhen Chen. Sponge backdoor attack: Increasing the latency of object detection exploiting non-maximum suppression. In *International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2024. 50

[156] Chen Ma, Ningfei Wang, Qi Alfred Chen, and Chao Shen. SlowTrack: Increasing the latency of camera-based perception in autonomous driving using adversarial examples. In *AAAI Conference on Artificial Intelligence (AAAI)*, volume 38, pages 4062–4070, 2024. 50

[157] Andreas Müller and Erwin Quiring. The impact of uniform inputs on activation sparsity and energy-latency attacks in computer vision. *arXiv preprint arXiv:2403.18587*, 2024. 48

[158] Jonas Geiping, Alex Stein, Manli Shu, Khalid Saifullah, Yuxin Wen, and Tom Goldstein. Coercing LLMs to do and reveal (almost) anything. In *International Conference on Learning Representations (ICLR) Workshop*, 2024. 48, 51

[159] Jianshuo Dong, Ziyuan Zhang, Qingjie Zhang, Tianwei Zhang, Hao Wang, Hewu Li, Qi Li, Chao Zhang, Ke Xu, and Han Qiu. An engorgio prompt makes large language model babble on. *arXiv preprint arXiv:2412.19394*, 2024. 51, 63

[160] Abhinav Kumar, Jaechul Roh, Ali Naseh, Marzena Karpinska, Mohit Iyyer, Amir Houmansadr, and Eugene Bagdasarian. Overthink: Slowdown attacks on reasoning LLMs, 2025. 48, 51

[161] Kuofeng Gao, Yang Bai, Jindong Gu, Shu-Tao Xia, Philip Torr, Zhifeng Li, and Wei Liu. Inducing high energy-latency of large vision-language models with verbose images. *arXiv preprint arXiv:2401.11170*, 2024. 48, 51

[162] Xuan Zhou, Souvik Kundu, Dake Chen, Jie Huang, and Peter Beerel. What makes vision transformers robust towards bit-flip attack? In *International Conference on Pattern Recognition (ICPR)*, pages 424–438. Springer, 2024. 49

[163] Kevin Meng, David Bau, Alex Andonian, and Yonatan Belinkov. Locating and editing factual associations in GPT. *Advances in Neural Information Processing Systems (NeurIPS)*, 35:17359–17372, 2022. 49

[164] Simin Chen, Zihe Song, Mirazul Haque, Cong Liu, and Wei Yang. NICGSlowdown: Evaluating the efficiency robustness of neural image caption generation models. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 15365–15374, 2022. 50

[165] Simin Chen, Cong Liu, Mirazul Haque, Zihe Song, and Wei Yang. NMTsloth: Understanding and testing efficiency degradation of neural machine translation systems. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 1148–1160, 2022. 50

[166] OpenAI. API pricing. `https://openai.com/api/pricing/`, 2025. 51

[167] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in Neural Information Processing Systems (NeurIPS)*, 2017. 51

[168] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. LLM.int8(): 8-bit matrix multiplication for transformers at scale, 2022. 52, 62

[169] 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. 53, 111

[170] 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 *IEEE Symposium on Security and Privacy (S&P)*, pages 245–261, 2018. 53

[171] Patrick Jattke, Victor Van Der Veen, Pietro Frigo, Stijn Gunter, and Kaveh Razavi. Blacksmith: Scalable rowhammering in the frequency domain. In *IEEE Symposium on Security and Privacy (S&P)*, pages 716–734, 2022. 53

[172] Andreas Kogler, Jonas Juffinger, Salman Qazi, Yoongu Kim, Moritz Lipp, Nicolas Boichat, Eric Shiu, Mattias Nissler, and Daniel Gruss. Half-Double: Hammering from the next row over. In *USENIX Security Symposium*, 2022. 53

[173] Chris S Lin, Joyce Qu, and Gururaj Saileshwar. GPUHammer: RowHammer attacks on GPU memories are practical. *arXiv preprint arXiv:2507.08166*, 2025. 53

[174] Ataberk Olgun, Majd Osseiran, A. Giray Yağlıkçı, Yahya Can Tuğrul, Haocong Luo, Steve Rhyner, Behzad Salami, Juan Gomez Luna, and Onur Mutlu. Read disturbance in high bandwidth memory: A detailed experimental study on HBM2 DRAM chips. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2024. 53

[175] Jianshuo Dong, Han Qiu, Yiming Li, Tianwei Zhang, Yuanjie Li, Zeqi Lai, Chao Zhang, and Shu-Tao Xia. One-bit flip is all you need: When bit-flip attack meets model training. In *IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 4688–4698, 2023. 53

[176] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. Flip feng shui: Hammering a needle in the software stack. In *USENIX Security Symposium*, pages 1–18, 2016. 53, 114

[177] Adnan Siraj Rakin, Md Hafizul Islam Chowdhuryy, Fan Yao, and Deliang Fan. DeepSteal: Advanced model extractions leveraging efficient weight stealing in memories. In *IEEE Symposium on Security and Privacy (S&P)*, pages 1157–1174, 2022. 53

[178] Qi Liu, Jieming Yin, Wujie Wen, Chengmo Yang, and Shi Sha. NeuroPots: Realtime proactive defense against bit-flip attacks in neural networks. In *USENIX Security Symposium*, pages 6347–6364, 2023. 54, 109, 110, 112, 125

[179] 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. In *International Symposium on Computer Architecture (ISCA)*, pages 361–372, 2014. 54

[180] Meta. How companies are using Meta Llama — Meta. `https://about.fb.com/news/2024/05/how-companies-are-using-meta-llama/`, 2024. 55

[181] AWS. Serving LLMs using vLLM and Amazon EC2 instances with AWS AI chips. `https://aws.amazon.com/blogs/machine-learning/serving-llms-using-vllm-and-amazon-ec2-instances-with-aws-ai-chips/`, 2024.

[182] Microsoft. Deployment overview for Azure AI Foundry models. https://learn.microsoft.com/en-us/azure/ai-foundry/concepts/deployments-overview/, 2025. 55

[183] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM addressing for cross-CPU attacks. In *USENIX Security Symposium*, pages 565–581, 2016. 57

[184] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. RAMBleed: Reading bits in memory without accessing them. In *IEEE Symposium on Security and Privacy (S&P)*, pages 695–711, 2020. 57

[185] DeepSeek-AI. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning, 2025. 62

[186] Jinze Bai et al. Qwen technical report. *arXiv preprint arXiv:2309.16609*, 2023. 62

[187] Samantha. https://huggingface.co/cognitivecomputations/Samantha-1.11-7b, 2023. 62

[188] Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. Vicuna: An open-source chatbot impressing GPT-4 with 90%* ChatGPT quality, 2023. URL https://lmsys.org/blog/2023-03-30-vicuna/. 62

[189] Llama-2-7b-chat-hf. https://huggingface.co/meta-llama/Llama-2-7b-chat-hf, 2023. 62

[190] Mistral. https://huggingface.co/mistralai/Mistral-7B-v0.3, 2024. 62

[191] AI@Meta. Llama 3 model card. 2024. URL https://github.com/meta-llama/llama3/blob/main/MODEL_CARD.md. 62

[192] Qwen Team. Qwen2.5: A party of foundation models, September 2024. URL https://qwenlm.github.io/blog/qwen2.5/. 62

[193] Stanford. Alpaca. https://github.com/tatsu-lab/stanford_alpaca/, 2025. 62

[194] Yiming Li, Shuo Shao, Yu He, Junfeng Guo, Tianwei Zhang, Zhan Qin, Pin-Yu Chen, Michael Backes, Philip Torr, Dacheng Tao, et al. Rethinking data protection in the (generative) artificial intelligence era. *arXiv preprint arXiv:2507.03034*, 2025. 66

[195] Zitao Chen, Guanpeng Li, and Karthik Pattabiraman. A low-cost fault corrector for deep neural networks through range restriction. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–13, 2021. 69

[196] Adam Dziedzic, Muhammad Ahmad Kaleem, Yu Shen Lu, and Nicolas Papernot. Increasing the cost of model extraction with calibrated proof of work. *arXiv preprint arXiv:2201.09243*, 2022. 80, 83

[197] Justin Grana. Perturbing inputs to prevent model stealing. In *IEEE Conference on Communications and Network Security (CNS)*, pages 1–9, 2020. 83

[198] Jiliang Zhang, Shuang Peng, Yansong Gao, Zhi Zhang, and Qinghui Hong. APMSA: Adversarial perturbation against model stealing attacks. *IEEE Transactions on Information Forensics and Security*, 18:1667–1679, 2023. 80, 83

[199] Lingxiao Wei, Bo Luo, Yu Li, Yannan Liu, and Qiang Xu. I know what you see: Power side-channel attack on convolutional neural network accelerators. In *Annual Computer Security Applications Conference (ACSAC)*, 2018. 80

[200] Naina Gupta, Arpan Jati, and Anupam Chattopadhyay. AI attacks AI: Recovering neural network architecture from NVDLA using AI-assisted side channel attack. Cryptology ePrint Archive, Paper 2023/368, 2023. 81

[201] Mika Juuti, Sebastian Szyller, Samuel Marchal, and N Asokan. PRADA: Protecting against DNN model stealing attacks. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 512–527, 2019. 83

[202] Sanjay Kariyappa and Moinuddin K Qureshi. Defending against model stealing attacks with adaptive misinformation. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2020. 83

[203] Microsoft. Real-time AI: Microsoft announces preview of Project Brainwave. `https://blogs.microsoft.com/ai/build-2018-projectbrainwave/`, 2018. 83

[204] Yue Zha and Jing Li. Virtualizing FPGAs in the cloud. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 845–858, 2020. 83

[205] Onur Aciiçmez. Yet another microarchitectural attack: Exploiting I-cache. In *ACM Workshop on Computer Security Architecture*, pages 11–18, 2007. 84

[206] Naomi Benger, Joop Van de Pol, Nigel P Smart, and Yuval Yarom. "Ooh Aah... just a little bit": A small amount of side channel can go a long way. In *Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, pages 75–92, 2014. 84

[207] Mark Randolph and William Diehl. Power side-channel attack analysis: A review of 20 years of study for the layman. *Cryptography*, 4(2):15, 2020. 85

[208] Risto Miikkulainen, Jason Liang, Elliot Meyerson, Aditya Rawal, Daniel Fink, Olivier Francon, Bala Raju, Hormoz Shahrzad, Arshak Navruzyan, Nigel Duffy, et al. Evolving deep neural networks. In *Artificial Intelligence in the Age of Neural Networks and Brain Computing*, pages 293–312. Elsevier, 2019. 90

[209] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016. 91

[210] Geoffrey F Miller, Peter M Todd, and Shailesh U Hegde. Designing neural networks using genetic algorithms. In *International Conference on Genetic Algorithms (ICGA)*, volume 89, pages 379–384, 1989. 90

[211] Kenneth M. Zick, Meeta Srivastav, Wei Zhang, and Matthew French. Sensing nanosecond-scale voltage attacks and natural transients in FPGAs. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 101–104, 2013. 93

[212] Andreas Herkle, Holger Mandry, Joachim Becker, and Maurits Ortmanns. In-depth analysis and enhancements of RO-PUFs with a partial reconfiguration framework on Xilinx Zynq-7000 SoC FPGAs. In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 238–247, 2019. 93

[213] Joseph Gravellier, Jean-Max Dutertre, Yannick Teglia, and Philippe Loubet-Moundi. High-speed ring oscillator based sensors for remote side-channel attacks on FPGAs. In *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8, 2019. 98

[214] Yuan Yao, Pantea Kiaei, Richa Singh, Shahin Tajik, and Patrick Schaumont. Programmable RO (PRO): A multipurpose countermeasure against side-channel and fault injection attack. *arXiv preprint arXiv:2106.13784*, 2021. 99

[215] Javier Rando, Jie Zhang, Nicholas Carlini, and Florian Tramèr. Adversarial ML problems are getting harder to solve and to evaluate. *arXiv preprint arXiv:2502.02260*, 2025. 103

[216] Cihang Xie, Jianyu Wang, Zhishuai Zhang, Zhou Ren, and Alan Yuille. Mitigating adversarial effects through randomization. *arXiv preprint arXiv:1711.01991*, 2017. 103

[217] Jeremy Cohen, Elan Rosenfeld, and Zico Kolter. Certified adversarial robustness via randomized smoothing. In *International Conference on Machine Learning (ICML)*, pages 1310–1320, 2019. 103

[218] Chuan Guo, Mayank Rana, Moustapha Cisse, and Laurens Van Der Maaten. Countering adversarial images using input transformations. *arXiv preprint arXiv:1711.00117*, 2017. 103

[219] Takami Sato, Junjie Shen, Ningfei Wang, Yunhan Jia, Xue Lin, and Qi Alfred Chen. Dirty road can attack: Security of deep learning based automated lane centering under physical-world attack. In *USENIX Security Symposium*, pages 3309–3326, 2021. 108

[220] Weiting Zhang, Dong Yang, Wen Wu, Haixia Peng, Ning Zhang, Hongke Zhang, and Xuemin Shen. Optimizing federated learning in distributed industrial IoT: A multi-agent approach. *IEEE Journal on Selected Areas in Communications*, 39(12):3688–3703, 2021. 108

[221] Yannan Liu, Lingxiao Wei, Bo Luo, and Qiang Xu. Fault injection attack on deep neural network. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 131–138, 2017. 108

[222] Biresh Kumar Joardar, Tyler K Bletsch, and Krishnendu Chakrabarty. Machine learning-based RowHammer mitigation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 42(5):1393–1405, 2022. 109

[223] Fateme S Hosseini, Qi Liu, Fanruo Meng, Chengmo Yang, and Wujie Wen. Safeguarding the intelligence of neural networks with built-in light-weight integrity marks (LIMA). In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 1–12, 2021. 109, 110

[224] Qi Liu, Wujie Wen, and Yanzhi Wang. Concurrent weight encoding-based detection for bit-flip attack on neural network accelerators. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2020. 109, 110

[225] Huili Chen, Cheng Fu, Bita Darvish Rouhani, Jishen Zhao, and Farinaz Koushanfar. DeepAttest: An end-to-end attestation framework for deep neural networks. In *International Symposium on Computer Architecture (ISCA)*, pages 487–498, 2019. 110

[226] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, et al. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907*, 2018. 112

[227] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *IEEE Symposium on Security and Privacy (S&P)*, pages 574–588, 2013. 113

[228] Andrew Adiletta, Caner Tol, and Berk Sunar. LeapFrog: The RowHammer instruction skip attack. *arXiv preprint arXiv:2404.07878*, 2024. 114

[229] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. SoK: Automated software diversity. In *IEEE Symposium on Security and Privacy (S&P)*, pages 276–291, 2014. 114

[230] Andrei Homescu, Steven Neisius, Per Larsen, Stefan Brunthaler, and Michael Franz. Profile-guided automated software diversity. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–11, 2013. 114

[231] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009. 125

[232] Sebastian Houben, Johannes Stallkamp, Jan Salmen, Marc Schlipsing, and Christian Igel. Detection of traffic signs in real-world images: The German Traffic Sign Detection Benchmark. In *International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2013. 125

[233] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. ImageNet large scale visual recognition challenge. *International Journal of Computer Vision*, 115:211–252, 2015. 125

[234] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014. 125

[235] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016. 125