ABSTRACT

Title of Dissertation: BUILDING SECURE AND RELIABLE DEEP LEARNING SYSTEMS FROM A SYSTEMS SECURITY PERSPECTIVE

Sanghyun Hong
Doctor of Philosophy, 2021

Dissertation Directed by: Professor Tudor Dumitraş
Department of Computer Science

As deep learning (DL) is becoming a key component in many business and safety-critical systems, such as self-driving cars or AI-assisted robotic surgery, adversaries have started placing them on their radar. To understand their potential threats, recent work studied the worst-case behaviors of deep neural networks (DNNs), such as mispredictions caused by adversarial examples or models altered by data poisoning attacks. However, most of the prior work narrowly considers DNNs as an isolated mathematical concept, and this perspective overlooks a holistic picture—leaving out the security threats that involve vulnerable interactions between DNNs and *hardware or system-level components*.

In this dissertation, on three separate projects, I conduct a study on how DL systems, owing to the computational properties of DNNs, become particularly vulnerable to existing well-studied attacks. First, I study how over-parameterization hurts a system's resilience to fault-injection attacks. Even with a single bit-flip, when chosen carefully, an attacker can inflict an accuracy drop up to 100%, and half of a DNN's parameters have at least one-bit that degrades its accuracy over 10%. An adversary who wields Rowhammer, a fault attack that flips random or targeted bits in the physical memory (DRAM), can exploit this graceless degradation in practice. Second, I study how computational regularities compromise the confidentiality of a system. Leveraging the information leaked by a DNN processing a single sample, an adversary can steal the DNN's often proprietary architecture. An attacker armed with Flush+Reload, a remote side-channel attack, can accurately perform this reconstruction against a DNN deployed in the cloud. Third, I will show how input-adaptive DNNs, *e.g.*, multi-exit networks, fail to promise computational efficiency in an adversarial setting. By adding imperceptible input perturbations, an attacker can significantly increase a multi-exit network's computations to have predictions on an input. This vulnerability also leads to exploitation in resource-constrained settings such as an IoT scenario, where input-adaptive networks are gaining traction. Finally, building on the lessons learned from my projects, I conclude my dissertation by outlining future research directions for designing secure and reliable DL systems.

BUILDING SECURE AND RELIABLE DEEP LEARNING SYSTEMS FROM A SYSTEMS SECURITY PERSPECTIVE

by

Sanghyun Hong

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2021

To my wife who has always stood by me.

Acknowledgments

At this moment, when I am about to take my first step as an assistant professor, I remember that there are many who helped me along the way on this journey. Looking back, I realize how lucky I was. This dissertation would not have been possible without their support and assistance. I want to spare this moment to express my deepest gratitude to them.

I first would like to thank my advisor, Prof. Tudor Dumitraş, for his advice and support over the past six years. I was able to build the *security mindset* through conducting research projects under his guidance. In every project that I worked on with him, I was also happy. This was because of the *freedom*—he always emphasizes to his students—to choose the research area that I want. I felt privileged to have this freedom. I could also learn the joy of advising prospective researchers from the projects that he allowed me to work with our summer interns. He was further in my personal life: he invited me (and his students) to dinners and special occasions. This journey could have been lonely and detached without them. I am thankful for all those moments that I had with him.

I'd also like to thank my committee members, Prof. Michael Hicks, Prof. Dana Dachman-Soled, Prof. Leonidas Lampropoulos, Prof. Abhinav Shrivastava, Prof. Nicolas Papernot, and Dr. Nicolas Carlini, for their timely advice and invaluable feedback on this dissertation. I am thankful for all the discussions that I had with Prof. Michael Hicks. His feedback on this dissertation from his *programming language perspectives* enabled some of my future work. I thank Prof. Dana Dachman-Soled for her advice and support. From the research projects that I worked on with her, I had a chance to learn hardware *side-channel attacks* and how to formulate a security problem *mathematically*. I would like to mention Prof. Nicolas Papernot and Dr. Nicholas Carlini. As frontiers in machine learning security, their advice and feedback on this dissertation was a *compass*—I could explore the right directions. In particular, I could broaden my research horizon from the three months that I spent in Google Brain under the supervision of Dr. Nicholas Carlini.

You can extend your expertise by working with others. I want to show my sincere thanks to my awesome collaborators: Prof. Christiano Giuffrida, Prof. Noseong Park, Dr. Abhinav Srivastava, and Dr. Alex Kurakin. I built my expertise in *Rowhammer* attacks from the collaborations with Prof. Christiano Giuffrida and his student, Pietro Frigo. I could have a chance to learn *cloud security* while working with Dr. Abhinav Srivasta. I could learn from Dr. Alex Kurakin (and Dr. Nicholas Carlini) about *hidden insights* that the researchers in machine learning security had in their initial work during 2012–15.

We sometimes learn from friends and fellow students. I'd like to mention Yiğitcan Kaya, Erin Avllazagaj, Dr. Ronny Huang, and Varun Chandrasekaran. Special thanks to Yiğitcan. I won't forget the last four years when I had many many *discussions* with you on thousands of security and machine learning papers. On average, we met at least one time per week and discussed more than four hours every time. Thanks to Erin Avllazagaj and Dr. Ronny Huang. Erin, we were the last two persons staying in the office almost every day with caffeinated brains. I could learn about *malware* and how real-world hackers think. I built my expertise in data poisoning from all the discussions that I had with Dr. Ronny Huang. I thank to Varun Chandrasekaran. While working with you, I could learn how to be a *collaborative and resourceful person* in the community. I feel fortunate to be surrounded by many talented colleagues: Dr. Bumjun Kwon, Dr. Ziyun Zhu, Prof. Doowon Kim, Octavian, Stephanie, Sicheng and Bang. Thanks to them for all the comments and feedback on my research projects; they helped *steer my project directions*. I finally thank the research interns that I mentored: Michi Panaitescu, Ionuţ-Vlad Modoranu, Michael Davinroy, Khasmamad Shabanovi, Stefan Matcovici, Alina-Elena Nicolae, Stuart Nevans Locke, Ian Rackow, and Kevin Kulda. I enjoyed working with you and learned the importance of *leading by example and holding your hands*.

Special thanks to Dana Purcell in UMIACS and Tom Hurst in the CS department. I could not have focused on my research without their support in administrative issues.

I could not have even taken my first step towards this journey without my family's love, support, and encouragement. I'd like to save this moment to say thank you to my parents and grandparents for their endless love and support. Most importantly, there have been many ups and downs along this journey, and I thank my wife, Ahrong, for always being there with me. In loving memory, I dedicate this dissertation to all of them.

There are no words to express my gratitude to all of them adequately. I will advise and support at my best those I will meet in the future. As my parents always taught me:

The best way to thank your parents is to show the love you received to your children.

Sanghyun.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

A sound mind in a sound body.

— Thales of Miletus.

Over the past few years, machine learning has enabled breakthroughs in many applications, from object recognition [2], speech recognition [3] to language translation [4], where human inputs and supervision are required previously. Most of these advancements have been driven by deep neural networks (DNNs). Trained on large-scale datasets collected from various sources, DNNs start outperforming humans. For example, in object recognition, one of the most difficult tasks in computer vision, a convolutional neural network trained on millions of natural images performs better than humans since 2015 [2]. Nowadays, a network trained on an extensive collection of books is used as a backbone for DNNs in language tasks [5]. As shown successful, they are becoming a *core* component of safety- or security-critical systems, such as self-driving cars [6], robotic surgery [7], or brain implants [8], and compose *deep learning systems*.

Despite their remarkable successes, we often face catastrophic failures of deep learning systems in real-world environments. The Death of Elaine Herzberg happened in Tempa, Arizona, in 2018 could be such an example—a self-driving car on testing by

Uber hit a pedestrian riding a bicycle, and she died for her injuries from the accident [9]. From the investigation conducted by the National Transportation Safety Board (NTSB), the object recognition networks that identify obstacles from camera inputs could not spot the bicycler who had appeared in front of the vehicle. In addition to this *reliability failure*, we have seen *security failures*. In Black Hat USA 2019, researchers from Tencent bypassed Apple's FaceID by simply wearing glasses with adversarial patterns on their faces [10]. These examples suggest that an adversary can exploit the failure modes of DNNs to manipulate a victim's systems, which raises an important research question:

How can we build secure and reliable deep learning systems?

As understanding failure modes is the first step towards answering this question, researchers in adversarial machine learning study attack surfaces of deep learning systems. Their focus is to identify properties of DNNs that can lead to the worst-case behaviors of the systems. Prior work, for example, exposed that DNNs' predictions are sensitive to small changes in their inputs and demonstrated adversarial examples [11]. An adversary can craft human-imperceptible perturbations that fool the decisions of a victim DNN once they are added to clean inputs [12, 13, 14, 15, 16, 17, 18, 19]. Another line of work showed data poisoning attacks [20, 21, 22, 23, 24]. DNNs, unlike the traditional software that performs predefined behaviors, can *learn* correct behaviors from observations. A motivated attacker can exploit this property to alter the behaviors of deep learning systems. This attack could be useful when the attacker cannot modify a systems' inputs.

However, those work focuses only on the *limited* attack surfaces, such as predictions manipulated by adversarial examples or by data poisoning attacks, and leaves out many other attack surfaces in deep learning systems. In practice, deep learning systems rely on various system components, such as hardware, system software, communication networks, or humans who operate them, to perform DNN computations. As shown in security, each of those components is equally subject to adversarial pressure. In addition, there are advancements in components to enable faster and resource-efficient DNN computations—just as we had the "AlexNet moment" when DNN computations become practical by using GPUs. Yet, the vulnerability of those new components in adversarial settings have not been fully understood. In consequence, a motivated attacker can exploit such weaknesses in system components or combine them with the properties of DNNs to pose critical security (or reliability) threats to deep learning systems.

This dissertation presents my first step towards building secure and reliable deep learning systems—characterizing new threats. Specifically, I explore this hypothesis:

> *DNNs have computational properties, that traditional software does not have, which makes them particularly vulnerable to existing, under-examined attacks.*

To conduct this study, *I employ a new perspective: DNNs—like any other software—a computational tool running on computer systems.* Unlike previous studies that mostly examine the security of individual components, this systems security perspective allows us to focus on the desired properties that secure and reliable systems should have. Armed with this perspective, I examine whether deep learning systems are able to satisfy those properties in adversarial settings. Over the three research projects, corresponding to three desired properties: (1) I first study attack vectors an adversary can exploit to prevent a system from satisfying each property. The adversary here can utilize existing attacks, but under-examined against deep learning systems, such as hardware-level attacks, or well-studied threats such as adversarial input perturbations for different attack objectives. (2) I then characterize the vulnerabilities of DNNs that I identify in the above step. In particular, I examine the impact of various factors—that an adversary or a victim can control over a DNN—on the vulnerabilities, *e.g.*, datasets (or tasks), architecture choices, training procedures, or its representations in a system. (3) To understand the practical impacts, I further study the exploitation of those vulnerabilities in typical DNN deployment scenarios. (4) I finally discuss some directions for building defenses.

1.1 Approach: A Systems Security Perspective

How can we build secure and reliable deep learning systems?

To provide answers to the main research question, we need to address two key challenges.

(1) This study requires both systems and machine learning expertise. For example, to understand how cache-based side-channel attacks [25, 26, 27, 28] work, the knowledge of memory architectures is essential. In addition, to analyze their impact on systems in practice, we should also know how the memory architectures are used to form computing environments, *e.g.*, the cloud. Separately, in machine learning, expertise in understanding how DNNs work and developing tools for analyzing their behaviors is necessary for characterizing the vulnerability of DNNs in adversarial settings. Such efforts often result in identifying distinct behaviors of DNNs under adversarial pressure [29, 30], which becomes key ideas in developing defense mechanisms by suppressing malicious behaviors.

(2) The lack of a holistic perspective. Prior work in systems and machine learning studies the security of deep learning systems separately, which leads to *a false sense of security*. For instance, a common belief in machine learning is that DNNs are resilient to small perturbations to their parameters [31, 32, 33]. In consequence, it has been believed that deep learning systems will be fault-resilient—a small number of faults cannot make the systems lose their functionalities. Prior work on systems security also contributed to this view—causing random errors, such as random bit-flips in memory—is an ineffective attack [34, 35]. However, I will show later in my work [36] that a single bit-flip in the in-memory representation of a DNN can lead to an accuracy drop over 10%, and an adversary can exploit this vulnerability by using fault attacks, *e.g.*, Rowhammer, to make a victim system easily unavailable.

My approach here to address those challenges is to have:

A systems security perspective on studying the security of deep learning systems.

This holistic approach allows us to focus more on the security of a system as a whole than the security of each specific component—in this context, DNNs—of a system. In systems security, there are the desirable properties that *any* system should have to be secure and reliable. Regardless of how a system is composed, the system should satisfy them.

In this dissertation, I focus on the following three properties: (1) *Graceful degradation*: a system should be fault-tolerant—it maintains limited functionalities even when a large portion of it has been destroyed [37, 38, 39]. (2) *Confidentiality*: a system should not reveal its secrets, such as data or algorithms. (3) *Computational efficiency*: a system is desirable when it utilizes computationally efficient algorithms [40, 41].

A clear advantage of focusing on those system properties is that it enables us to consider attack methodology and objectives that are under-studied in prior work. For example, hardware or system-level threats such as fault-injection or side-channel attacks have not been considered as a practical attack vector against deep learning systems. However, as my approach considers the vulnerability in hardware or system software, they become a viable methodology that an adversary can exploit. Separately, most of the prior work manipulates a DNN's predictions that are only relevant to the availability of deep learning systems. Nevertheless, my studies concern other attack objectives, such as jeopardizing computational efficiency. It is unclear whether (and how) an adversary can achieve those goals. From this unique view, this dissertation exposes new threats causing unexpected damage to a system and ultimately tackles the false sense of security.

1.2 Desired System Properties

In this section, I provide an overview of each research project that corresponds to the study of each desired system property. Each study answers the following research questions:

On systems security side:

(1) How vulnerable are each desired property of deep learning systems to an adversary?

(2) How can the attacker exploit this vulnerability in practical settings?

(3) How can we defeat (or mitigate) the vulnerability by using system-level defenses?

On machine learning side:

(1) What are the computational properties of DNNs influence the vulnerability?

(2) What are the distinct internal behaviors we observe from a DNN under this threat?

(3) How can we suppress those adversarial behaviors by using DNN-level defenses?

1.2.1 Graceful Degradation

A system is secure and reliable when it can tolerate failures. In machine learning, DNNs have been shown to tolerate faults [31]: cumulative changes to their parameters (*e.g.*, pruning, quantization, or random noise) result in a graceful degradation of classification accuracy [32, 33, 42, 43]. However, the limits of this natural resilience are not well-understood in the presence of small adversarial changes to the DNN parameters' underlying memory representation, such as bit-flips that may be induced by hardware fault attacks.

My first study examines the effects of bitwise corruptions on 19 DNN models—six architectures on three image classification tasks. We show the *graceless degradation of DNNs induced by a single bitwise corruption*. Most models have at least one parameter that, after a specific bit-flip in their bitwise representation, causes an accuracy loss of over 90%. We employ simple heuristics to efficiently identify the parameters likely to be vulnerable. We estimate that 40–50% of the parameters in a model might lead to an accuracy drop greater than 10% when individually subjected to such one-bit perturbations. We also show that this vulnerability is prevalent—the number of vulnerable parameters in a model are similar across different datasets, architecture choices, or training techniques.

To demonstrate how an adversary can take advantage of this vulnerability, we study the impact of an exemplary hardware fault attack, Rowhammer, on DNNs. Specifically, we show that a Rowhammer-enabled attacker co-located in the same physical machine can inflict significant accuracy drops (up to 99%) even with single bit-flip corruptions and no knowledge of the model. Our results expose the limits of DNNs' resilience against parameter perturbations induced by real-world fault attacks. We conclude by discussing possible defenses and future research directions towards fault attack-resilient DNNs.

1.2.2 Confidentiality

Novel data processing pipelines and network architectures increasingly drive the success of deep learning. In consequence, the industry considers top-performing architectures as intellectual property (or trade secrets) and devotes considerable computational resources to discovering such architectures through neural architecture search. This practice provides an incentive for adversaries to steal these unique architectures; when used in the cloud, to provide Machine Learning as a Service, the adversaries also have an opportunity to reconstruct the architectures by exploiting a range of hardware side-channels. However, it is challenging to reconstruct those unique components without knowing the computational graph (*e.g.*, the layers, branches or skip connections), the architecture configurations (*e.g.*, the number of filters in a convolutional layer) or the specific pre-processing steps.

My second study shows that an adversary can extract this confidential information by exploiting hardware-level side-channels. To this end, we presents an algorithm that reconstructs the key components of a novel deep learning systems by exploiting a small amount of information leakage from a cache side-channel attack, Flush+Reload [26]. We exploit the computational regularities that DNNs have: A DNN processes an input sequentially with its layers, and the time it takes to process the input depends on its layer configurations. Flush+Reload reliably extracts the sequence of computations and the timing for each computation. Using those information, our attack generates candidate DNN architectures from the sequence and eliminates incompatible candidates through a parameter estimation process. This is a prevalent vulnerability as popular deep learning libraries, *e.g.*, PyTorch and Tensorflow, implement DNN computations in the same way.

In our experiments, we demonstrate that one can reconstruct MalConv [44], a novel data pre-processing pipeline for malware detection, and ProxylessNAS-CPU [45], a novel network architecture for the ImageNet classification optimized to run on CPUs, without knowing the architecture family. In both cases, we achieve 0% error. Our results suggest that hardware side-channels are a practical attack vector against MLaaS, and more efforts should be devoted to understanding their impact on the security of deep learning systems. Finally, we conclude by discussing some potential defenses against this new threat.

1.2.3 Computational Efficiency

Recent increases in the computational demands of DNNs, combined with the observation that most input samples require only shallow models, have sparked interest in input-adaptive multi-exit architectures such as MSDNets [46] or Shallow-Deep Networks [29]. Those architectures enable faster inferences and could bring DNNs to low-power devices, *e.g.*, in the Internet of Things (IoT). However, it is unknown whether the computational efficiency provided by this approach is robust against adversarial pressure. In particular, an adversary may aim to slowdown adaptive DNNs by increasing their average inference time—a threat analogous to the denial-of-service attacks from the Internet.

My last work shows that an adversary can jeopardize the computational savings provided by input-adaptive inference mechanisms. We study this threat systematically by experimenting with three generic multi-exit DNNs (based on VGG16, MobileNet, and ResNet56) and a custom multi-exit architecture (MSDNet), on two popular image classification benchmarks (CIFAR-10 and Tiny ImageNet). To this end, we show that adversarial example-crafting techniques can be modified to cause slowdowns. We also propose a metric for comparing the impact of our attacks on different architectures.

Our experiments show that the slowdown attack reduces the efficacy of multi-exit DNNs by 90–100%, and it amplifies the latency by 1.5–5× in a typical IoT deployment. We also show that it is possible to craft universal, reusable perturbations and that the attack can be effective in realistic black-box scenarios, where the attacker has limited knowledge about the victim. Finally, we show that adversarial training provides limited protection against slowdowns. These results suggest that further research is needed for defending multi-exit architectures against this emerging threat.

1.3 Contributions

In summary, this dissertation makes the following contributions:

This dissertation tackles the conventional perspective on the security of deep learning systems and makes a step towards characterizing new threats. To this end, I employ a systems security perspective that views DNNs as a computational tool running in systems. Using this holistic approach, I expose emerging threats that exploit new attack vectors, *e.g.*, hardware-level attacks, or study new objectives, *e.g.*, reconstructing unique DNN architectures or reducing computational savings of input-adaptive DNNs. Over the three research projects, I show that deep learning systems are more vulnerable to those new threats than we previously assumed. My findings suggest that further research is required for defending against the threats.

In the study that examines the graceless degradation of deep learning systems:

This study shows that DNN models are more vulnerable to bit-flip corruptions than previously assumed. In particular, I show that adversarial bitwise corruptions induced by hardware fault attacks can easily inflict severe indiscriminate damages by drastically increasing or decreasing the value of a model parameter. This work shows that on average, $\sim$50% of model parameters are vulnerable to single bit-flip corruptions, causing relative accuracy drops above 10%, and that all the networks that we examine include parameters that can cause an accuracy drop of over 90%.

This work conducts comprehensive analysis of DNN models' behavior against single bit-flips and characterize the vulnerability that a hardware fault attack can trigger. I examine the impact of various factors: the bit position, bit-flip direction, parameter sign, layer width, activation function, normalization, and model architecture. The analysis shows that the vulnerability is prevalent—the max. accuracy drop and the number of vulnerable parameters are the same across all the DNNs.

Based on this analysis, I examine the impact of practical hardware fault attacks in a representative deep learning scenario. I show that a Rowhammer-enabled attacker can inflict significant accuracy drops (up to 99%) on a victim's DNN running in the cloud even with constrained bit-flip corruptions and no knowledge of the model. I further show that the same attacker can inflict an accuracy drop of over 10% within a minute even when the attacker cannot control the bit-flip locations.

In the study that examines the confidentiality of deep learning systems:

My work shows that an adversary can steal unique deep learning systems by exploiting hardware side-channel attacks. To this end, I design an algorithm that reconstructs the key components of a victim deep learning system by exploiting a small amount of information leakage from a cache side-channel attack, Flush+Reload. I show that Flush+Reload reliably extracts the computational trace and exposes the time each computation takes in a practical cloud scenario. Using the extracted information, the attacker estimates the computational graph and the architectural configurations.

This study also shows that the attacker can reconstruct unique architectures in practical scenarios. In evaluation, I show the reconstruction of two unique DNN architectures with no error: (1) ProxylessNAS-CPU, a novel architecture for object recognition found from neural architecture search and (2) MalConv, a novel manually-designed data pre-processing pipeline for malware detection.

This study further shows that the vulnerability to our reconstruction attack is prevalent. I first show the reconstruction of a network architecture by exploiting the vulnerability shared across common deep learning frameworks, PyTorch and TensorFlow. More importantly, I also show that computational regularities of DNNs make the vulnerability to reconstruction attacks difficult to defeat.

In the study that examines the computational efficiency of deep learning systems:

This is the first study that examines the robustness of input-adaptive multi-exit DNNs against adversarial slowdowns. To this end, I first show that examples crafted by prior evasion attacks fail to bypass the victim model's early exits. I then show that an adversary can adapt such attacks to the goal of model slowdown by modifying its objective function—the DeepSloth attack. This work also proposes an efficacy metric for comparing slowdowns across different multi-exit architectures.

My work conducts a comprehensive evaluation of this new vulnerability. I experiment with three generic multi-exit DNNs (based on VGG16, ResNet56 and MobileNet) and a specially designed multi-exit architecture, MSDNets, on two popular image classification benchmarks (CIFAR-10 and Tiny ImageNet). The results show that DeepSloth reduces the efficacy of multiexit DNNs by 90–100%, *i.e.*, the perturbations render nearly all early exits ineffective. In a scenario typical for IoT deployments, where a DNN is partitioned between the edge and the cloud, DeepSloth amplifies the latency by 1.5–5×, negating the benefits of partitioning.

This study also shows that this is a prevalent vulnerability. I show that it is possible to craft a universal DeepSloth perturbation, which can cause slowdowns on either all or a class of inputs. While more constrained, this attack still reduces the efficacy by 5–45%. Further, I observe that DeepSloth can be effective in some black-box scenarios, where the attacker has limited knowledge about the victim.

Finally, I show that a standard defense against adversarial samples—adversarial

training—is inadequate against DeepSloth. I expose a tradeoff between being efficient, robust against standard adversarial examples, and resilient to DeepSloth. This suggests that further research is needed for protecting input-adaptive multi-exit networks against this emerging security threat.

1.4 Dissertation Structure

This section describes the structure of this dissertation. In Chapter 2, I introduce basic concepts that I will use throughout this dissertation. Here, I also review literature relevant to my works. In Chapter 3, I present my first work that exposes the graceless degradation of DNNs under hardware fault attacks. Next, I present my second work on attacking the confidentiality of DNNs by reconstructing their unique architectures from hardware side-channels (Chapter 4). In Chapter 5, I present my last work on nullifying computational savings provided by input-adaptive DNNs via adversarial input perturbations. Finally, I conclude this dissertation and discuss future research directions (in Chapter 6).

Chapter 2: Backgrounds and Related Work

If I have seen further it is by standing on the shoulders of Giants.

— Isaac Newton, 1675.

This chapter provides an overview of the background knowledge that is necessary to understand and discuss the three emerging threats this dissertation presents. At first, these attacks target the core component of a system, *i.e.*, neural networks. Hence, I overview how neural networks work, especially in classification settings, and how we train and deploy them in practice (Section 2.1). In Section 2.2, I discuss the conventional attacks on neural networks and defenses against them studied in the literature.

2.1 Neural Network

A neural network is a function $f : \mathcal{X} \rightarrow \mathcal{Y}$ that operates as a sequence of layers transforming an input $\mathbf{x}$ from the domain $\mathcal{X} \subseteq \mathbb{R}^d$ to an output $\mathbf{y}$ from the range $\mathcal{Y} \subseteq \mathbb{R}^n$. In a neural network, each layer contains *neurons* that apply a linear and non-linear transformation to their inputs—the outputs (activations) of the previous layer's neurons. The number of neurons in a layer is called its *width*, and the total number of layers is its *depth*. The *i*-th layer computes $f_i(\mathbf{a}) = g_i(\mathbf{w}_i \cdot \mathbf{a} + \mathbf{b}_i)$, where g_i represents a non-linear activation function, and $\mathbf{w}_i, \mathbf{b}_i$ are the trainable parameters (the *weights* and *biases*, respectively). The most commonly used activation function is ReLU$(\mathbf{a}) = max(\mathbf{a}, \mathbf{0})$, and the activation function is often followed by other optional layer structures such as dropout [47], pooling, or batch normalization [48].

A *fully-connected* network contains exclusively layers that connect each neuron in one layer to every neuron in the following layer. To make more efficient use of each parameter, *convolutional neural networks* [49] apply a two dimensional convolution over the input and are thus more time- and space-efficient. To maintain efficiency, the *window size* of the convolution is typically small (*e.g.*, 3×3 or 5×5). The convolutional neural networks use the same filters for every window, called *weight sharing*, making them require far fewer parameters than fully-connected networks.

In this dissertation, we will focus on neural networks in classification settings where the output $\mathbf{y}$ is a probability distribution over n class labels. The raw output of $f(\mathbf{x})$ is referred to as its *logits*, and the softmax $\sigma(\cdot)$ is applied to convert logits into a probability distribution. The inferred label of an input $\mathbf{x}$ by a network is then defined as $\hat{y} = argmax_i\, \sigma(f(\mathbf{x}))$ where $i \in \{1, 2, ..., n\}$. In case of converting an oracle label j into a distribution, one commonly uses *one-hot encoding* $\mathbf{y}_j = \mathbb{1}_i(j)$, *i.e.*, a vector $\mathbf{y}_j$ has one only at the index j and zero at the others.

2.1.1 Training and Inference of Neural Networks

Training. Neural networks are trained by *empirical risk minimization* (ERM). Given a dataset for training $\mathcal{D}_{tr} \subseteq \mathcal{X} \times \mathcal{Y}$, training minimizes the empirical risk $\mathbb{E}_{(\mathbf{x},\mathbf{y})\sim\mathcal{D}}[\mathcal{L}(f(\mathbf{x}), y)]$. Here, $\mathcal{L}$ is the loss function that quantifies how correct (or incorrect) the prediction of a network $f(\mathbf{x})$ is for an input $\mathbf{x}$ and its oracle label $\mathbf{y}$. A commonly used loss function is the cross-entropy loss.

Stochastic gradient descent (SGD) is the de-facto approach for minimizing the loss. Known as the backpropagation algorithm [50] in the context of neural networks, SGD updates network parameters θ with the gradients computed over a randomly sampled mini-batch B at each iteration t:

$$\theta_{t+1} \leftarrow \theta_t - \eta \cdot \frac{1}{|B|} \sum_{i \in B} \nabla_{\theta_t} \mathcal{L}(f_{\theta_t}(\mathbf{x}_i), \mathbf{y}_i)$$

where η is the *learning rate* and controls the magnitude of updates made at each iteration. During training, the network is tested based on how well it generalizes to the unseen samples in $\mathcal{D}_{ts} \subset \mathcal{D}$ and not in the training data $\mathcal{D}_{tr}$.

Inference. Once the network converges to an acceptable error rate, training stops, and the network, along with its parameters θ, will be stored as a *model* f_θ. During inference (or testing), we load this model into the memory and produce the prediction for a given input sample, usually not in the training data. We here fix the values of model parameters θ.

2.2 Related Work

This section discusses two conventional attacks relevant to our work.

2.2.1 Adversarial Examples

Szegedy *et al.* [11] were the first work that exposed the sensitivity of neural networks to test-time input perturbations. They found that the total amount of perturbations to an input required to change a network's prediction on it is small and called samples with such perturbations as *adversarial examples*. Inspired by this work, a vast literature [51, 52, 53, 54, 55, 56, 57] presented algorithms for crafting strong adversarial examples—*i.e.*, samples that fool a model's predictions with only small perturbations and high confidence.

Projected Gradient Descent (PGD) [54] is the most popular algorithm that computes adversarial input perturbations v for a test-time input (x, y) as follows:

$$v^{t+1} = \prod_{||v||_\infty < \epsilon} \Big(v^t + \alpha \ \mathrm{sgn}\big(\nabla_v \mathcal{L} \left(f_\theta(\mathbf{x}), \mathbf{y} \right) \big) \Big)$$

where t is the current attack iteration; α is the step-size; Π is the projection operator that bounds the perturbation on $||v||_\infty < \epsilon$ and $\mathcal{L}$ is the loss function. At each iteration t, this attack makes an update to the perturbations towards increasing the loss value on (x, y). PGD is commonly used for evaluating the reliability of a model's predictions on inputs.

Transferability of adversarial examples. The attacks above assume a *white-box* adversary who knows a victim model's internals and uses them for crafting adversarial examples.

However, there are scenarios where an adversary does not have the knowledge of a victim model. This *black-box* adversary can exploit a *surrogate* model instead for launching the attack and still hurt an unknown victim [55, 56, 57]. This *transferability* leads to adversarial examples in more practical black-box scenarios.

Adversarial training. Although many defenses [58, 59, 60, 61, 62] have been proposed against this threat, *adversarial training* (AT) has become the frontrunner [54].

2.2.2 Data Poisoning

Poisoning attacks [20, 21, 22, 24, 63, 64, 65] exploit the training process to manipulate a trained model's behavior by injecting maliciously crafted samples, *i.e.*, poisons, into the training set. If a model is trained on the contaminated training set, the attacker can (1) prevent the trained model from generalizing well to the test data [20, 21] or (2) cause misclassifications of specific test-time samples, *i.e.*, targets, without degrading the generalization performance of the model [22, 24, 63, 64, 65]. The former attacks are known as *indiscriminate* poisoning while the latter is *targeted* poisoning. Data poisoning is highly effective in attack scenarios where an adversary cannot control the test-time samples (unlike adversarial examples) or an attacker cannot alter the training procedures.

Chapter 3: Graceless Degradation Caused by Hardware Fault Attacks

There is no more great danger than underestimating your enemy.

— Lao Tzu's The Art of War (S. Griffith, Trans.), 1971 [66].

In 1990, Dr. Yann Lecun, a computer scientist and a professor at New York University, published the seminal paper "Optimal Brain Damage." [31]. This is the first work that reported the *graceful degradation* property: typically, cumulative changes to the network's parameters result in a graceful degradation of classification accuracy. This property has been harnessed in a broad range of techniques, such as network pruning [32], which significantly reduces the number of parameters in the network and leads to improved inference times. Besides structural resilience, DNN models can tolerate slight noise in their parameters with minimal accuracy degradation [67]. Researchers have proposed utilizing this property in defensive techniques, such as adding Gaussian noise to model parameters to strengthen DNN models against adversarial examples [43]. As a result, this natural resilience is believed to make it difficult for attackers to significantly degrade the overall accuracy by corrupting network parameters.

Recent work has explored the impact of *hardware faults* on DNN models [34, 42, 68]. Such faults can corrupt the memory storing the victim model's parameters, stress-testing DNNs' resilience to bitwise corruptions. For example, Qin et al. [42], confirming speculation from previous studies [32, 34], showed that a DNN model for CIFAR10 image classification does not lose more than 5% accuracy when as many as 2,600 parameters out of 2.5 million are corrupted by *random faults*. However, this analysis is limited to a specific scenario and only considers accidental errors rather than attacker-induced corruptions by means of fault attacks. The widespread usage of DNNs in many mission-critical systems, such as self-driving cars or aviation [69, 70], requires a comprehensive understanding of the security implications of such adversarial bitwise errors.

In this chapter, we explore the security properties of DNNs under bitwise errors that can be induced by practical hardware fault attacks. Specifically, we ask the question:

How vulnerable are DNNs to the atomic corruption

that a hardware fault attacker can induce?

Here, we focus on single bit-flip attacks that are realistic as they well-approximate the constrained memory corruption primitive of practical hardware fault attacks, such as Rowhammer [71]. To answer this question, we conduct a comprehensive study that characterizes the DNN model's responses to single-bit corruptions in each of its parameters.

First, we implement a systematic vulnerability analysis framework that flips each bit in a given model's parameters and measures the misclassification rates on a validation set. Using our framework, we analyze 19 DNN models; consisting of six different architectures and their variants on three popular image classification tasks—MNIST, CIFAR10, and ImageNet. Our experiments show that, on average, $\sim$50% of model parameters are vulnerable to single bit-flip corruptions, causing relative accuracy drops above 10%.

Further, all 19 DNN models contain parameters that can cause an accuracy drop of over 90%[1]. These show that adversarial bitwise errors can lead to a *graceless degradation* of classification accuracy; exposing the limits of DNNs' resilience to numerical changes.

Our framework also allows us to characterize the vulnerability by examining the impact of various factors: the bit position, bit-flip direction, parameter sign, layer width, activation function, normalization and model architecture. Our key findings include: 1) the vulnerability is caused by drastic spikes in a parameter's value; 2) the spikes in positive parameters are more threatening, however, an activation function that allows negative outputs renders the negative parameters vulnerable as well; 3) the number of vulnerable parameters increases proportionally as the DNN's layers get wider; 4) two common training techniques, e.g., dropout [72] and batch normalization [73], are ineffective in preventing the massive spikes bit-flips cause; and 5) the ratio of vulnerable parameters is almost constant across different architectures (*e.g.*, AlexNet, VGG16, and so on). Further, building on these findings, we propose heuristics for speeding up the analysis of vulnerable parameters in large models.

Second, to understand the practical impact of this vulnerability, we use Rowhammer [74] as an exemplary hardware fault attack. While a variety of hardware fault attacks are documented in literature [74, 75, 76, 77], Rowhammer is particularly amenable to practical, real-world exploitation. Rowhammer takes advantage of a widespread vulnerability in modern DRAM modules and provides an attacker with the ability to trigger controlled memory corruptions directly from unprivileged software execution. As a result, even a

[1]The vulnerability of a parameter requires a *specific* bit in its bitwise representation to be flipped. There also might be multiple such bits in the representation that, when flipped separately, trigger the vulnerability.

constrained Rowhammer-enabled attacker, who only needs to perform a specific memory access pattern, can mount practical attacks in a variety of real-world environments, including cloud [78, 79], browsers [71, 80, 81, 82], mobile [82, 83], and servers [84, 85].

We analyze the feasibility of Rowhammer attacks on DNNs by simulating a Machine-Learning-as-a-Service (MLaaS) scenario, where the victim and attacker VMs are co-located on the same host machine in the cloud. The co-location causes the victim and the attacker to share the same physical memory, enabling the attacker to trigger Rowhammer bit-flips in the victim's data [78, 79]. We focus our analysis on models with an applicable memory footprint, which can realistically be targeted by hardware fault attacks such as Rowhammer.

Our Rowhammer results show that in a *surgical* attack scenario, with the capability of flipping specific bits, the attacker can reliably cause severe accuracy drops in practical settings. Further, even in a *blind* attack scenario, the attacker can still mount successful attacks without any control over the memory locations of the bit-flips. Moreover, we also reveal a potential vulnerability in the *transfer learning* scenario; in which a surgical attack targets the parameters in the layers victim model contains in common with a public one.

Lastly, we discuss directions for viable protection mechanisms, such as reducing the number of vulnerable parameters by preventing significant changes in a parameter value. In particular, this can be done by (1) restricting activation magnitudes and (2) using low-precision numbers for model parameters via quantization or binarization. We show that, when we restrict the activations using the ReLU6 activation function, the ratio of vulnerable parameters decreases from 47% to 3% in AlexNet, and also, the accuracy drops are largely contained within 10%. Moreover, quantization and binarization reduce the vulnerable parameter ratio from 50% to 1–2% in MNIST. While promising, such solutions cannot deter practical hardware fault attacks in the general case, and often require training the victim model from scratch; hinting that more research is required towards fault attack-resilient DNNs.

3.1 Preliminaries

Here, we provide an overview of the required background knowledge.

3.1.1 Single Precision Floating Point Numbers

The parameters of a DNN model are usually represented as IEEE754 32-bit single-precision floating-point numbers. This format leverages the exponential notation and trades off the large range of possible values for reduced precision. For instance, the number 0.15625 in exponential notation is represented as 1.25×2^{-3}. Here, 1.25 expresses the *mantissa*; whereas -3 is the *exponent*. The IEEE754 single-precision floating-point format defines 23 bits to store the mantissa, 8 bits for the exponent, and one bit for the sign of the value. The fact that different bits have different influence on the represented value makes this format interesting from an adversarial perspective. For instance, continuing or example, flipping the 16th bit in the mantissa increases the value from 0.15625 to 0.15625828; hence, a usually negligible. On the other hand, a flipping the highest exponent bit would turn the value into 1.25×2^{125}. Although both of these rely on the same bit corruption primitive, they yield vastly different results. In Section 3.3, we analyze how this might lead to a vulnerability when a DNN's parameters are corrupted via single bit-flips.

3.1.2 Rowhammer Attacks

Rowhammer is the most common instance of software-induced fault attacks [71, 78, 79, 80, 81, 82, 83, 85]. This vulnerability provides an aggressor with a single-bit corruption primitive at DRAM level; thus, it is an ideal attack for the purpose of our analysis. Rowhammer is a remarkably versatile fault attack since it only requires an attacker to be able to access content in DRAM; an ubiquitous feature of every modern system. By simply carrying out specific memory access patterns—which we explain in Section 3.4—the attacker is able to cause extreme stress on other memory locations triggering faults on other stored data.

3.2 Threat Model

Prior work has extensively validated a DNN's resilience to parameter changes [31, 32, 34, 42, 43, 67], by considering random or deliberate perturbations. However, from a security perspective, these results provide only limited insights as they study a network's expected performance under cumulative changes. In contrast, towards a successful and feasible attack, an adversary is usually interested in inflicting the worst-case damage under minimal changes.

We consider a class of modifications that an adversary, using hardware fault attacks, can induce in practice. We assume a cloud environment where the victim's deep learning system is deployed inside a VM—or a container—to serve the requests of external users.

For making test-time inferences, the trained DNN model and its parameters are loaded into the system's (shared) memory and remain constant in normal operation. Recent studies describe this as a typical scenario in MLaaS [86].

To understand the DNN's vulnerability in this setup, we consider the atomic change that an attacker may induce—the single bit-flip—and we, in Section 3.3, systematically characterize the damage such change may cause. We then, in Section 3.4, investigate the feasibility of inducing this damage in practice, by considering adversaries with different capabilities and levels of knowledge.

Capabilities. We consider an attacker *co-located* in the same physical host machine as the victim's deep learning system. The attacker, due to co-location, can take advantage of a well-known software-induced fault attack, Rowhammer [78, 79], for corrupting the victim model stored in DRAM. We take into account two possible scenarios: 1) a *surgical* attack scenario where the attacker can cause a bit-flip at an intended location in the victim's process memory by leveraging advanced memory massaging primitives [78, 83] to obtain more precise results; and 2) a *blind* attack where the attacker lacks fine-grained control over the bit-flips; thus, is completely unaware of where a bit-flip lands in the layout of the model.

Knowledge. Using the existing terminology, we consider two levels for the attacker's knowledge of the victim model, e.g., the model's architecture and its parameters as well as their placement in memory: 1) a *black-box* setting where the attacker has no knowledge of the victim model. Here, both the surgical and blind attackers only hope to trigger an accuracy drop as they cannot anticipate what the impact of their bit-flips would be;

and 2) a *white-box* setting where the attacker knows the victim model, at least partially. Here, the surgical attacker can deliberately tune the attack's inflicted accuracy drop—from minor to catastrophic damage. Optionally, the attacker can force the victim model to misclassify a specific input sample without significantly damaging the overall accuracy. However, the blind attacker gains no significant advantage over the black-box scenario as the lack of capability prevents the attacker from acting on the knowledge.

3.3 Single-Bit Corruptions on DNNs

In this section, we expose the vulnerability of DNNs to single bit-flips. We start with an overview of our experimental setup and methodology. We then present our findings on DNNs' vulnerability to single bit corruptions. For characterizing the vulnerability, we analyze the impact of 1) the bitwise representation of the corrupted parameter, and 2) various DNN properties; on the resulting *indiscriminate damage*[2]. We also discuss the broader implications of the vulnerability for both the blind and surgical attackers. Finally, we turn our attention to two distinct attack scenarios single bit-flips lead to.

3.3.1 Experimental Setup and Methodology

Our vulnerability analysis framework systematically flips the bits in a model, individually, and quantifies the impact using the metrics we define. We implement the framework using Python 3.7[3] and PyTorch 1.0[4] that supports CUDA 9.0 for accelerating computations by

[2]We use this term to indicate the severe overall accuracy drop in the model.
[3]`https://www.python.org`
[4]`https://pytorch.org`

using GPUs. Our experiments run on the high performance computing cluster that has 488 nodes, where each is equipped with Intel E5-2680v2 2.8GHz 20-core processors, 180 GB of RAM, and 40 of which have 2 Nvidia Tesla K20m GPUs. We achieve a significant amount of speed-up by leveraging a parameter-level parallelism.

Datasets. We use three popular image classification datasets: MNIST [87], CIFAR10 [88], and ImageNet [2]. MNIST is a grayscale image dataset used for handwritten digits (zero to nine) recognition, containing 60,000 training and 10,000 validation images of 28x28 pixels. CIFAR10 and ImageNet are colored image datasets used for object recognition. CIFAR10 includes 32x32 pixels, colored natural images of 10 classes, containing 50,000 training and 10,000 validation images. For ImageNet, we use the ILSVRC-2012 subset [89], resized at 224x224 pixels, composed of 1,281,167 training and 50,000 validation images from 1,000 classes.

Models. We conduct our analysis on 19 different DNN models. For MNIST, we define a baseline architecture, Base (B), and generate four variants with different layer configurations: B-Wide, B-PReLU, B-Dropout, and B-DP-Norm. We also examine well-known LeNet5 (L5) [87] and test two variants of it: L5-Dropout and L5-D-Norm. Table 3.1 describes those architectures. For CIFAR10, we employ the architecture from [63] as a baseline and experiment on its three variants: B-Slim, B-Dropout and B-D-Norm. In the following sections, we discuss why we generate these variants. For CIFAR10, we also employ two off-the-shelf network architectures: AlexNet [91] and VGG16 [92]. For ImageNet, we use five well-known DNNs to understand the vulnerability of large models: AlexNet,

Base		Base (Wide)		Base (Dropout)		Base (PReLU)	
Layer Type	**Layer Size**	**Layer Type**	**Layer Size**	**Layer Type**	**Layer Size**	**Layer Type**	**Layer Size**
Conv (R)	5x5x10 (2)	Conv (R)	5x5x20 (2)	Conv (R)	5x5x10 (2)	Conv (P)	5x5x10 (2)
Conv (R)	5x5x20 (2)	Conv (R)	5x5x40 (2)	Conv (-)	5x5x20 (2)	Conv (P)	5x5x20 (2)
-	-	-	-	Dropout (R)	0.5	-	-
FC (R)	50	FC (R)	100	FC (R)	50	FC (P)	50
-	-	-	-	Dropout (R)	0.5	-	-
FC (S)	10	FC (S)	10	FC (S)	10	FC (S)	10

Base (D-BNorm)		LeNet5 [90]		LeNet5 (Dropout)		LeNet5 (D-BNorm)	
Layer Type	**Layer Size**	**Layer Type**	**Layer Size**	**Layer Type**	**Layer Size**	**Layer Type**	**Layer Size**
Conv (-)	5x5x10 (2)	Conv (R)	5x5x6 (2)	Conv (R)	5x5x6 (2)	Conv (-)	5x5x6 (2)
BatchNorm (R)	10	-	-	-	-	BatchNorm (R)	6
-	-	MaxPool (-)	2x2	MaxPool (-)	2x2	MaxPool (-)	2x2
Conv (-)	5x5x20 (2)	Conv (R)	5x5x16 (2)	Conv (R)	5x5x16 (2)	Conv (-)	5x5x16 (2)
BatchNorm (R)	20	-	-	-	-	BatchNorm (R)	16
-	-	MaxPool (-)	2x2	MaxPool (-)	2x2	MaxPool (-)	2x2
-	-	Conv (R)	5x5x120 (2)	Conv (R)	5x5x120 (2)	Conv (R)	5x5x120 (2)
-	-	-	-	-	-	BatchNorm (R)	120
Dropout (R)	0.5	-	-	Dropout (R)	0.5	Dropout (R)	0.5
-	-	MaxPool (-)	2x2	MaxPool (-)	2x2	MaxPool (-)	2x2
-	-	Conv (R)	5x5x120 (1)	Conv (R)	5x5x120 (1)	Conv (R)	5x5x120 (1)
FC (R)	50	FC (R)	84	FC (R)	84	FC (R)	84
Dropout (R)	0.5	-	-	Dropout (R)	0.5	Dropout (R)	0.5
FC (S)	10	FC (S)	10	FC (S)	10	FC (S)	10

Table 3.1: **Network Architectures used for MNIST.** We take two base architectures (Base and LeNet5) and make four and two variants from them, respectively. Note that we highlight the variations from the base architectures in red color.

VGG16, ResNet50 [93], DenseNet161 [94] and InceptionV3 [95][5].

Hyper-parameters. We use the following hyper-parameters for training. In MNIST, we use SGD, 40 epochs, 0.01 learning rate (lr), 64 batch, 0.1 momentum, and adjust learning rate by 0.1, in every 10 epochs. In CIFAR10, we use: (1) SGD, 50 epochs, 0.02 lr, 32 batch, 0.1 momentum, and adjust lr by 0.5, in every 10 epochs (for Base models); (2) 300 epochs, 0.01 lr, 64 batch, 0.1 momentum, and adjust lr by 0.95, in every 10 epochs (for AlexNet); and (3) 300 epochs, 0.01 lr, 128 batch, 0.1 momentum, and adjust lr by 0.15, in every 100 epochs (for VGG16). In our experiments on transfer learning scenarios (Sec 3.3.6), we fine-tune VGG16 pre-trained on ImageNet, on GTSRB, using: SGD, 40

[5]The pre-trained ImageNet models we use are available at:
`https://pytorch.org/docs/stable/torchvision/models.html`.

epochs, 0.01 lr, 32 batch, 0.1 momentum, and adjust lr by 0.1 and 0.05, in 15 and 25 epochs. We fine-tune ResNet50 pre-trained on ImageNet, on Flower102, using: SGD, 40 epochs, 0.01 lr, 50 batch, 0.1 momentum, and adjust lr by 0.1, in 15 and 25 epochs. In both scenarios, we freeze the parameters of the first 10 layers.

Metrics. To quantify the indiscriminate damage of single bit-flips, we define the Relative Accuracy Drop as:

$$\text{RAD} = \frac{(\text{Acc}_{pristine} - \text{Acc}_{corrupted})}{\text{Acc}_{pristine}}$$

where $\text{Acc}_{pristine}$ and $\text{Acc}_{corrupted}$ denote the classification accuracies of the pristine and the corrupted models, respectively. In our experiments, we use [RAD$>$ 0.1] as the criterion for indiscriminate damage on the model. We also measure the accuracy of each class in the validation set to analyze whether a single bit-flip causes a subset of classes to dominate the rest. In MNIST and CIFAR10, we simply compute the Top-1 accuracy on the test data (as a percentage) and use the accuracy for analysis. For ImageNet, we consider both the Top-1 and Top-5 accuracy; however, for the sake of comparability, we report only Top-1 accuracy in Table 3.2. We consider a parameter as *vulnerable* if it, in its bitwise representation, contains *at least one bit* that triggers severe indiscriminate damage when flipped. For quantifying the vulnerability of a model, we simply count the number of these vulnerable parameters.

Methodology. On our 8 MNIST models, we carry out a complete analysis: we flip each bit in all parameters of a model, in both directions—(0→1) and (1→0)—and compute the RAD over the entire validation set. However, a complete analysis of the larger models requires infeasible computational time—the VGG16 model for ImageNet with 138M parameters would take ≈ 942 days on our setup. Therefore, based on our initial results, we devise three speed-up heuristics that aid the analysis of CIFAR10 and ImageNet models.

Speed-up heuristics. The following three heuristics allow us to feasibly and accurately estimate the vulnerability in larger models:

• *Sampled validation set (SV).* After a bit-flip, deciding whether the bit-flip leads to a vulnerability [RAD$>$ 0.1] requires testing the corrupted model on the validation set; which might be cost prohibitive. This heuristic says that we can still estimate the model accuracy—and the RAD—on a sizable subset of the validation set. Thus, we randomly sample 10% the instances from each class in the respective validation sets, in both CIFAR10 and ImageNet experiments.

• *Inspect only specific bits (SB).* In Sec 3.1, we showed how flipping different bits of a IEEE754 floating-point number results in vastly different outcomes. Our the initial MNIST analysis in Sec 3.3.3 shows that mainly the exponent bits lead to perturbations strong enough to cause indiscriminate damage. This observation is the basis of our *SB* heuristic that tells us to examine the effects of flipping only the exponent bits for CIFAR10 models. For ImageNet models, we use a stronger SB heuristic and only inspect the most significant exponent bit of a parameter to achieve a greater speed-up. This heuristic causes us to miss the vulnerability the remaining bits might lead to, therefore, its results can be interpreted as a conservative estimate of the actual number of vulnerable parameters.

• *Sampled parameters (SP) set.* Our MNIST analysis also reveals that almost 50% of all parameters are vulnerable to bit-flips. This leads to our third heuristic: uniformly sampling from the parameters of a model would still yield an accurate estimation of the vulnerability. We utilize the SP heuristic for ImageNet models and uniformly sample a fixed number of parameters—20,000—from all parameters in a model. In our experiments, we perform this sampling five times and report the average vulnerability across all runs. Uniform sampling also reflects the fact that a black-box attacker has a uniform probability of corrupting any parameter.

3.3.2 Quantifying the Vulnerability That Leads to Indiscriminate Damage

Table 3.2 presents the results of our experiments on single-bit corruptions, for 19 different DNN models. We reveal that an attacker, armed with a single bit-flip attack primitive, can successfully cause indiscriminate damage [RAD> 0.1] and that the ratio of vulnerable parameters in a model varies between 40% to 99%; depending on the model. The consistency between MNIST experiments, in which we examine every possible bit-flip, and the rest, in which we heuristically examine only a subset, shows that, in a DNN model, approximately half of the parameters are vulnerable to single bit-flips. Our experiments also show small variability in the chances of a successful attack—indicated by the ratio of vulnerable parameters. With 40% vulnerable parameters, the InceptionV3 model is the most apparent outlier among the other ImageNet models; compared to 42-49% for the rest. Here, we define the vulnerability based on [RAD> 0.1].

Dataset	Network	Base acc.	# Params	Speed-up heuristics			Vulnerablility	
				SV	SB	SP	# Params	Ratio
MNIST	B(ase)	95.71	21,840	✗	✗	✗	10,972	50.24%
	B-Wide	98.46	85,670	✗	✗	✗	42,812	49.97%
	B-PReLU	98.13	21,843	✗	✗	✗	21,663	99.18%
	B-Dropout	96.86	21,840	✗	✗	✗	10,770	49.35%
	B-DP-Norm	97.97	21,962	✗	✗	✗	11,195	50.97%
	L5	98.81	61,706	✗	✗	✗	28,879	46.80%
	L5-Dropout	98.72	61,706	✗	✗	✗	27,806	45.06%
	L5-D-Norm	99.05	62,598	✗	✗	✗	30,686	49.02%
CIFAR10	B(ase)	83.74	776,394	✓(83.74)	✓(exp.)	✗	363,630	46.84%
	B-Slim	82.19	197,726	✓(82.60)	✓(exp.)	✗	92,058	46.68%
	B-Dropout	81.18	776,394	✓(80.70)	✓(exp.)	✗	314,745	40.54%
	B-D-Norm	80.17	777,806	✓(80.17)	✓(exp.)	✗	357,448	45.96%
	AlexNet	83.96	2,506,570	✓(85.00)	✓(exp.)	✗	1,185,957	47.31%
	VGG16	91.34	14,736,727	✓(91.34)	✓(exp.)	✗	6,812,359	46.23%
ImageNet	AlexNet	56.52 / 79.07	61,100,840	✓(51.12 / 75.66)	✓(31st bit)	✓(20,000)	9,467$^{\mathrm{SP}}$	47.34%
	VGG16	79.52 / 90.38	138,357,544	✓(64.28 / 86.56)	✓(31st bit)	✓(20,000)	8,414$^{\mathrm{SP}}$	42.07%
	ResNet50	76.13 / 92.86	25,610,152	✓(69.76 / 89.86)	✓(31st bit)	✓(20,000)	9,565$^{\mathrm{SP}}$	47.82%
	DenseNet161	77.13 / 93.56	28,900,936	✓(72.48 / 90.94)	✓(31st bit)	✓(20,000)	9,790$^{\mathrm{SP}}$	48.95%
	InceptionV3	69.54 / 88.65	27,197,488	✓(65.74 / 86.24)	✓(31st bit)	✓(20,000)	8,161$^{\mathrm{SP}}$	40.84%

SV = Sampled Validation set   SB = Specific Bits   SP = Sampled Parameters set

Table 3.2: **Indiscriminate damages to 19 DNN models caused by single bit-flips.**

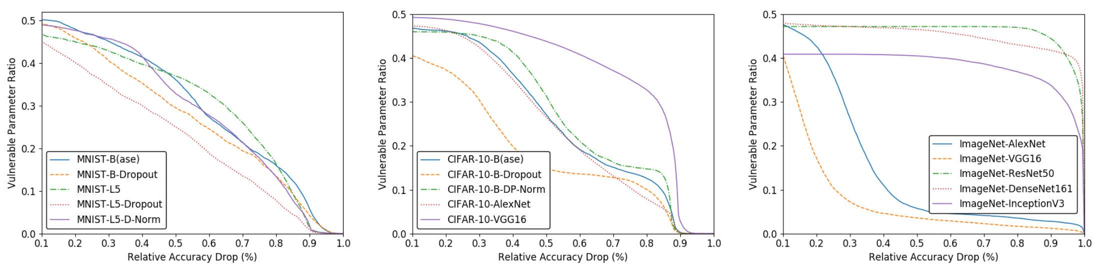


Figure 3.1: **The vulnerability of 15 DNN models using different criteria.** We plot the vulnerable parameter ratio based on the different RADs that an attacker aims; five from MNIST (left), five from CIFAR10 (middle), and five from ImageNets (right).

We also examine how the vulnerability changes within the range [$0.1 \leqslant$ RAD$\leqslant 1$]. Figure 3.1 describes the vulnerability on a specific RAD criterion; for instance, in MNIST-L5, the model has 40% of vulnerable parameters that cause [RAD> 0.5], which estimates the upper bound of the blind attacker. In MNIST, CIFAR10, and two ImageNet models, the vulnerability decreases as the attacker aims to inflict the severe damage. However, in ImageNet, ResNet50, DenseNet161, and InceptionV3 have almost the same vulnerability ($\sim$50%) even with the high criterion [RAD> 0.8]. In the following subsections, we characterize the vulnerability in relation to various factors and discuss our results in detail.

3.3.3 Characterizing the Vulnerability: Bitwise Representation

We characterize the interaction how the features of a parameter's bitwise representation govern its vulnerability.

Impact of the bit-flip position. To examine how much change in a parameter's value leads to indiscriminate damage, we focus on the position of the corrupted bits. In Figure 3.2, for each bit position, we present the number of bits that causes indiscriminate damage when flipped, on MNIST-L5 and CIFAR10-AlexNet models. In our MNIST experiments,

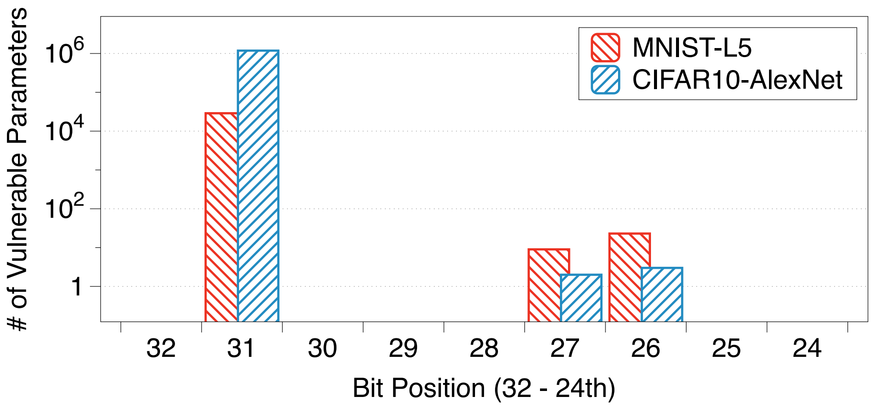


Figure 3.2: **The impact of the bit position.** This plot shows the number of vulnerable parameters in bit positions from 32nd to 24th in the log-scale.

we examine all bit positions and we observe that bit positions other than the exponents mostly do not lead to significant damage; therefore, we only consider the exponent bits. We find that *the exponent bits, especially the 31st-bit, lead to indiscriminate damage*. The reason is that a bit-flip in the exponents causes to a drastic change of a parameter value, whereas a flip in the mantissa only increases or decreases the value by a small amount—$[0, 1]$. We also observe that flipping the 30th to 28th bits is mostly inconsequential as these bits, in the IEEE754 representation, are already set to one for most values a DNN parameter usually takes—[3.0517×10^{-5}, 2].

Direction	**Models (M: MNIST, C: CIFAR10)**				
(32-24th bits)	**M-B**	**M-PReLU**	**M-L5**	**C-B**	**C-AlexNet**
(0→1)	11,019	21,711	28,902	314,768	1,185,964
(1→0)	0	0	0	0	0
Total	11,019	21,711	28,902	314,768	1,185,964

Table 3.3: **The impact of the flip direction.** The number of effective bit-flips in three MNIST and two CIFAR10 models.

Impact of the flip direction. We answer which direction of the bit-flip, (0→1) or (1→0), leads to greater indiscriminate damage. In Table 3.3, we report the number of effective bit-flips, i.e., those that inflict [RAD ¿ 0.1] for each direction, on 3 MNIST and 2 CIFAR10 models. We observe that *only (0→1) flips cause indiscriminate damage* and *no (1→0) flip leads to vulnerability*. The reason is that a (1→0) flip can only decrease a parameter's value, unlike a (0→1) flip. The values of model parameters are usually normally distributed—$N(0, 1)$—that places most of the values within [-1,1] range. Therefore, a (1→0) flip, in the exponents, can decrease the magnitude of a typical parameter at most by one; which is not a strong enough change to inflict critical damage. Similarly, in the sign bit, both (0→1) and (1→0) flips cannot cause severe damage because they change the magnitude of a parameter at most by two. On the other hand, a (0→1) flip, in the exponents, can increase the parameter value significantly; thus, during the forward-pass, the extreme neuron activation caused by the corrupted parameter overrides the rest of the activations.

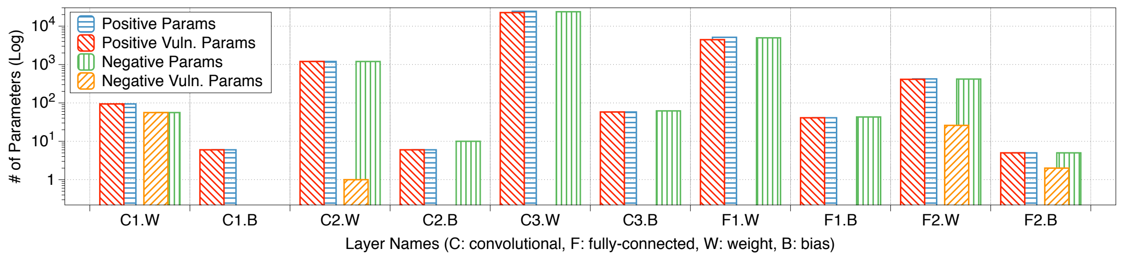


Figure 3.3: **The impact of the parameter sign.** The number of vulnerable positive and negative parameters, in each layer of MNIST-L5.

Impact of the parameter sign. As our third feature, we investigate whether the sign—positive or negative—of the corrupted parameter impacts the vulnerability. In Figure 3.3, we examine the MNIST-L5 model and present the number of vulnerable positive and negative parameters in each layer—in the log-scale. Our results suggest that *positive* *parameters are more vulnerable to single bit-flips than negative parameters*. We identify the common ReLU activation function as the reason: ReLU immediately zeroes out the negative activation values, which are usually caused by the negative parameters. As a result, the detrimental effects of corrupting a negative parameter fail to propagate further in the model. Moreover, we observe that *in the first and last layers, the negative parameters, as well as the positive ones, are vulnerable*. We hypothesize that, in the first convolutional layer, changes in the parameters yield a similar effect to corrupting the model inputs directly. On the other hand, in their last layers, DNNs usually have the Softmax function that does not have the same zeroing-out effect as ReLU.

3.3.4 Characterizing the Vulnerability: DNN Properties

We continue our analysis by investigating how various properties of a DNN model affect the model's vulnerability to single bit-flips.

Impact of the layer width. We start our analysis by asking whether increasing the width of a DNN affects the number of vulnerable parameters. In Table 3.2, in terms of the number of vulnerable parameters, we compare the MNIST-B model with the MNIST-B-Wide model. In the wide model, all the convolutional and fully-connected layers are twice as wide as the corresponding layer in the base model. We see that the ratio of vulnerable parameters is almost the same for both models: 50.2% vs 50.0%. Further, experiments on the CIFAR10-B-Slim and CIFAR10-B—twice as wide as the slim model—produce consistent results: 46.7% and 46.8%. We conclude that *the number of vulnerable parameters grows proportionally with the DNN's width* and, as a result, *the ratio of vulnerable*

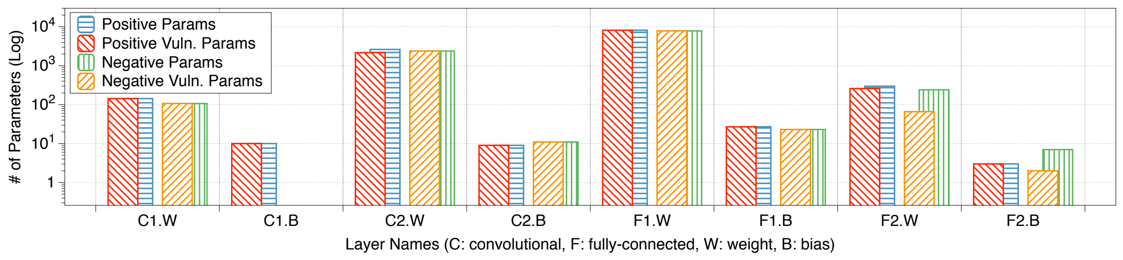


Figure 3.4: **The impact of the activation function.** The number of vulnerable positive and negative parameters, in each layer of MNIST-PReLU.

parameters remains constant at around 50%.

Impact of the activation function. Next, we explore whether the choice of activation function affects the vulnerability. Previously, we showed that ReLU can neutralize the effects of large negative parameters caused by a bit-flip; thus, we experiment on different activation functions that allow negative outputs, e.g., PReLU [96], LeakyReLU, or RReLU [97]. These ReLU variants have been shown to improve the training performance and the accuracy of a DNN. In this experiment, we train the MNIST-B-PReLU model; which is exactly the same as the MNIST-B model, except that it replaces ReLU with PReLU. Figure 3.4 presents the layer-wise number of vulnerable positive and negative parameters in MNIST-B-PReLU. We observe that *using PReLU causes the negative parameters to become vulnerable and, as a result, leads to a DNN approximately twice as vulnerable as the one that uses ReLU*—50.2% vs. 99.2% vulnerable parameters.

Impact of dropout and batch normalization. We confirmed that successful bit-flip attacks increase a parameter's value drastically to cause indiscriminate damage. In consequence, we hypothesize that common techniques that tend to constrain the model parameter values to improve the performance, e.g., dropout [72] or batch normalization [73], would result

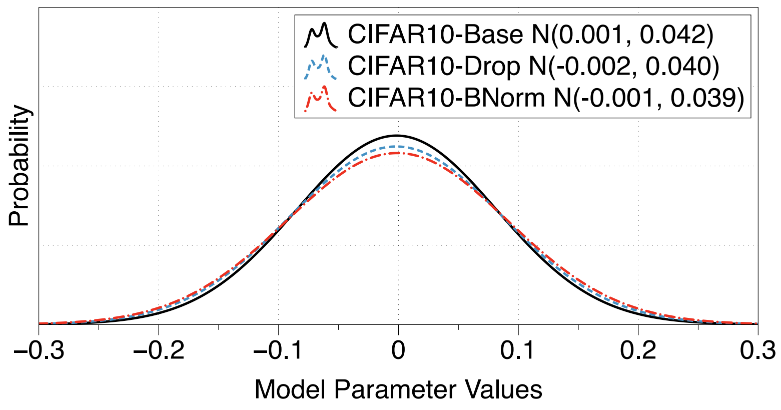


Figure 3.5: **The impact of the dropout and batch normalization.** The distributions of the parameter values of three CIFAR10 models variants.

in a model more resilient to single bit-flips. Besides the base CIFAR10 and MNIST models, we train the B-Dropout and B-DNorm models for comparison. In B-Dropout models, we apply dropout before and after the first fully-connected layers; in B-DNorm models, in addition to dropout, we also apply batch normalization after each convolutional layer. In Figure 3.5, we compare our three CIFAR10 models and show how dropout and batch normalization have the effect of reducing the parameter values. However, when we look into the vulnerability of these models, we surprisingly find that *the vulnerability is mostly persistent regardless of dropout or batch normalization*—with at most 6.3% reduction in vulnerable parameter ratio over the base network.

Impact of the model architecture. Table 3.2 shows that the vulnerable parameter ratio is mostly consistent across different DNN architectures. However, we see that the InceptionV3 model for ImageNet has a relatively lower ratio—40.8%—compared to the other models—between 42.1% and 48.9%. We hypothesize that the reason is the auxiliary classifiers in the InceptionV3 architecture that have no function at test-time. To confirm our hypothesis, we simply remove the parameters in the auxiliary classifiers; which bring the vulnerability ratio closer to the other models—46.5%. Interestingly, we also observe that the parameters in batch normalization layers are resilient to a bit-flip: corrupting `running_mean` and `running_var` cause negligible damage. In consequence, excluding the parameters in InceptionV3's multiple batch normalization layers leads to a slight increase in vulnerability—by 0.02%.

3.3.5 Implications for the Adversaries

In Sec 3.2, we defined four attack scenarios: the blind and surgical attackers, in the black-box and white-box settings. First, we consider the strongest attacker: the surgical, who can flip a bit at a specific memory location; white-box, with the model knowledge for anticipating the impact of flipping the said bit. To carry out the attack, this attacker identifies: 1) how much indiscriminate damage, the RAD goal, she intends to inflict, 2) a vulnerable parameter that can lead to the RAD goal, 3) in this parameter, the bit location, e.g., 31st-bit, and the flip direction, e.g., (0→1), for inflicting the damage. Based on our [RAD>0.1] criterion, approximately 50% of the parameters are vulnerable in all models; thus, for this goal, the attacker can easily achieve 100% success rate. For more severe goals [$0.1 \leqslant$RAD$\leqslant 0.9$], our results in Appendix **??** suggest that the attacker can still find vulnerable parameters. In Sec 3.4.3, we discuss the necessary primitives, in a practical setting, for this attacker.

For a black-box surgical attacker, on the other hand, the best course of action is to target the 31st-bit of a parameter. This strategy maximizes the attacker's chance of causing indiscriminate damage, even without knowing what, or where, the corrupted

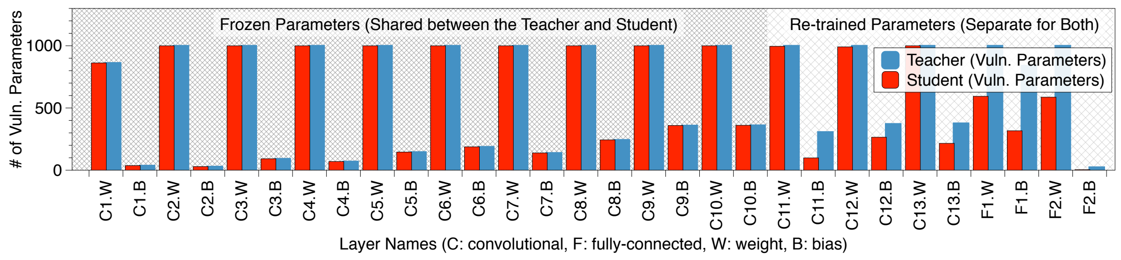


Figure 3.6: **The security threat in a transfer learning scenario.** The victim model—student—that is trained by transfer learning is vulnerable to the surgical attacker, who can see the parameters the victim has in common with the teacher model.

parameter is. Considering, the VGG16 model for ImageNet, the attack's success rate is 42.1% as we report in Table 3.2; which is an upper-bound for the black-box attackers. For the weakest—black-box blind—attacker that cannot specifically target the 31st-bit, we conservatively estimate the lower-bound as 42.1% / 32-bits = 1.32%; assuming only the 31st-bits lead to indiscriminate damage. Note that the success rate for the white-box blind attacker is still 1.32% as acting upon the knowledge of the vulnerable parameters requires an attacker to target specific parameters. In Sec 3.4.4, we evaluate the practical success rate of a blind attacker.

3.3.6 Distinct Attack Scenarios

In this section, other than causing indiscriminate damage, we discuss two distinct attack scenarios single bit-flips might enable: transfer learning and targeted misclassification.

Transfer learning scenario. Transfer learning is a common technique for *transferring* the knowledge in a pre-trained *teacher* model to a *student* model; which, in many cases, outperforms training a model from scratch. In a practical scenario, a service provider might rely on publicly available teacher as a starting point to train commercial student models. The teacher's knowledge is transferred by *freezing* some of its layers and embedding them into the student model; which, then, trains the remaining layers for its own task. The security risk is that, for an attacker who knows the teacher but not the student, a black-box attack on the student might escalate into a white-box attack on the teacher's frozen layers. The attacker first downloads the pre-trained teacher from the Internet. She then loads the teacher into the memory and waits for the *deduplication* [98] to happen. During deduplication, the memory pages with the same contents—the frozen layers—are merged into the shared pages between the victim and attacker. This essentially promotes a blind threat to the victim's memory to a stronger surgical threat to the attacker's own memory. In consequence, a bit-flip in the attacker's own pages can also affect the student model in the victim's memory.

We hypothesize that an attacker, who can identify the teacher's vulnerable parameters and trigger bit-flips in these parameters, can cause indiscriminate damage to the student model. In our experiments, we examine two transfer learning tasks in [99]: the traffic sign (GTSRB) [100] and flower recognition (Flower102) [101]. We initialize the student model by transferring first ten frozen layers of the teacher—VGG16 or ResNet50 on ImageNet. We then append a new classification layer and train the resulting student network for its respective task by only updating the new unfrozen layer. We corrupt the 1,000 parameters sampled from each layer in the teacher and monitor the damage to the student model. Figure 3.6 reports our results: we find that *all vulnerable parameters in the frozen layers and more than a half in the re-trained layers are shared by the teacher and the student*.

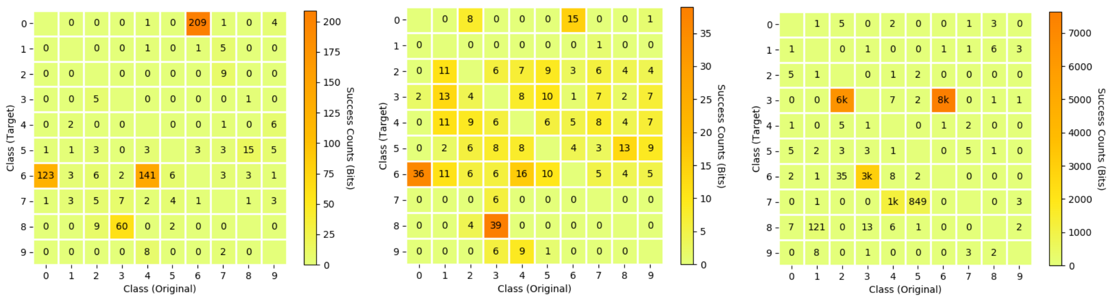


Figure 3.7: **The vulnerable parameters for a targeted attack in 3 DNN models.** Each cell reports the number of bits that lead to the misclassification of a target sample, whose original class is given by the x-axis, as the target class, which is given by the y-axis. From left to right, the models are MNIST-B, MNIST-L5 and CIFAR10-AlexNet.

Targeted misclassification. Our main focus is showing DNNs' graceless degradation, but we conduct an additional experiment and ask whether a single bit-flip primitive could be used in the context of *targeted misclassification* attacks. A targeted attack aims to preserve the victim model's overall accuracy while causing it to misclassify a specific target sample into the target class. We experiment with a target sample from each class in MNIST or CIFAR10—we use MNIST-B, MNIST-L5 and CIFAR10-AlexNet models. Our white-box surgical attacker also preserves the accuracy by limiting the [RAD< 0.05] as in [63]. We find that *the number of vulnerable parameters for targeted misclassifications is lower than that of for causing indiscriminate damage*. In Figure 3.7, we also see that for some *(original–target class) pairs*, the vulnerability is more evident. For example, in MNIST-B, there are 141 vulnerable parameters for (class 4–class 6) and 209 parameters for (class 6–class 0). Simlarly, in CIFAR10-AlexNet, there are 6,000 parameters for (class 2–class 3); 3,000 parameters for (class 3–class 6); and 8,000 parameters for (class 6–class 3).

3.4 Exploiting The Vulnerability Using Rowhammer

In order to corroborate the analysis made in Sec 3.3 and prove the viability of hardware fault attacks against DNN, we test the resiliency of these models against Rowhammer. At a high level, Rowhammer is a software-induced fault attack that provides the attacker with a single-bit write primitive to specific physical memory locations. That is, an attacker capable of performing specific memory access patterns (at DRAM-level) can induce persistent and repeatable bit corruptions from software. Given that we focus on single-bit perturbations on DNN's parameters in practical settings, Rowhammer represents the perfect candidate for the task.

3.4.1 Backgrounds on Rowhammer

DRAM internals. In Figure 3.8, we show the internals of a DRAM *bank*. A bank is a bi-dimensional array of memory *cells* connected to a *row buffer*. Every DRAM chip contains multiple banks. The cells are the actual storage of one's data. They contain a capacitor whose charge determines the value of a specific bit in memory. When a read is issued to

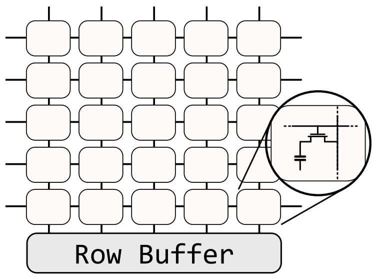


Figure 3.8: **DRAM bank structure.** Zoom-in on a cell containing the capacitor storing data.

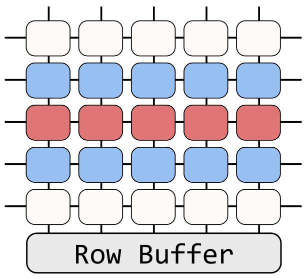


Figure 3.9: **Double-sided Rowhammer.** Aggressor rows in blue color, and a victim row is colored in red
.

a specific row, this row gets *activated*, which means that its content gets transferred to the row buffer before being sent to the CPU. Activation is often requested to recharge a row's capacitors (i.e., *refresh* operation) since they leak charge over time.

Rowhammer mechanism. Rowhammer is a DRAM disturbance error that causes spurious bit-flips in DRAM cells generated by frequent activations of a neighboring row. Here, we focus on double-sided Rowhammer, the most common and effective Rowhammer variant used in practical attacks [78, 82, 83]. Figure 3.9 exemplifies a typical double-sided Rowhammer attack. The victim's data is stored in a row enclosed between two aggressor rows that are repeatedly accessed by the attacker. Due to the continuous activations of the neighboring rows, the victim's data is under intense duress. Thus, there is a large probability of bit-flips on its content.

To implement such attack variant, the attacker usually needs some knowledge or control over the physical memory layout. Depending on the attack scenario, a Rowhammer-enabled attacker can rely on a different set of primitives for this purpose. In our analysis, we consider two possible scenarios: 1) we initially consider the *surgical* attacker; that is, an attacker with the capability of causing bit-flips at the specific locations, and we demonstrate how, under these assumptions, she can induce indiscriminate damage to a co-located DNN application. 2) We then deprive the attacker of this ability to analyze the outcome of a *blind* attacker and we demonstrate that, even in a more restricted environment, the attacker can still cause indiscriminate damage by causing bitwise corruptions.

DRAM	# (0→1) flips	DRAM	# (0→1) flips
A_2	21,538	A_4	5,577
E_2	16,320	I_1	4,781
H_1	10,608	J_1	4,725
G_1	7,851	E_1	4,175
A_1	4,367	A_3	1,541
F_1	5,927	C_1	1,365

Table 3.4: **Hammertime database** [1]. We report the number of (0→1) bit-flips in 12 different DRAM setups. (The rows in gray are used for the experiments in Figure 3.10.)

3.4.2 Experimental Setup

For our analysis, we constructed a simulated environment[6] relying on a database of the Rowhammer vulnerability in 12 DRAM chips, provided by Tatar et al [1]. Different memory chips have a different degree of susceptibility to the Rowhammer vulnerability, enabling us to study the impact of Rowhammer attacks on DNNs in different real-world scenarios. Table 3.4 reports the susceptibility of the different memory chips to Rowhammer. Here, we only include the numbers for (0→1) bit-flips since these are the more interesting ones for the attacker targeting a DNN model according to our earlier analysis in Sec 3.3.3 and Sec 3.3.4.

We perform our analysis on an exemplary deep learning application implemented in PyTorch, constantly querying an ImageNet model. We use ImageNet models since we focus on a scenario where the victim has a relevant memory footprint that can be realistically be targeted by hardware fault attacks such as Rowhammer in practical settings.

[6]Note that we first implemented all the steps described in our paper on a physical system, considering using end-to-end attacks for our analysis. After preliminary testing of this strategy on our own DRAMs, we concluded it would be hard to generalize the findings of such an analysis and decided against it—in line with observations from prior work [1].

Network	**Vuln. Objects** *(Vuln./Total)*	**Vuln. Params** *(in 20k params)*	**#Hammer Attempts** *(min / med / max)*
AlexNet	7 / 16	9,522	4 / 64 / 4, 679
VGG16	12 / 32	8,140	4 / 64 / 4, 679
ResNet50	9 / 102	3,466	4 / 64 / 4, 679
DenseNet161	63 / 806	5,117	4 / 64 / 4, 679
InceptionV3	53 / 483	6,711	4 / 64 / 4, 679

Table 3.5: **Effectiveness of surgical attacks.** We examine five different ImageNet models analyzed in Sec 3.3.

While small models are also potential targets, the number of interesting locations to corrupt is typically limited to draw general conclusions on the practical effectiveness of the attack.

3.4.3 Surgical Attack Using Rowhammer

We start our analysis by discussing a surgical attacker, who has the capability of causing a bit-flip at the specific location in memory. The two surgical attackers are available: the attacker with the knowledge of the victim model (white-box) and without (black-box). However, in this section, we assume that the strongest attacker knows the parameters to compromise and is capable of triggering bit-flips on its corresponding memory location. Then, this attacker can take advantage of accurate memory massing primitives (e.g., memory deduplication) to achieve 100% attack success rate.

Memory templating. Since a surgical attacker knows the location of vulnerable parameters, she can *template* the memory up front [78]. That is, the attacker scans the memory by inducing Rowhammer bit-flips in her own allocated chunks and looking for exploitable bit-flips. A surgical attacker aims at specific bit-flips. Hence, while templating the memory, the attacker simplifies the scan by looking for bit-flips located at specific offsets from the start address of a memory *page* (i.e., 4 KB)—the smallest possible chunk allocated from the OS. This allows the attacker to find memory pages vulnerable to Rowhammer bit-flips at a given page offset (i.e., *vulnerable templates*), which they can later use to predictably attack the victim data stored at that location.

Vulnerable templates. To locate the parameters of the attacker's interest (i.e., vulnerable parameters) within the memory page, she needs to find *page-aligned* data in the victim model. Modern memory allocators improve performances by storing large objects (usually multiples of the page size) page-aligned whereas smaller objects are not. Thus, we first analyze the allocations performed by the PyTorch framework running on Python to understand if it performs such optimized page-aligned allocations for large objects similar to other programs [102, 103]. We discovered this to be the case for all the objects larger than 1 MB—i.e., our attacker needs to target the parameters such as weight, bias, and so on, stored as tensor objects in layers, larger than 1 MB.

Then, again focusing on the ImageNet models, we analyzed them to identify the objects that satisfy this condition. Even if the ratio between the total number of objects and target objects may seem often unbalanced in favor of the small ones[7], we found that the number of vulnerable parameters in the target objects is still significant (see Table 3.5). Furthermore, it is important to note that when considering a surgical attacker, she only needs one single vulnerable template to compromise the victim model, and there is only 1,024 possible offsets where we can store a 4-byte parameter within a 4 KB page.

[7]The `bias` in convolutional or dense layers, and the `running_mean` and `running_var` in batch-norms are usually the small objects (¡ 1 MB).

Memory massaging. After finding a vulnerable template, the attacker needs to *massage* the memory to land the victim's data on the vulnerable template. This can be achieved, for instance, by exploiting memory deduplication [78, 79, 81]. Memory deduplication is a system-level memory optimization that merges read-only pages for different processes or VMs when they contain the same data. These pages re-split when a write is issued to them. However, Rowhammer behaves as invisible bit-wise writes that do not trigger the spit, breaking the process boundaries. If the attacker knows (even if partially) the content of the victim model can take advantage of this merging primitive to compromise the victim service.

Experimental results. Based on the results of the experiments in Sec **??** and Sec **??**, we analyze the requirements for a surgical (white-box) attacker to carry out a successful attack. Here, we used one set of the five sampled parameters for each model. In Table 3.5, we report *min, median, and max* values of the number of rows that an attacker needs to hammer to find the first vulnerable template on the 12 different DRAM setups for each model. This provides a meaningful metric to understand the success rate of a surgical attack. As you can see in Table 3.5, the results remain unchanged among all the different models. That is, for every model we tested in the best case, it required us to hammer only 4 rows (*A_2* DRAM setup) to find a vulnerable template all the way up to 4,679 in the worst case scenario (*C_1*). The reason why the results are equal among the different models is *due to the number of vulnerable parameters which largely exceeds the number of possible offsets within a page that can store such parameters* (i.e., 1024). Since every vulnerable parameter yields indiscriminate damage [RAD$>$ 0.1], we simply need to identify a template that could match any given vulnerable parameter. This means that an attacker can find a vulnerable template at best in a matter of few seconds[8] and at worst still within minutes. Once the vulnerable template is found, the attacker can leverage memory deduplication to mount an effective attack against the DNN model—with no interference with the rest of the system.

3.4.4 Blind Attack Using Rowhammer

While in Sec 3.4.3 we analyzed the outcome of a surgical attack, here we abstract some of the assumptions made above and study the effectiveness of a blind attacker oblivious of the bit-flip location in memory. To bound the time of the lengthy blind Rowhammer attack analysis, we specifically focus our experiments on the ImageNet-VGG16 model.

We run our application under the pressure of Rowhammer bit-flips indiscriminately targeting both code and data regions of the process's memory. Our goal is twofold: (1) to understand the effectiveness of such attack vector in a less controlled environment and (2) to examine the robustness of a running DNN application to Rowhammer bit-flips by measuring the number of failures (i.e., crashes) that our blind attacker may inadvertently induce.

Capabilities. We consider a blind attacker who cannot control the bit-flips caused by Rowhammer. As a result, the attacker may corrupt bits in the DNN's parameters as well as the code blocks in the victim process's memory. In principle, since Rowhammer bit-flips propagate at the DRAM level, a fully blind Rowhammer attacker may also inadvertently

[8]We assume 200ms to hammer a row.

flip bits in other system memory locations. In practice, even an attacker with limited knowledge of the system memory allocator, can heavily influence the physical memory layout by means of specially crafted memory allocations [104, 105]. Since this strategy allows attackers to achieve co-location with the victim memory and avoid unnecessary fault propagation in practical settings, we restrict our analysis to a scenario where bit-flips can only (blindly) corrupt memory of the victim deep learning process. This also generalizes our analysis to arbitrary deployment scenarios, since the effectiveness of blind attacks targeting arbitrary system memory is inherently environment-specific.

Methods. For every one of the 12 vulnerable DRAM setups available in the database, we carried out 25 experiments where we performed at most 300 "hammering" attempts—value chosen after the surgical attack analysis where a median of 64 attempts was required. The experiment has three possible outcomes: (1) we trigger one(or more) effective bit-flip(s) that compromise the model, and we record the relative accuracy drop when performing our testing queries; (2) we trigger one(or more) effective bit-flip(s) in other victim memory locations that result in a crash of the deep learning process; (3) we reach the "timeout" value of 300 hammering attempts. We set such "timeout" value to bound our experimental analysis which would otherwise result too lengthy.

Experimental results. In Figure 3.10, we present the results for three sampled DRAM setups. We picked *A_2*, *I_1*, and *C_1* as representative samples since they are the most, least, and moderately vulnerable DRAM chips (see Table 3.4). Depending on the DRAM setup, we obtain fairly different results. We found *A_2* obtains successful indiscriminate damages to the model in 24 out of 25 experiments while, in less vulnerable environments

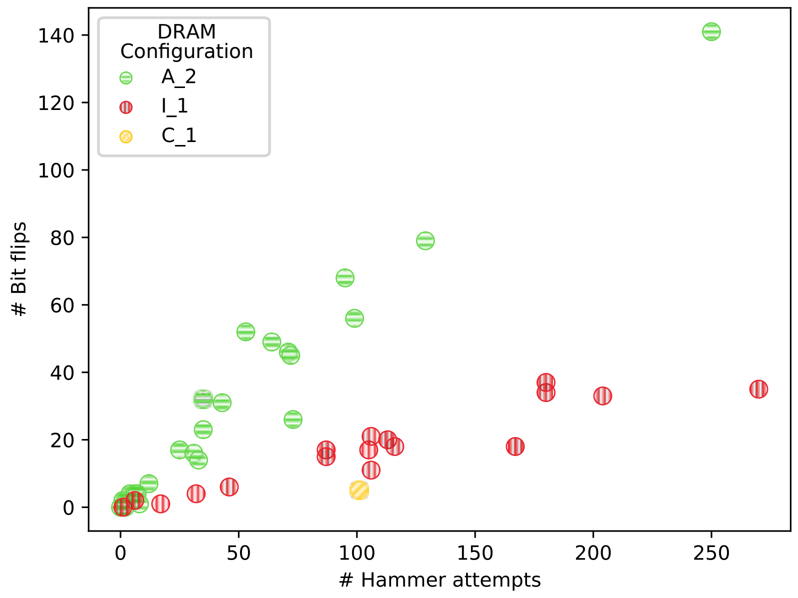


Figure 3.10: **The successful runs of a blind attack execution over three different DRAM setups (*A_2*-most, *I_1*-least, and *C_1*-moderately vulnerable).** We report the success in terms of $\#flips$ and $\#hammer\ attempts$ required to obtain an indiscriminate damage to the victim model. We observe the successes within few hammering attempts.

such as *C_1*, the number of successes decreases to only one while the other 24 times out. However, it is important to note that a timeout does not represent a negative result—a crash. Contrarily, while *C_1* only had a single successful attack, it also represents a peculiar case corroborating the analysis presented in Sec **??**. The corruption generated in this single successful experiment was induced by a single bit-flip, which caused one of the most significant RADs detected in the entire experiment, i.e., 0.9992 and 0.9959 in Top-1 and Top-5. Regardless of this edge case, we report a mean of 15.6 out of 25 effective attacks for this Rowhammer variant over the different DRAM setups. Moreover, we report the distribution of accuracy drops for Top-1 and Top-5 in Figure 3.11. In particular, the median drop for Top-1 and Top-5 confirms the claims made in the previous sections, i.e., the blind attacker can expect [RAD>0.1] on average.

Interestingly, when studying the robustness of the victim process to Rowhammer,

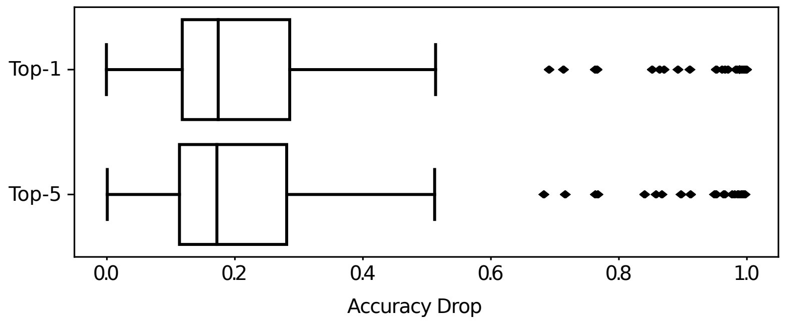


Figure 3.11: **The distribution of relative accuracy drop for Top-1 and Top-5.** We compute them over the effective $\#flips$ in our experiments on the ImageNet-VGG16 model.

we discovered it to be quite resilient to spurious bit-flips. We registered only 6 crashes over all the different DRAM configurations and experiments—300 in total. This shows that the model effectively dominates the memory footprint of the victim process and confirms findings from our earlier analysis that bit-flips in non-vulnerable model elements have essentially no noticeable impact.

3.4.5 Synopsis

Throughout the section, we analyzed the outcome of surgical and blind attacks against large DNN models and demonstrated how Rowhammer can be deployed as a feasible attack vector against these models. These results corroborate our findings in Sec 3.3 where we estimated at least 40% of a model's parameters to be vulnerable to single-bit corruptions. Due to this large attack surface, in Sec 3.4.3, we showed that a Rowhammer-enabled attacker armed with knowledge of the network's parameters and powerful memory massaging primitives [78, 79, 83] can carry out precise and effective indiscriminate attacks in a matter of, at most, few minutes in our simulated environment. Furthermore, this property, combined with the resiliency to spurious bit-flips of the (perhaps idle) code regions, allowed us to build successful blind attacks against the ImageNet-VGG16 model and inflict "terminal brain damage" even when the model is hidden from the attacker.

3.5 Discussion

In this section, we discuss and evaluate some potential mitigation mechanisms to protect against single-bit attacks on DNN models. We discuss two research directions towards making DNN models resilient to bit-flips: *restricting activation magnitudes* and *using low-precision numbers*. Prior work on defenses against Rowhammer attacks suggests system-level defenses [106, 107] that often even require specific hardware support [74, 108]. Yet they have not been widely deployed since they require infrastructure-wide changes from cloud host providers. Moreover, even though the infrastructure is resilient to Rowhammer attacks, an adversary can leverage other vectors to exploit bit-flip attacks for corrupting a model. Thus, we focus on the solutions that can be directly implemented on the defended DNN model.

3.5.1 Restricting Activation Magnitudes

In Sec 3.3.4, we showed that the vulnerable parameter ratio varies based on inherent properties of a DNN; for instance, using PReLU activation function causes a model to propagate negative extreme activations. We hypothesize that an activation function, which produces its output in a constrained range, would make indiscriminate damage harder to induce via bit-flips. There are several functions, such as Tanh or HardTanh [109],

Network	**Train**	**Base acc.**	**# Params**	**Vulnerability**
Base (ReLU)	Scr	98.13	21,840	10,972 (50.2%)
Base (ReLU6)	Scr	98.16		313 (1.4%)
Base (Tanh)	Scr	97.25		507 (2.3%)
Base (ReLU6)	Sub	95.71		542 (2.4%)
AlexNet (ReLU)	-	56.52 / 79.07	20,000 (61M)	9.467 (47.34%)
AlexNet (ReLU6)	Sub	39.80 / 65.82		560 (2.8%)
AlexNet (ReLUA)	Sub	56.52 / 79.07		1,063 (5.32%)
VGG16 (ReLU)	-	64.28 / 86.56	20,000 (138M)	8,227 (41.13%)
VGG16 (ReLU6)	Sub	38.58 / 64.84		2,339 (11.67%)
VGG16 (ReLUA)	Sub	64.28 / 86.56		2,427 (12.14%)

Table 3.6: **Effectiveness of restricting activation.**

that suppress the activations; however, using ReLU-6 [110] function provides two key advantages over the others: 1) the defender only needs to substitute the existing activation functions from ReLU to ReLU6 without re-training, and 2) ReLUA functions allow the defender to control the activation range by modifying the A, e.g., using $A > 6$ to minimize the performance loss the substitution causes. A defender can monitor the activation values over the validation set and to determine an activation range that only suppresses the abnormal values, potentially caused by bit-flips. In our experiments on ImageNet-AlexNet, we set the range as $[0, max]$, where max is determined adaptively by observing the maximum activation in each layer (ReLU-A).

Experiments. We use three DNN models in Sec 3.3: the MNIST-B, ImageNet-AlexNet, and ImageNet-VGG16 models. We evaluate four activation functions: ReLU (default), Tanh, ReLU6, and ReLUA (only for AlexNet and VGG16); and two training methods: training a model from scratch (*Scr*) or substituting the existing activation into another (*Sub*). In our notation, we denote the model's name together with its activation function, e.g., AlexNet (ReLU6) For larger models, we also rely on our speed-up heuristics for estimating the vulnerability.

Table 3.6 shows the effectiveness of this defensive mechanism. For each network (Column 1), we list the training method, the base accuracy, the number of sampled parameters, and the vulnerability (Column 2-5). We find that, in some cases, restricting activation magnitudes with Tanh and ReLU6 reduces the vulnerability. For instance, in the MNIST models, we observe that the ratio of vulnerable parameters drops to 1.4-2.4% from 50%; without incurring any significant performance loss. Further, we discover that the substitution with ReLU6 achieves a similar effect without re-training; however, it fails to prevent the vulnerability in the last layer, which uses Softmax instead of ReLU. In AlexNet and VGG16, we also observe a decrease in the ratio of vulnerable parameters—47.34% to 2.8% and 41.13% to 11.67%—; however, with significant loss of accuracy. To minimize the loss, we set the range of activation in AlexNet (ReLUA) and VGG16 (ReLUA) by selecting the maximum activation value in each layer. We see that ReLUA leads to a trade-off between the ratio of vulnerable parameters and the accuracy.

Takeaways. Our experimental results on restricting activation magnitudes suggest that this mechanism 1) allows a defender to control the trade-off between the relative accuracy drop and reducing the vulnerable parameters and 2) enables ad-hoc defenses to DNN models, which does not require training the network from scratch. However, the remaining number of vulnerable parameters shows that the Rowhammer attacker still could inflict damage, with a reduced success rate.

Network	Method	Base acc.	# Params	Vulnerability
L5	-	99.24	62,598	30,686 (49.0%)
L5	8-bit Quantized	99.03	62,600	0 (0.0%)
L5	XNOR Binarized	98.39	62,286	623 (1.0%)

Table 3.7: **Effectiveness of using low-precision.**

3.5.2 Using Low-precision Numbers

Another direction is to represent the model parameters as low-precision numbers by using quantization and binarization. In Sec 3.3.3, we found that the vulnerability exploits the bitwise representation of the corrupted parameter to induce the dramatic chances in the parameter's value. We hypothesize that the use of low-precision numbers would make a parameter more resilient to such changes. For example, an integer expressed as the 8-bit quantized format can be increased at most 128 by a flip in the most significant bit—8th bit. Therefore, the attacker only can cause restricted increases in a model parameter. Training models using low-precision numbers are supported by the popular deep learning frameworks such as TensorFlow[9]. The victim can train and deploy the model with quantized or binarized parameters on these frameworks.

Experiments. To test our hypothesis, we use 3 DNN models: the MNIST-L5 (baseline) and its quantized and binarized variants. To quantize the MNIST-L5 model, we use the 8-bit quantization in [33, 99], which converts the model parameters in all layers into integers between 0 and 255. For the binarization, we employ XNOR-Net [111], which converts the model parameters to -1 and 1, except for the first convolutional layer. Using these variants, we evaluate the vulnerability to single bit-flips and report the accuracy,

[9]https://www.tensorflow.org/lite/performance/post_training_quantization

total parameters and vulnerability; without the speed-up heuristics.

Table 3.7 shows the effectiveness of using low-precision parameters. For each network (Column 1), we report the quantization method, the accuracy, the number of vulnerable parameters and their percentage (Columns 2-5). We find that *using low-precision parameters reduces the vulnerability*: in all cases, the ratio of vulnerable parameters drops from 49% (Baseline) to 0-2% (surprisingly 0% with the quantization). We also observe that, in the binarized model, the first convolutional and the last classification layers contain most of the vulnerable parameters; with 150 and 420 vulnerable parameters, respectively. This also corroborates with our results in Sec 3.3.3.

Takeaways. Even if 8-bit quantization mitigates the vulnerability, in a real-world scenario, training a large model, such as [4], from scratch can take a week on a supercomputing cluster. This computational burden lessens the practicality of this defensive mechanism.

3.6 Conclusions

This work exposes the limits of DNN's resilience against parameter perturbations. We study the vulnerability of DNN models to single bit-flips. We evaluate 19 DNN models with six architectures on three image classification tasks and estimate that 40-50% of a DNN's parameters are vulnerable. An attacker, with only a single-bit corruption of these vulnerable parameters, can cause indiscriminate damage [RAD> 0.1]. We further characterize this vulnerability based on the impact of various factors: the bit position, bit-flip direction, parameter sign, layer width, activation function, training techniques, and model architecture. To demonstrate the feasibility of the bit-flip attacks in practice, we leverage a software-induced fault injection attack, Rowhammer. In experiments with Rowhammer, we find that, without knowing the victim's deep learning system, the attacker can inflict indiscriminate damage without system crashes. Lastly, motivated by the attacks, we discuss two potential directions of mitigation: restricting activation magnitudes and using low-precision numbers. We believe that our work is an important step for understanding and mitigating this emerging threat that can compromise the security of critical deep learning systems.

Chapter 4: Jeopardizing Confidentiality via Side-Channel Attacks

In this case, love just lost Germany the whole bloody war.

— The Imitation Game, 2014.

To continue outperforming state-of-the-art results, research in deep learning has shifted from manually engineering features to engineering deep learning systems, including novel data pre-processing pipelines [44, 112] and novel neural architectures [45, 113]. For example, a recent malware detection system MalConv, with a manually designed pipeline that combines embeddings and convolutions, achieves 6% better detection rate over previous state-of-the-art technique without pre-processing [44]. In addition to designing data pre-processing pipelines, other research efforts focus on neural architecture search (NAS)—a method to automatically generate novel architectures that are faster, more accurate and more compact. For instance, the recent work of ProxylessNAS [45] can generate a novel architecture with 10% less error rate and 5x fewer parameters than previous state-of-the-art generic architecture. As a result, in the industry such novel deep learning architectures are kept as trade secrets or intellectual property as they give their owners a competitive edge [95].

These novel deep learning architectures are usually costly to obtain: generating the

NASNet architectures [113] takes almost 40K GPU hours and the MalConv authors had to test a large number of failed designs in the process of finding a successful architecture. As a result, an adversary who wishes to have the benefits of such deep learning architectures without incurring the costs has an incentive to steal them. Compared to stealing a trained model (including all the weights), stealing the *architectural details* that make the victim deep learning system novel provides the benefit that the new architectures and pipelines are usually applicable to multiple tasks. Training new deep learning systems based on these stolen details still provides the benefits, even when the training data is different. After obtaining these details, an attacker can train a functioning model, even on a different data set, and still benefit from the stolen deep learning architectures [112, 114]. Further, against a novel system, stealing its architectural details increases the reliability of black-box poisoning and evasion attacks [115]. Moreover, stealing leads to threats such as Camouflage attacks [116] that trigger misclassifications by exploiting the image scaling algorithms that are common in DNN pre-processing pipelines.

The emerging Machine-Learning-as-a-Service (MLaaS) model that offers deep learning computation tools in the cloud makes remote hardware side-channel attacks a practical vector for stealing deep learning architectures [117]. Unlike prior stealing attacks, these attacks do not require physical proximity to the hardare that runs the system [118, 119] or direct query access to train an approximate model [86]. Cache side-channel attacks have especially been shown as practical in cloud computing for stealing sensitive information, such as cryptographic keys [117]. Cache side-channel attacks are ubiquitous and difficult to defeat as they are inherent to the micro architectural design of modern CPUs [120].

In this chapter, considering the incentives to steal a novel deep learning architectures and applicability of cache side-channel attacks in modern deep learning settings, we design *a practical attack to steal novel deep learning architectures by leveraging only the cache side-channel leakage*. Simulating a common cloud computing scenario, our attacker has a co-located VM on the same host machine as the victim deep learning system, and shares the last-level cache with the victim [117]. As a result, even though the VMs are running on separate processor cores, the attacker can monitor the cache accesses a deep learning framework—PyTorch or TensorFlow—makes while the victim system is running [117].

The first step of our attack is launching a cache side-channel attack, Flush+Reload [121], to extract a single trace of victim's function calls (Section **??**). This trace corresponds to the execution of specific network operations a deep learning framework performs, e.g., convolutions or batch-normalizations, while processing an input sample. However, the trace has little information about the computational graph, e.g., the layers, branches or skip connections, or the architectural parameters, e.g., the number of filters in a convolutional layer. The limited prior work on side-channel attacks against deep learning systems assumed knowledge of the architecture family of the victim DNN [122, 123]; therefore, these attacks are only able to extract variants of generic architectures, such as VGG [92] or ResNet [93]. To overcome this challenge, we also extract the approximate time each deep learning operation takes, in addition to the trace, and we leverage this information to estimate the architectural parameters. This enables us to develop a *reconstruction algorithm* that generates a set of candidate graphs given the trace and eliminates the incompatible candidates given the parameters (Section 4.3). We apply our technique to two exemplar deep learning architectures: the MalConv data pre-processing pipeline and a novel neural architecture produced by ProxylessNAS.

4.1 Related Work

Here, we discuss prior efforts in both crafting and stealing network architectures. There is a growing interest in crafting novel DL systems as they significantly outperform their generic counterparts. The immense effort and computational costs of crafting them, however, motivates the adversaries to steal them.

4.1.1 Effort to Design Deep Learning Systems

Creating deep learning systems traditionally takes the form of human design through expert knowledge and experience. Some problems require novel designs to manipulate the input in a domain-specific way that DNNs can process more effectively. For example, MalConv malware detection system [44] uses a manually designed pre-processing pipeline that can digest raw executable files as a whole. Pseudo LIDAR [112], by pre-processing the output of a simple camera sensor into a LIDAR-like representation, achieves four times better object detection accuracy than previous state-of-the-art technique. Moreover, recent work also focuses on automatically generating optimal architectures via neural architecture search (NAS). For example, reinforcement learning [124] or gradient-based approaches [45] have been proposed for learning to generate optimal architectures. Even though NAS procedures have been shown to produce more accurate, more compact and faster neural networks, the computational cost of the search can be an order of magnitude higher than training a generic architecture [113].

4.1.2 Effort to Steal Deep Learning Systems

Prior work on stealing DNN systems focus on two main threat models based on whether the attacker has physical access to the victim's hardware. Physical access attacks have been proposed against hardware accelerators and they rely on precise timing measurements [125] or electromagnetic emanations [118]. These attacks are not applicable in the cloud setting we consider. The remote attacks that are applicable in the cloud setting, on the other hand, have limitation of requiring precise measurements that are impractical in the cloud [123]. Further, the attack without this limitation [126] requires the attacker to know the family the target architecture comes from; thus, it cannot steal novel architectures. In our work, we design an attack to reconstruct novel DL systems by utilizing a practical cache side-channel attack in the cloud setting.

4.2 Extracting the Sequence of Computations via Flush+Reload

4.2.1 Threat Model

We consider an attacker who aims to steal the key components in a novel DL system, i.e., a novel pre-processing pipeline or a novel network architecture. We first launch a Flush+Reload [121] attack to extract cache side-channel information leaked by DL computation. Our target setting is a cloud environment, where the victim's DL system is deployed inside a VM—or a container—to serve the requests of external users. Flush+Reload, in this setting, is known to be a weak, and practical, side-channel attack [117]. Further, as in MLaaS products in the cloud, the victim uses popular open-source DL frameworks, such as PyTorch [127] or TensorFlow [128].

Capabilities. We consider an attacker that owns a *co-located* VM—or a container—in the same physical host machine as the victim's system. Prior work has shown that spinning-up the co-located VM in the third-party cloud computing services does not require sophisticated techniques [129, 130, 131, 132, 133]. Due to the co-location, the last-level cache (L3 cache) in the physical host is shared between multiple cores where the attacker's and victim's processes are; thus, our attacker can monitor the victim's computations leaked at the L3 cache. We also note that, even if the victim uses GPUs, our attacker can still observe the same computations used for CPUs via cache side-channels (see our discussion in Appendix A.2).

Knowledge. We consider our attacker and the victim use the same version of the same open-source DL framework. This is realistic, in MLaaS scenarios such as AWS SageMaker or Google Cloud's AutoML, as cloud providers recommend practitioners to use the common frameworks to construct their systems[1]. These common practices also allow our attacker to reverse-engineer the frameworks offline and identify the lines of code to monitor with the Flush+Reload technique.

[1]For example, AWS provides convenient deployment options for both PyTorch and TensorFlow: `https://docs.aws.amazon.com/sagemaker/latest/dg/pytorch.html`, and `https://docs.aws.amazon.com/sagemaker/latest/dg/tf.html`.

4.2.2 Flush+Reload Mechanism

Flush+Reload allows an adversary to continually monitor victim's instruction access patterns by observing the time taken to load them from memory. This technique is effective to extract the computation flow of the victim's program when the attacker and victim share memory (i.e., a shared library or page deduplication [81]). The attacker flushes specific lines of code in a shared DL framework from the co-located machine's cache-hierarchy and then measure the amount of time it takes to reload the lines of code. If the victim invokes the monitored line of code, the instruction will be reloaded into the shared cache, and when the attacker reloads the instruction, the access to it will be noticably faster. On the other hand, if the victim does not call the monitored line of code, the access to it will be slower because the instruction needs to be loaded from main memory (DRAM). By repeating this process, our attacker can tell when a victim has accessed a line of code.

4.2.3 Overview of Our Attack Procedure

Online (Co-located)	**Offline (Separate)**
② Monitor the lines of code via Flush+Reload	① Identify the lines of code to monitor ③ De-noise the Flush+Reload observations ④ Profile the computations ⑤ Perform the reconstruction process

Figure 4.1: **Overview of our attack procedure.** Our attacker only requires to be online (co-located) while the attacker is monitoring the computations of a victim DL system.

In Figure 4.1, we illustrate our attack procedure. We split the steps into two phases: the online phase and the offline phase. In the online phase (step ②), the attacker needs co-location to monitor the computations from the victim's system. In the offline phase (steps ①, ③, ④, and ⑤) attacker does not require the co-location with the victim.

① First, our attacker analyzes the open-source DL framework to identify the lines of code to monitor. The attacker monitors the first line of each function that corresponds to the start of a DL computation.

② Next, the attacker spins up a co-located VM and launches the Flush+Reload attack to extract the trace of victim system's function calls. As the trace does not depend on the input sample, we only require to extract a single trace from one full invocation of the victim system.

③ Since the raw observations with Flush+Reload are noisy, the attacker applies filtering to highlight the regularities of DL computations reflected in the trace.

④ To estimate the architectural parameters, e.g., the input/output channels, kernel size, or strides, our attacker creates a lookup tables of timings and performed number of matrix multiplications by collecting traces from various parameter combinations.

⑤ Finally, using the victim's computational trace lookup tables for estimating architectural parameters, the attacker starts reconstruction to steal the victim's DL system (Sec 4.3).

4.2.4 Monitoring the Toy Network Computations via Flush+Reload

Experimental Setup. We implement our attack on Ubuntu 18.04 running on a host machine equipped with the Intel E3-1245v6 3.7GHz processors (8 cores, 32GB memory

ToyNet (PyTorch)

```
class ToyNet(nn.Module):
  def __init__(self, inplace=True):
    super(ToyNet, self).__init__()
      self.c1 = nn.Conv2d(3,10,1,1)
      self.b1 = nn.BatchNorm2d(10)
      self.c2 = nn.Conv2d(10,10,1,10)
      self.b2 = nn.BatchNorm2d(10)
      self.r2 = nn.ReLU6(inplace)

  def forward(self, x):
      x = self.c1(x)
      x = self.b1(x)
      skip = x  # skip connection
      x = self.c2(x)
      x = self.b2(x)
      x = self.r2(x)
      x += skip # skip connection
      return x
```

Raw Trace

```
52230076,Conv2d
...
53440820,GEMM(conv)
53440820,GEMM(oncopy)
...
54242076,BatchNorm2d
...
55600076,Conv2d
56040820,GEMM(conv)
56040820,GEMM(oncopy)
...
59268076,BatchNorm2d
59880076,ReLU6
...
60270076,add
```

Processed Trace

```
[0] Conv2D,52230076,1,2
[1] BatchNorm2D,54230076,0,0
[2] Conv2D,55600076,10,10
[3] BatchNorm2D,59268076,0,0
[4] ReLU6,59880076,0,0
[5] TensorAdd,60270076,0,0
```

Figure 4.2: **Extracted traces while ToyNet is in use.** In the left, we place our network definition in PyTorch. We show the raw trace from Flush+Reload (middle) and the denoised trace (right).

and 8MB cache shared between cores). For the step ①, we analyze two popular open-source DL frameworks, PyTorch and TensorFlow, and identify the list of functions to monitor (see Appendix A.1 for the full list of functions). We leverage the Mastik toolkit [134] to launch the Flush+Reload attack, and while a victim DL system is running on a VM, our attacker monitors the list of functions—step ②. For the reconstruction process conducted in offline after the extraction, we use Python v3.6[2] to implement the procedure.

ToyNet Results. In Figure 4.2, we demonstrate the extracted trace via Flush+Reload while ToyNet is processing an input. ToyNet is composed of one convolution followed by a batch-norm and one depthwise convolution followed by a batch-norm and a ReLU activation. The 1st convolution has the parameters (in, out, kernel, stride) as $(3, 10, 3, 1)$, and the depthwise convolution's parameters are $(10, 10, 1, 1)$. The network has a skip connection that adds the intermediate output (from the 1st convolution) to the final output. During inference, we feed in an input with dimensions 3x32x32.

[2] `https://www.python.org`

In the middle panel of Figure 4.2, we also show the raw—noisy—trace from the Flush+Reload output. The trace only includes cache-hits where the attacker's accesses to the lines of code are faster, i.e., when the victim invokes the function. Each element of the trace includes a timestamp and a function name. The name corresponds to the ToyNet layers, such as Conv2d and BatchNorm2d, and it also contains additional information such as the tensor (add) and the BLAS operations, e.g., GEMM(oncopy).

Our attacker filters the raw trace according to the regular patterns in the DL computation. For example, a long function call, e.g., Conv2d in the ToyNet trace, can appear multiple times in the trace, as the cache can hit multiple times during Flush+Reload. In this case, we condense the multiple occurrences into a single invocation using a heuristic based on how close the timestamps are. We also observe the matrix multiplications such as GEMM(conv) and GEMM(oncopy) while DL computation is being processed. We count the individual occurrences and sum them up them based on the timestamp. After obtaining the processed trace (in the right panel), the attacker starts the reconstruction procedure.

4.3 Reconstructing Novel Deep Learning Systems

After processing the Flush+Reload trace, our attacker reconstructs the key components of the victim's DL system. In this process, the attacker aims to *generate the candidate computational graphs* of the victim system and to *eliminate the incompatible candidates* by estimating the correct parameter set for each computation. For instance, in our ToyNet example, the attacker wants to identify the computational orders and the location of the start and end of a branch connection (computational graph). Also, the same attacker wants to estimate the parameters for each computation; for example, the input/output channels and the kernel size in the 1st Conv2d. In this small network that has one branch, there are only 10 candidate computational graphs; however, considering all possible combinations of parameters, this will result in untractable number of candidates. Prior work, in reconstruction, only considered generic architectures such as VGGs or ResNets with the unrealistic assumption that an attacker knows the architecture family (*backbone*); however, as our aim is to steal novel DL systems, we do not make this assumption. To overcome this problem, we design a reconstruction procedure, which we describe next.

Knowledge of Our Attacker in Reconstruction. Here, we consider our attacker knows what tensor operations and functions to monitor in the victim's open-source DL framework. These functions are model-independent; they correspond to architectural attributes designated by the deep learning framework (see Appendix A.1). We show that this knowledge is sufficient to reconstruct novel data-preprocessing pipelines, such as MalConv, that are usually shallower than the network architectures.

To reconstruct the deeper network architecture automatically designed by NAS algorithms, we assume our attacker has some knowledge about the NAS search space—e.g., NASNet search space [113]—the victim's search process relies on. This knowledge includes the list of layers used and the fact that a set of layers (known as blocks) are repeatedly used such as Normal and Reduction Blocks in NASNet. We make this assumption because, from the sequence of computations observed via Flush+Reload, our attacker can easily identify a set of layers and the repetitions of the layers. However, we do mot assume how each block is composed by using the layer observations directly; instead, we identify candidate blocks by using a sequence mining algorithm. We demonstrate that, under these assumptions, our attack reconstructs the ProxylessNAS-CPU in 12 CPU hours rather than running a NAS algorithm from scratch that takes 40k GPU hours[3].

4.3.1 Overview of Our Reconstruction Procedure.

We first focus on the invariant rules in the computations used for DL computations. For instance, there are unary operations and binary operations. The tensor addition used to implement a skip connection is binary operation; thus, our attacker can supplement the reconstruction process by pruning the incompatible candidates. We also exploit the fact that computation time is proportional to the number of element-wise multiplications in a computation. In the ToyNet example, the time the 1st convolution (2 million cycles) takes is shorter than the 2nd depthwise convolution (3.668 million cycles); thus, our attacker further eliminates the candidates by comparing the possible parameters for a computation with her offline profiling data—the lookup table.

Our reconstruction procedures consist of two steps:

① *Generation*: The attacker can generate the candidate computational graphs from the Flush+Reload trace based on the invariant rules in DL computations. Using the rules, our attacker reduces the number of candidates significantly.

② *Elimination*: Our attacker compares the time for each computation takes with profiling data and prunes the incompatible candidates. We estimate the parameters sequentially

[3]Note that ProxylessNAS starts its searching process from a backbone architecture such as NASNet; thus, even if the paper reported a search took 200 GPU hours, this number does not include the time spent searching a backbone architecture, i.e., the 40k GPU hours to find NASNet.

				Reconstruction Errors	
Victim	**Type**	**# Candidates**	**# Compatibles**	**Topology (GED)**	**Parameters (ℓ_1)**
MalConv	Pre-processing Pipeline	20	**1**	**0**	**0**
ProxylessNAS-CPU	Deep Neural Network	180,224	**1**	**0**	**0**

Table 4.1: **Reconstruction results.** We report the number of candidates and compatible architectures. Also, we report the reconstruction errors in the resulting topology and parameters, and they are 0%.

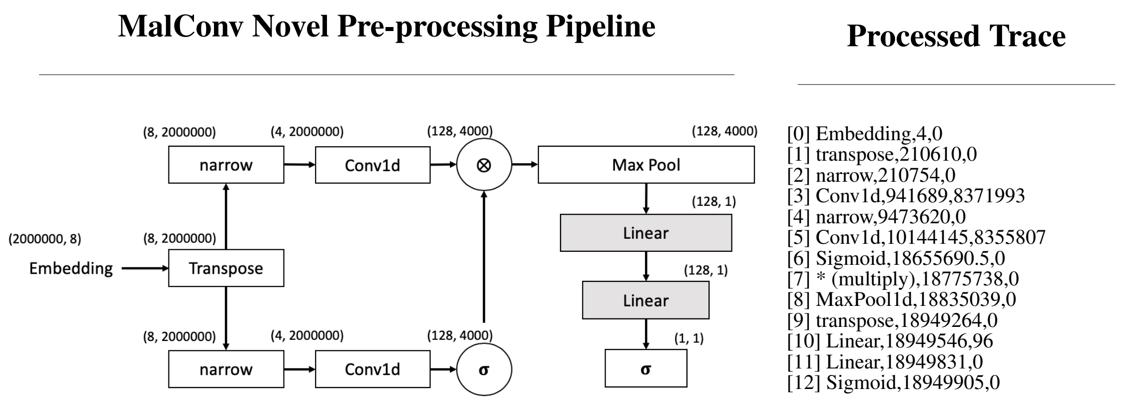
starting from the input. When the output dimension from a candidate does not match with the observation, we eliminate.

Error Metrics. To quantify the error of our reconstruction result, we use two similarity metrics. First, we use the graph edit distance (GED) [135] to compare the reconstructed computational graph with that of the victim. Second, we use the ℓ_1-distance to compute the error between the estimated architectural parameters and those in the victim system.

Victims. We first reconstruct the MalConv [44], a novel data pre-processing pipeline that converts the binary file into a specific format so that a neural network can digest easily. Also, we show that our attacker can reconstruct the novel ProxylessNAS [45] architecture that shows the improved accuracy on the ImageNet classification with the less computational cost on a CPU.

4.3.2 Reconstructing Novel Pre-processing Pipelines

Here, we elaborate the reconstruction process of the MalConv [44] data pre-processing pipeline. MalConv receives the raw bytes of `.exe` files and determines whether the file is malicious or not. The uniqueness of MalConv comes from the way that it treats the sequence of bytes: 1) Code instructions in binary file are correlated spatially, but the correlation has discontinuities from function calls and jump commands that are difficult to capture by the sequence models, e.g., RNNs. 2) Also, each sequence has the order of two million steps which far exceedes the length of an input to any previous neural network classifier. MalConv tackles this problem by pre-processing the sequence of bytes (Figure 4.3). It first splits the upper four bits and the lower four bits (narrow operations) of a byte information; this helps the network capture the locality of closer bytes and distant bytes. Next, the pipeline uses one dimensional convolution to extract such localities and performs the element-wise multiplications of two outputs. Before feeding this information to the neural network, the pipeline uses max-pooling to reduce the training time caused by processing inputs with large dimensions. All these heuristics are examined manually (see Section 4 of the original paper); thus, our attacker can save time and effort by stealing the pipeline.



```
[0] Embedding,4,0
[1] transpose,210610,0
[2] narrow,210754,0
[3] Conv1d,941689,8371993
[4] narrow,9473620,0
[5] Conv1d,10144145,8355807
[6] Sigmoid,18655690.5,0
[7] * (multiply),18775738,0
[8] MaxPool1d,18835039,0
[9] transpose,18949264,0
[10] Linear,18949546,96
[11] Linear,18949831,0
[12] Sigmoid,18949905,0
```

Figure 4.3: **MalConv and the extracted trace.** While the MalConv (left) is processing a sample, we extract a trace via Flush+Reload and process the trace (right) for the pipeline reconstruction.

Generate Computational Graphs. The first step of our attacker is to reconstruct the computational graph candidates for the victim pipeline from the Flush+Reload trace. As we can see in the trace in Figure 4.3, the attacker cannot simply connect the components in the traces sequentially because the branch connection, e.g., `[7]*(multiply)`. Also, from this component, our attacker knows which is the end of a branch but cannot know when the branch has started. We solve this problem by populating all possible candidates and pruning them later with the parameter estimation.

Algorithm 1 Populate computational graphs

```
1:  procedure POPULATEGRAPHS($T$)
2:    $t = T.pop()$
3:    if $t$ is empty then
4:      $g = CreateAGraph(t)$
5:      $l = CandidateList()$
6:      $l = l \cup g$
7:      return $l$
8:    else
9:      if $t$ is unary operator then
10:       $l = PopulateGraphs(T)$
11:       for $g$ in $l$ do
12:         $CreateAnEdge(t, g)$
13:       end for
14:       return $l$
15:     else if $t$ is binary operator then
16:       for each preceding layer $e$ in $T$ do
17:         $Tl, Tr = Split(T, t)$
18:         $ll = PopulateGraphs(Tl)$
19:         $lr = PopulateGraphs(Tr)$
20:         for each $gl, gr$ of $ll \times lr$ do
21:           $g = Compose(el, er)$
22:           $l = l \cup g$
23:         end for
24:       end for
25:       return $l$
26:     end if
27:   end if
28: end procedure
```

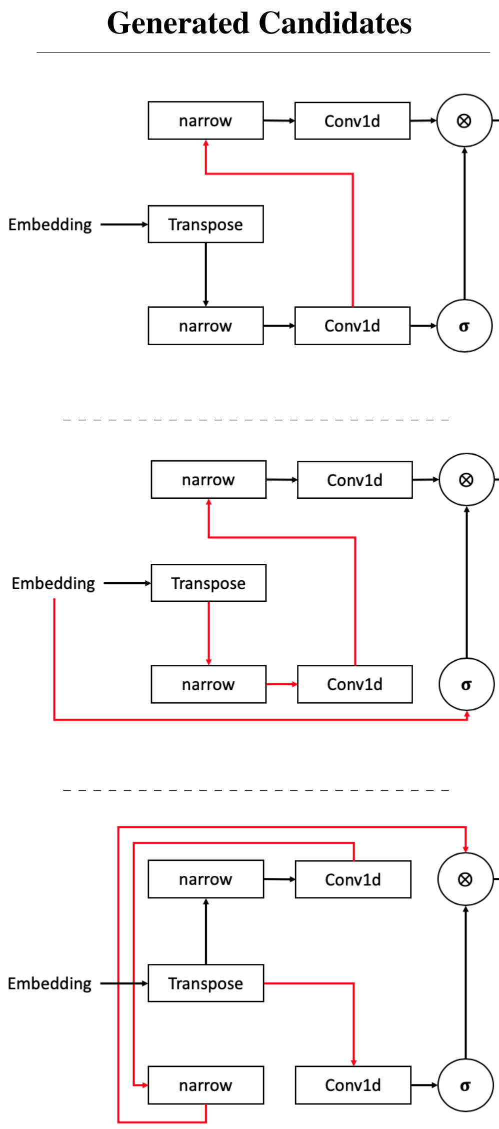


Figure 4.4: **The algorithm for searching candidate computational graphs.** We show the algorithm to populate the candidates of MalConv (left) and the samples (right).

Our algorithm populates the candidate computational graphs (see Figure 4.4). Our solution uses a recursive algorithm. Given a trace from Flush+Reload (T), we pop each computation t from the back and construct the list of candidates l. At a high-level, the algorithm first traverses all the possible connections starting from the last computation to the first by using recursion. Then, when the base condition is met (i.e., the algorithm arrives the first computation, Embeddings), we backtrack the recursions to construct the list of candidate computational graphs. We focus on the computation type in this backtracking process; there are unary and binary computations. For the unary operations, we simply connect the current and preceding computations. However, for the binary operations, we split all the preceding computations into a set of two lists. Each set of two lists corresponds to a branch, and we continue backtracking for each branch and include all of the construction into our results. At the end, *we found 20 candidates*.

Eliminate Candidates with Computational Parameters. Next, our attacker further prunes the candidates based on the computational parameter estimation process. Our attacker, most importantly, focuses on the fact that computation time is dependent on the size of the matrix multiplication. This enables our attacker to profile the computational time taken for a set of parameter combinations in advance. The attacker is able to perform this offline by taking advantage of cloud infrastructure: that the hardware and software stacks composing the cloud are consistent[4]. In the MalConv reconstruction, we profile the timing of the convolution and linear operations. For the convolutions, we consider input/output channels $\{1, 2, 2^2, ..., 2^8\}$, kernels $\{1, 3, 5, 7, 11, 100, 200, 500, 1k, 10k\}$, and strides $\{1, 2, 5, 10, 100, 200, 500, 1k, 10k\}$. For the linear layers, we use input $\{2^2, ..., 2^{11}\}$

[4]`https://aws.amazon.com/ec2/instance-types/`

and output dimensions $\{1, 10, 16, 20, 32, 40, 100, 128, 256, 512, 1k, 1024, 2048\}$.

Once our attacker has the timing profiles with these parameter combinations, the attacker defines the potential parameter sets for the convolutions and linear layers. Then, the attacker checks, in each candidate, if the computational graph returns the correct output dimension $(1,)$ for the input $(8, 2000000)$. In this pruning process, there are the other operations such as Sigmoid, * (multipy), transponse, narrow, or pooling. We applied the universal rules for each case: 1) the Sigmoid and multiply do not change the input/output dimensions, 2) the transpose only swaps two dimensions in an input, 3) the narrow slice one chosen dimension, e.g., (8,2000000) to (4,1000000); thus we consider all the possible slice in checking, and 4) the pooling only requires us to estimate its window size, so we match this value to the stride of a preceding convolution. At the end of this parameter estimation, *we can narrow down to only one architecture with the correct set of computational parameters, i.e., 0% error.*

4.3.3 Reconstructing Novel Network Architectures

Here, we show our attacker is able to steal a novel network architecture by describing the reconstruction process of the ProxylessNAS-CPU [45] that improves the accuracy of existing architecture, MobileNetV2 [136], and also reduces the computation time. Indeed, the NAS search procedure warm-starts from an over-parameterized MobileNetV2 as a backbone; however, in our attack, *we hypothesize our attacker is not aware of the backbone*. Instead, we assume our attacker only knows the search space of MNasNet [137] (see Appendix A.4) where the authors come up with the MobileNetV2, opposed to the recent attacks in Sec 4.1.

Knowing the search space does not, however, reduce the amount of efforts by our attacker in reconstruction. The network architectures found from the NAS procedure commonly are wide and deep, and they include multiple branch connections; thus, our attacker requires to consider exponential number of candidate computational graphs and the computation parameters, which makes the attack infeasible. To tackle this issue, we focus on the NAS procedure—this process factorizes the entire architecture into blocks by their functions. For instance, NASNet [113] is composed of normal cells (blocks) and reduction cells. Within each block, the process considers the architecture combinations that provides the optimal performance. Thus, we first identify the potential blocks before we initiate the process for reconstructing candidate computational graphs.

Block Size	**Counts**	**Identified Block**
1	1	AvgPool2d, Linear
2	38	Conv2d - BatchNorm2d
3	19	Conv2d - BatchNorm2d - ReLU
4	5	Conv2d - BatchNorm2d - Conv2d - BatchNorm2d
5	17	DepthConv2d - BatchNorm2d - ReLU6 - Conv2d - BatchNorm2d
6	10	DepthConv2d - BatchNorm2d - ReLU6 - Conv2d - BatchNorm2d - add
7	0	-
8	15	Conv2d - BatchNorm2d - ReLU6 - DepthConv2d - BatchNorm2d - ReLU6 - Conv2d - BatchNorm2d
9	8	Conv2d - BatchNorm2d - ReLU6 - DepthConv2d - BatchNorm2d - ReLU6 - Conv2d - BatchNorm2d - add

Table 4.2: **The 9 candidate blocks identified from the Flush+Reload trace.**

Identifying Candidate Blocks. We utilize the frequent subsequence mining (FSM) method to identify the blocks composing the ProxylessNAS-CPU architecture. Our FSM method is simple: we iterate over the Flush+Reload trace with the fixed windows and count the occurrences of each subsequence. Since the attacker knows that in the search space that the victim uses, a maximum of nine computations are used to compose a block, we consider the window size from one to nine. Once we count the number of occurrences for each subsequence (candidate blocks), and we prune them based on the rules in the search space: (1) a Conv2d operation is followed by a BatchNorm, (2) a block with a DepthConv2d must end with a Conv2d and BatchNorm (for a depthwise separable convolution), (3) a branch connection cannot merge (add) in the middle of the block, and (4) we take the most frequent block in each window. In Table 4.2, we describe the *9 identified blocks*. We then run the generation process of reconstructing candidate computational graphs with the blocks instead of using each computation in the trace. At the end, *we have 180,224 candidate computational graphs*.

Processed Trace	**Identified Block**	**Estimated Parameters**
...	...	...
[8] Conv2d,305693984,1,20	[B-8] Conv2d,1,20	C_1:24 in, 144 out channels
[9] BatchNorm2d,323455984,0,0	[B-8] BatchNorm2d,0,0	144 in/out channels
[10] ReLU6,376113984,1,1	[B-8] ReLU6,0,0	
[11] DepthConv2d,426259984,143,205	[B-8] DepthConv2d,143,205	144 in/out channels
[12] BatchNorm2d,585059984,0,0	[B-8] BatchNorm2d,0,0	144 in/out channels
[13] ReLU6,592321984,0,0	[B-8] ReLU6,0,0	
[14] Conv2d,609547984,1,7	[B-8] Conv2d,1,7	144 in, C_2:32 out channels
[15] BatchNorm2d,614331984,0,0	[B-8] BatchNorm2d,0,0	C_2:32 in/out channels
...	...	...

Figure 4.5: **The reconstruction of the ProxylessNAS-CPU architecture.** From the Flush+Reload trace (left), we find the candidate block (middle) and estimate the computation parameters (right).

Eliminate Candidate with Computational Parameters. For each candidate composed of known blocks, our attacker estimates the computation parameters. However, the number of parameter combinations are also exponential; for example, within the search space, a Conv2d can have any number of input/output channels, kernel size $\{1, 3, 5\}$, and strides $\{1, 2\}$. Thus, we focus on the computation rules in a block. (1) We first found that DepthConv2d is only possible to have the same in- / out-channels. Also, the channel size can be identified by the number of GEMM(conv) operations. For example, in Figure 4.5, the DepthConv2d has 143 GEMM(conv) invocations, which is close to the channel size. Since commonly the operation has an even number of channels, the attacker can easily reduce the candidates to 142 or 144. (2) We also know that the number of GEMM(oncopy) invocations is proportional to the matrix multiplication size in a Conv2d; thus, the attacker compares the offline profiling results with the processed traces and estimate the parameters. For instance, the 1st Conv2d has 20 GEMM(oncopy), and we approximately have a set of input dimensions, *e.g.*, (20-30,112,112) from the previous block estimation. Thus, our attacker only profiles the variations of input channels $\{20 - 30\}$, kernels $\{1, 3, 5\}$, and strides $\{1, 2\}$—total 60 cases and check if there is a match. Moreover, (3) the Conv2d after DepthConv2d is the pointwise linear operation whose kernel and stride is one, which will further reduce the attacker's efforts. Our attacker runs this elimination process and *finally narrows down to only one architecture with the correct set of computational parameters, i.e., 0% error*.

4.4 Discussion

We discuss defense mechanisms that prevent our attacker from reconstructing the victim's DL system with an exact match. Prior work on defenses against cache side-channel attacks proposed system-level solutions [120, 138, 139]. However, applying them requires infrastructure-wide changes from cloud providers. Also, even if the infrastructure is resilient to cache side-channel attacks, an attacker can leverage other attack vectors to leak similar information. Thus, we focus on the defenses that can be implemented in DL frameworks.

We design our defense mechanisms to obfuscate what the attacker observes via cache side-channels by increasing the noise in computations supported by DL frameworks. We discuss four approaches that blend noise into components of a DL framework; however, these mechanisms introduce a computational overhead by performing additional operations. This highlights that defending against our attack is not trivial and efficient countermeasures require further research.

Padding Zeros to the Matrix Multiplication Operands. Our reconstruction algorithm estimates the computational parameters such as kernel sizes or strides based on the time taken for matrix multiplication. Hence, we consider increasing the size of operands randomly by padding zeros to them. We keep the original sizes of the operands and, after the multiplication of augmented tensors, we convert the resulting tensor into that of the correct dimensions by removing the extra elements. With the augmentation, our attacker finds it difficult to reconstruct the victim's DL system exactly by monitoring a single query. However, if our attacker can observe computations with multiple queries, the attacker can cancel-out the noise and estimate the parameters correctly.

Adding Null/Useless Network Operations. This reconstruction attack assumes all the computations observed in the Flush+Reload trace are used to compute the output of a DL system. Thus, a defender can modify the victim's architecture so that it includes the identity layers or the branches whose outputs are not used. We hypothesize a small number of null/useless operations will not increase the attacker's computational burden significantly; this addition only increases the time needed to reconstruct the victim's architecture by a few hours. If the defender includes an excessive amount of null/useless layers or branches, this can significantly increase the reconstruction time. However, this defense suffers from two issues: 1) the defense may still not make the reconstruction impossible, and 2) the victim also requires to perform the additional operations, which increases network evaluation time significantly.

Shuffling the Computation Order. We have seen in popular DL frameworks that, once a network architecture is defined, the computational order of performing operations is also invariant. We are able to shuffle the computation order of the victim's DL system each time when the system processes an input. In particular, we can identify the dependency of operations in a victim's DL system and compute the independent operations in a different order each time. This approach will make the observations from cache side-channels inconsistent, which results in the exponential number of candidate architectures that our attacker needs to consider. However, to compute the independent operations separately, the defender needs to store intermediate results in memory while processing an input; thus, this approach increases the space overhead of the DL computations.

Running Decoy Operations in Parallel. Lastly, we can make a DL framework run separate networks (decoy operations) in parallel on the same physical host. These networks obfuscate what our attacker will observe via Flush+Reload. Here, the attacker cannot reconstruct the victim architecture by monitoring a single query because the computational order does not reflect how the victim's architecture is defined. However, if our attacker can observe the computations over multiple queries, the attacker can use the frequent sequence mining (FSM) that we used in the block identification to identify a repeated set of operations and can reconstruct the victim architecture. This defense also increases network evaluation time by running extra operations on the same machine.

4.5 Conclusions

This work presents an attack that reconstructs a victim's deep learning system through the information leakage from a cache side-channel, Flush+Reload. We steal key components of the victim's system: a novel pre-processing pipeline and a novel network architecture. Observing the deep learning computations and the time to complete each computation enables the attacker to populate all candidate computational graphs and prune them with our parameter estimation process. In experiments, we demonstrate the feasibility of this reconstruction attack by reconstructing MalConv, a unique pre-processing pipeline for malicious file detection, and ProxylessNAS-CPU, a unique architecture for the ImageNet classification optimized to run on CPUs. We do this with 0% error. As novel deep learning systems become trade secrets, our results highlight the demands for future work on defenses against model theft.

Chapter 5: Reducing Computational Efficiency via Adversarial Examples

Take time for all things: great haste makes great waste.

— Benjamin Franklin.

The inference-time computational demands of deep neural networks (DNNs) are increasing, owing to the "going deeper" [140] strategy for improving accuracy: as a DNN gets deeper, it progressively gains the ability to learn higher-level, complex representations. This strategy has enabled breakthroughs in many tasks, such as image classification [141] or speech recognition [142], at the price of costly inferences. For instance, with $4\times$ more inference cost, a 56-layer ResNet [93] improved the Top-1 accuracy on ImageNet by 19% over the 8-layer AlexNet. This trend continued with the 57-layer state-of-the-art EfficientNet [143]: it improved the accuracy by 10% over ResNet, with $9\times$ costlier inferences.

The accuracy improvements stem from the fact that the deeper networks *fix* the mistakes of the shallow ones [46]. This implies that some samples, which are already correctly classified by shallow networks, do not necessitate the extra complexity. This observation has motivated research on *input-adaptive* mechanisms, in particular, multi-exit architectures [29, 46, 144, 145]. Multi-exit architectures save computation by making

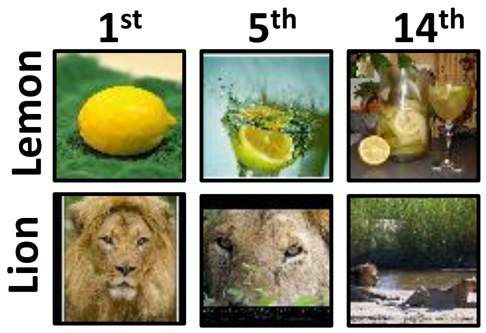


Figure 5.1: **Simple to complex inputs.** Some Tiny ImageNet images a multi-exit model based on VGG-16 can correctly classify if computation stops at the 1^{st}, 5^{th}, and 14^{th} layers.

input-specific decisions about bypassing the remaining layers, once the model becomes confident, and are orthogonal to techniques that achieve savings by permanently modifying the model [32, 33, 146, 147]. Figure 5.1 illustrates how a multi-exit model [29], based on a standard VGG-16 architecture, correctly classifies a selection of test images from 'Tiny ImageNet' before the final layer. We see that more typical samples, which have more supporting examples in the training set, require less depth and, therefore, less computation.

It is unclear whether the computational savings provided by multi-exit architectures are robust against adversarial pressure. Prior research showed that DNNs are vulnerable to a wide range of attacks, which involve imperceptible input perturbations [11, 51, 145, 148]. Considering that a multi-exit model, on the worst-case input, does not provide any computational savings, we ask:

Can the savings from multi-exit models be

maliciously negated by input perturbations?

As some natural inputs do require the full depth of the model, it may be possible to craft adversarial examples that delay the correct decision; it is unclear, however, how many inputs can be delayed with imperceptible perturbations. Furthermore, it is unknown if universal versions of these adversarial examples exist, if the examples transfer across multi-exit architectures and datasets, or if existing defenses (*e.g.* adversarial training) are effective against slowdown attacks.

We consider a new threat against DNNs, analogous to the *denial-of-service (DoS) attacks* that have been plaguing the Internet for decades. By imperceptibly perturbing the input to trigger this worst-case, the adversary aims to slow down the inferences and increase the cost of using the DNN. This is an important threat for many practical applications, which impose strict limits on the responsiveness and resource usage of DNN models (*e.g.* in the Internet-of-Things [147]), because the adversary could push the victim outside these limits. For example, against a commercial image classification system, such as Clarifai.com, a slowdown attack might waste valuable computational resources. Against a model partitioning scheme, such as Big-Little [149], it might introduce network latency by forcing excessive transmissions between local and remote models. A slowdown attack aims to force the victim to do more work than the adversary, *e.g.*. by amplifying the latency needed to process the sample or by crafting reusable perturbations. The adversary may have to achieve this with incomplete information about the multi-exit architecture targeted, the training data used by the victim or the classification task.

To our best knowledge, we conduct the first study of the robustness of multi-exit architectures against adversarial slowdowns. To this end, we find that examples crafted by prior evasion attacks [145, 150] fail to bypass the victim model's early exits, and we show that an adversary can adapt such attacks to the goal of model slowdown by modifying its objective function. We call the resulting attack *DeepSloth*. We also propose an *efficacy* metric for comparing slowdowns across different multi-exit architectures. We experiment with three generic multi-exit DNNs (based on VGG16, ResNet56 and MobileNet) [29] and a specially-designed multi-exit architecture, MSDNets [46], on two popular image classification benchmarks (CIFAR-10 and Tiny ImageNet). We find that DeepSloth reduces the efficacy of multi-exit DNNs by 90–100%, *i.e.*, the perturbations render nearly *all early exits* ineffective. In a scenario typical for IoT deployments, where the model is partitioned between edge devices and the cloud, our attack amplifies the latency by 1.5–5$\times$, negating the benefits of model partitioning. We also show that it is possible to craft a universal DeepSloth perturbation, which can slow down the model on either all or a class of inputs. While more constrained, this attack still reduces the efficacy by 5–45%. Further, we observe that DeepSloth can be effective in some black-box scenarios, where the attacker has limited knowledge about the victim. Finally, we show that a standard defense against adversarial samples—adversarial training—is inadequate against slowdowns. Our results suggest that further research will be required for protecting multi-exit architectures against this emerging security threat.

5.1 Related Work

In this section, we review the prior work's effort to reduce the computational requirement of DNNs and make them available in resource-constrained deployment scenarios.

5.1.1 Efficient Input-Adaptive Inference

Recent input-adaptive DNN architectures have brought two seemingly distant goals closer: achieving both high predictive quality and computational efficiency. There are two types of input-adaptive DNNs: adaptive neural networks (AdNNs) and multi-exit architectures. During the inference, AdNNs [151, 152] dynamically skip a certain part of the model to reduce the number of computations. This mechanism can be used only for ResNet-based architectures as they facilitate skipping within a network. On the other hand, multi-exit architectures [29, 46, 144] introduce multiple side branches—or early-exits—to a model. During the inference on an input sample, these models can preemptively stop the computation altogether once the stopping criteria are met at one of the branches. Kaya et al. [29] have also identified that standard, non-adaptive DNNs are susceptible to *overthinking*, *i.e.*, their inability to stop computation leads to inefficient inferences on many inputs.

[153] presented attacks specifically designed for reducing the energy-efficiency of AdNNs by using adversarial input perturbations. However, our work studies a new threat model that an adversary causes slowdowns on multi-exit architectures. By imperceptibly perturbing the inputs, our attacker can (i) introduce network latency to an infrastructure that utilizes multi-exit architectures and (ii) waste the victim's computational resources. To quantify this vulnerability, we define a new metric to measure the impact of adversarial input perturbation on different multi-exit architectures (Section 5.2). In Section 5.4, we also study practical attack scenarios and the transferability of adversarial input perturbations crafted by our attacker. Moreover, we discuss the potential defense mechanisms against

this vulnerability, by proposing a simple adaptation of adversarial training (Section 5.5). To the best of our knowledge, our work is the first systematic study of this new vulnerability.

5.1.2 Model Partitioning

Model partitioning has been proposed to bring DNNs to resource-constrained devices [147, 149]. These schemes split a multi-exit model into sequential components and deploy them in separate endpoints, e.g., a small, local on-device part and a large, cloud-based part. For bringing DNNs to the Internet of Things (IoT), partitioning is instrumental as it reduces the transmissions between endpoints, a major bottleneck. In Section 5.4.1, on a partitioning scenario, we show that our attack can force excessive transmissions.

5.2 Experimental Setup

Datasets. We use two datasets: CIFAR-10 [88] and Tiny-ImageNet [154]. For testing the cross-domain transferability of our attacks, we use the CIFAR-100 dataset.

Architectures and Hyper-parameters. To demonstrate that the vulnerability to adversarial slowdowns is common among multi-exit architectures, we experiment on two recent techniques: Shallow-Deep Networks (SDNs) [29] and MSDNets [46]. These architectures were designed for different purposes: SDNs are generic and can convert any DNN into a multi-exit model, and MSDNets are custom designed for efficiency. We evaluate an MSDNet architecture (6 exits) and three SDN architectures, based on VGG-16 [92] (14 exits), ResNet-56 [93] (27 exits), and MobileNet [136] (14 exits).

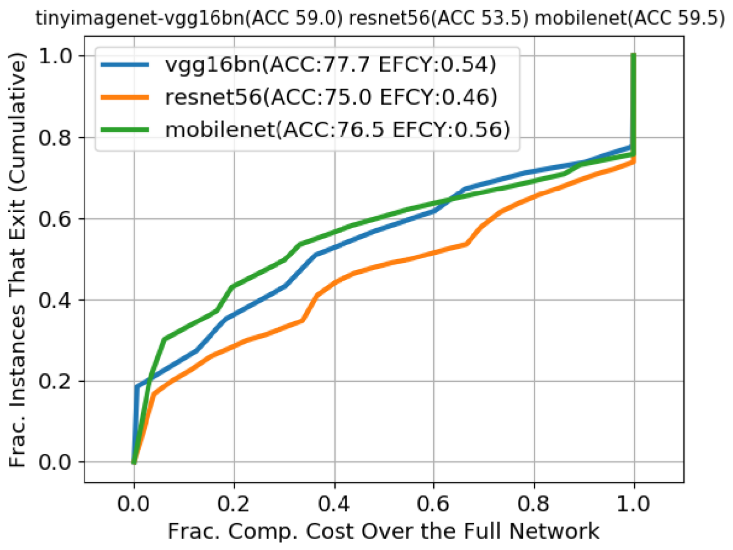


Figure 5.2: **The EEC curves.** Each shows the fraction of test samples a model classifies using a certain fraction of its full inference cost. 'EFCY' is short for the model's efficacy.

Metrics. We define the *early-exit capability* (EEC) curve of a multi-exit model to indicate the fraction of the test samples that exit early at a specific fraction of the model's full inference cost. Figure 5.2 shows the EEC curves of our SDNs on Tiny ImageNet, assuming that the computation stops when there is a correct classification at an exit point. For example, VGG-16-based SDN model can correctly classify $\sim$50% of the samples using $\sim$50% of its full cost. Note that this stopping criterion is impractical; in Section 5.3, we will discuss the practical ones.

We define the early-exit efficacy, or *efficacy* in short, to quantify a model's ability of utilizing its exit points. The efficacy of a multi-exit model is the area under its EEC curve, estimated via the trapezoidal rule. An ideal efficacy for a model is close to 1, when most of the input samples the computation stops very early; models that do not use their early exits have 0 efficacy. A model with low efficacy generally exhibits a higher *latency*; in a partitioned model, the low efficacy will cause more input transmissions to the cloud, and the latency is further *amplified* by the network round trips. A multi-exit model's efficacy and accuracy are dictated by its stopping criteria, which we discuss in the next section. As for the classification performance, we report the Top-1 accuracy on the test data.

5.3 Attacking the Multi-Exit Architectures

Setting. We consider the supervised classification setting with standard feedforward DNN architectures. A DNN model consists of N blocks, or layers, that process the input sample, $x \in \mathbb{R}^d$, from beginning to end and produce a classification. A classification, $F(x, \theta) \in \mathbb{R}^m$, is the predicted probability distribution of x belonging to each label $y \in M = \{1, ..., m\}$. Here, θ denotes the tunable parameters, or the weights, of the model. The parameters are learned on a training set $\mathcal{D}$ that contains multiple (x_i, y_i) pairs; where y_i is the ground-truth label of the training sample x_i. We use θ_i to denote the parameters at and before the i^{th} block; *i.e.*, $\theta_i \subset \theta_{i+1}$ and $\theta_N = \theta$. Once a model is trained, its performance is then tested on a set of unseen samples, $\mathcal{S}$.

Multi-Exit Architectures. A multi-exit model contains K *exit points*—internal classifiers—attached to a model's hidden blocks. We use F_i to denote the i^{th} exit point, which is attached to the j^{th} block. Using the output of the j^{th} ($j < N$) block on x, F_i produces an internal classification, i.e., $F_i(x, \theta_j)$, which we simply denote as $F_i(x)$. In our experiments, we set $K = N$ for SDNs, *i.e.*, one internal classifier at each block and $K = 6$ for MSDNets. Given $F_i(x)$, a multi-exit model uses deterministic criteria to decide between forwarding x to compute $F_{i+1}(x)$ and stopping for taking the *early-exit* at this block. Bypassing early-exits decreases a network's efficacy as each additional block increases the inference cost. Note that multi-exit models process each sample individually, not in batches.

Practical Stopping Criteria. Ideally, a multi-exit model stops when it reaches a correct classification at an exit point, *i.e.*, $\operatorname{argmax}_{j \in M} F_i^{(j)}(x) = \hat{y}_i = y$; y is the ground-truth label. However, for unseen samples, this is impractical as y is unknown. The prior work has proposed two simple strategies to judge whether $\hat{y}_i = y$: $F_i(x)$'s entropy [46, 144] or its confidence [29]. Our attack (see Section 5.3) leverages the fact that a uniform $F_i(x)$ has both the highest entropy and the lowest confidence. For generality, we experiment with both confidence-based—SDNs—and entropy-based—MSDNets—strategies.

A strategy selects confidence, or entropy, thresholds, T_i, that determine whether the model should take the i^{th} exit for an input sample. Conservative T_i's lead to fewer early exits and the opposite hurts the accuracy as the estimate of whether $\hat{y}_i = y$ becomes unreliable. As utility is a major practical concern, we set T_i's for balancing between efficiency and accuracy. On a holdout set, we set the thresholds to maximize a model's efficacy while keeping its relative accuracy drop (RAD) over its maximum accuracy within 5% and 15%. We refer to these two settings as RAD<5% and RAD<15%. Table 5.2 (*first* segment) shows how accuracy and efficacy change in each setting.

5.3.1 Threat Model

We consider an adversary who aims to decrease the early-exit efficacy of a *victim* model. The attacker crafts an *imperceptible* adversarial perturbation, $v \in \mathbb{R}^d$ that, when added to a test-time sample $x \in \mathcal{S}$, prevents the model from taking early-exits.

Capabilities. The attacker is able to modify the victim's test-time samples to apply the perturbations, *e.g.*, by compromising a camera that collects the data for inference. To ensure the imperceptibility, we focus on ℓ_∞ norm bounded perturbations as they (i) are well are studied; (ii) have successful defenses [54]; (iii) have prior extension to multi-exit models [145]; and (iv) are usually the most efficient to craft. We show results on ℓ_2 and ℓ_1 perturbations in Appendix A.7. In line with the prior work, we bound the perturbations as follows: for CIFAR-10, $||v||_\infty \leqslant \epsilon = 0.03$ [150], $||v||_1 \leqslant 8$ [155] and $||v||_2 \leqslant 0.35$ [156]; for Tiny ImageNet, $||v||_\infty \leqslant \epsilon = 0.03$ [65], $||v||_1 \leqslant 16$ and $||v||_2 \leqslant 0.6$.

Knowledge. To assess the security vulnerability of multi-exit architectures, we study *white-box scenarios*, *i.e.*, the attacker knows all the details of the victim model, including its $\mathcal{D}$ and θ. Further, in Section 5.4.2, we study more practical *black-box scenarios*, *i.e.*, the attacker crafts v on a *surrogate* model and applies it to an unknown victim model.

Goals. We consider three DeepSloth variants, (i) the *standard*, (ii) the *universal* and (iii) the *class-universal*. The adversary, in (i) crafts a different v for each $x \in \mathcal{S}$; in (ii) crafts a single v for all $x \in \mathcal{S}$; in (iii) crafts a single v for a target class $i \in M$. Further, although the adversary does not explicitly target it; we observe that DeepSloth usually hurts the accuracy. By modifying the objective function we describe in Section 5.3, we also experiment with DeepSloth variants that can explicitly preserve or hurt the accuracy, in addition to causing slowdowns.

5.3.2 Motivating Examples

We discuss two exemplary scenarios where an adversary can exploit the slowdown attacks.

- **(Case 1) Attacks on cloud-based IoT applications.** In most cases, cloud-based IoT applications, such as Apple Siri, Google Now, or Microsoft Cortana, run their DNN inferences in the cloud. This cloud-only approach puts all the computational burden on cloud servers and increases the communications between the servers and IoT devices. In consequence, recent work [157, 158, 159] utilizes multi-exit architectures for bringing computationally expensive models, *e.g.* language models [160, 161], in the cloud to IoT (or mobile) devices. They split a multi-exit model into two partitions and deploy each of them to a server and IoT devices, respectively. Under this scheme, the cloud server only takes care of complex inputs that the shallow partition cannot correctly classify at the edge. As a result, one can reduce the computations in the cloud and decrease communications between the cloud and edge.

  On the other hand, our adversary, by applying human-imperceptible perturbations, can convert simple inputs into complex inputs. These adversarial inputs will bypass early-exits and, as a result, reduce (or even offset) the computational and communication savings provided by prior work.

  Here, a defender may deploy DoS defenses such as firewalls or rate-limiting. In this setting, the attacker may not cause DoS because defenses keep the communications between the server and IoT devices under a certain-level. Nevertheless, the attacker still increases: (i) the computations at the edge (by making inputs skip early-exits) and (ii) the number of samples that cloud servers process. Recall that a VGG-16 SDN model classifies 90% of clean CIFAR-10 instances correctly at the first exit. If the adversarial examples crafted by the attacker bypass only the first exit, one can easily increase the computations on IoT devices and make them send requests to the cloud.

- **(Case 2) Attacks on real-time DNN inference for resource- and time-constrained scenarios.** Recent work on the real-time systems [162, 163] harnesses multi-exit architectures and model partitioning as a solution to optimize real-time DNN inference for resource- and time-constrained scenarios. [162] showed a real-world prototype of an optimal model partitioning, which is based on a self-driving car video dataset, can improve latency and throughput of partitioned models on the cloud and edge by 6.5–14×.

  However, the prior work does not consider the danger of slowdown attacks; our threat model has not been discussed before in the literature. Our results in Sec 5.4 suggest that slowdown can be induced adversarially, potentially violating real-time guarantees. For example, our attacker can force partitioned models on the cloud and edge to use maximal computations for inference. Further, the same adversarial examples also require the inference results from the model running on the cloud, which potentially increases the response time of the edge devices by 1.5–5×. Our work shows that multi-exit architectures should be used with caution in real-time systems.

5.3.3 Standard Adversarial Attacks Do Not Cause Delays

NETWORK	NO ATTACK	PGD-20	PGD-20 (AVG.)	PGD-20 (MAX.)	UAP
VGG-16	0.77 / 89%	0.79 / 29%	0.85 / 10%	0.81 / 27%	0.71 / 68%
RESNET-56	0.52 / 87%	0.55 / 12%	0.82 / 1%	0.70 / 6%	0.55 / 44%
MOBILENET	0.83 / 87%	0.85 / 14%	0.93 / 3%	0.89 / 12%	0.77 / 60%

Table 5.1: **Impact of existing evasion attacks on efficacy.** Each entry shows a model's efficacy (*left*) and accuracy (*right*) when subjected to the respective attack. The multi-exit models are trained on CIFAR-10 and use RAD<5% as their early-exit strategy.

To motivate DeepSloth, we evaluate whether previous adversarial attacks have any effect on the efficacy of multi-exit models. These attacks add human-imperceptible perturbations to a victim's test-time samples to force misclassifications. We experiment with the standard PGD attack [150]; PGD-avg and PGD-max variants against multi-exit models [145] and the Universal Adversarial Perturbation (UAP) attack that crafts a single perturbation for all test samples [164]. Table 5.1 summarizes our findings that these attacks, although they hurt the accuracy, fail to cause any meaningful decrease in efficacy. In many cases, we observe that the attacks actually increase the efficacy. These experiments help us to identify the critical elements of the objective function of a slowdown attack.

5.3.4 The DeepSloth Attack

The Layer-Wise Objective Function. Figure 5.3 shows that the attacks that only optimize for the final output, *e.g.*, PGD or UAP, do not perturb the model's earlier layer representations. This does not bypass the early-exits, which makes these attacks ineffective for decreasing the efficacy. Therefore, we modify the objective functions of adversarial example-crafting algorithms to incorporate the outputs of all $F_i|i < K$. For crafting ℓ_∞, ℓ_2 and ℓ_1-bounded perturbations, we adapt the PGD [150], the DDN [165] and the SLIDE algorithms [155], respectively. Next, we describe how we modify the PGD algorithm—we modify the others similarly:

$$v^{t+1} = \Pi_{||v||_\infty < \epsilon} \left(v^t + \alpha \text{ sgn} \left(\nabla_v \sum_{x \in D'} \sum_{0<i<K} \mathcal{L} \left(F_i \left(x + v \right), \bar{y} \right) \right) \right)$$

Here, t is the current attack iteration; α is the step size; Π is the projection operator that enforces $||v||_\infty < \epsilon$ and $\mathcal{L}$ is the cross-entropy loss function. The selection of $\mathcal{D}'$ determines the type of the attack. For the standard variant: $\mathcal{D}' = \{x\}$, *i.e.*, a single test-time sample. For the universal variant: $\mathcal{D}' = \mathcal{D}$, *i.e.*, the whole training set. For the class-universal variant against the target class $i \in M$: $\mathcal{D}' = \{(x, y) \in \mathcal{D} | y = i\}$, *i.e.*, the training set samples from the i^{th} class. Finally, $\bar{y}$ is the target label distribution our objective pushes $F_i(x)$ towards. Next, we explain how we select $\bar{y}$.

Pushing $F_i(x)$ Towards a Uniform Distribution. Despite including all F_i, attacks such as PGD-avg and PGD-max [145] still fail to decrease efficacy. How these attacks select $\bar{y}$ reflects their goal of causing misclassifications and, therefore, they trigger errors in early-exits, *i.e.*, $\text{argmax}_{j \in M} F_i^{(j)}(x) = \bar{y} \neq y$. However, as the early-exits still have high confidence, or low entropy, the model still stops its computation early. We select $\bar{y}$ as a *uniform distribution* over the class labels, *i.e.*, $\bar{y}^{(i)} = 1/m$. This ensures that $(x + v)$ bypasses common stopping criteria as a uniform $F_i(x)$ has both the lowest confidence and the highest entropy.

5.4 Empirical Evaluation

Here, we present the results for ℓ_∞ DeepSloth against two SDNs—VGG-16 and MobileNet-based—and against the MSDNets. In the Appendix, we report the hyperparameters; the ℓ_1 and ℓ_2 attacks; the results on ResNet-56-based SDNs; the cost of the attacks; and some perturbed samples. Overall, we observe that ℓ_∞-bounded perturbations are more effective for slowdowns. The optimization challenges might explain this, as ℓ_1 and ℓ_2 attacks are usually harder to optimize [53, 155]. Unlike objectives for mispredictions, the objective for slowdowns involves multiple loss terms and optimizes over all the output logits.

5.4.1 White-Box Scenarios

NETWORK	MSDNET		VGG16		MOBILENET	
SET.	RAD<5%	RAD<15%	RAD<5%	RAD<15%	RAD<5%	RAD<15%
BASELINE (NO ATTACK)						
C10	0.89 / 85%	0.89 / 85%	0.77 / 88%	0.89 / 79%	0.83 / 87%	0.92 / 79%
TI	0.64 / 55%	0.83 / 50%	0.39 / 57%	0.51 / 52%	0.42 / 57%	0.59 / 51%
DEEPSLOTH						
C10	0.06 / 17%	0.06 / 17%	0.01 / 13%	0.04 / 16%	0.01 / 12%	0.06 / 16%
TI	0.06 / 7%	0.06 / 7%	0.00 / 2%	0.01 / 2%	0.02 / 6%	0.04 / 6%
UNIVERSAL DEEPSLOTH						
C10	0.85 / 65%	0.85 / 65%	0.62 / 65%	0.86 / 60%	0.73 / 61%	0.90 / 59%
TI	0.58 / 46%	0.81 / 41%	0.31 / 47%	0.44 / 44%	0.33 / 47%	0.51 / 43%
CLASS-UNIVERSAL DEEPSLOTH						
C10	0.82 / 32%	0.82 / 32%	0.47 / 35%	0.78 / 33%	0.60 / 30%	0.85 / 27%
TI	0.41 / 21%	0.71 / 17%	0.20 / 28%	0.33 / 27%	0.21 / 27%	0.38 / 25%

Table 5.2: **The effectiveness of ℓ_∞ DeepSloth.** 'RAD<5,15%' columns list the results in each early-exit setting. Each entry includes the model's efficacy (*left*) and accuracy (*right*). The class-universal attack's results are an average of 10 classes. 'TI': Tiny ImageNet and 'C10': CIFAR-10.

Perturbations Eliminate Early-Exits. Table 5.2 (*second* segment) shows that the victim models have ~ 0 efficacy on the samples perturbed by DeepSloth. Across the board, the attack makes the early-exit completely ineffective and force the victim models to forward all input samples till the end. Further, DeepSloth also drops the victim's accuracy by 75–99%, comparable to the PGD attack. These results give an answer to our main research question: *the multi-exit mechanisms are vulnerable and their benefits can be maliciously offset by adversarial input perturbations*. In particular, as SDN modification mitigates overthinking in standard, non-adaptive DNNs [29], DeepSloth also leads SDN-based models to overthink on almost all samples by forcing extra computations.

Note that crafting a single perturbation requires multiple back-propagations through the model and more floating points operations (*FLOPs*) than the forward pass. The high cost of crafting, relative to the computational damage to the victim, might make this vulnerability unattractive for the adversary. In the next sections, we highlight scenarios where this vulnerability might lead to practical exploitation. First, we show that in an IoT-like scenarios, the input transmission is a major bottleneck and DeepSloth can exploit it. Second, we evaluate universal DeepSloth attacks that enable the adversary to craft the perturbation only once and reuse it on multiple inputs.

Attacking an IoT Scenario. Many IoT scenarios, *e.g.*, health monitoring for elderly [166], require collecting data from edge devices and making low-latency inferences on this data. However, complex deep learning models are impractical for low-power edge devices, such as an Arduino, that are common in the IoT scenarios [167]. For example, on standard hardware, an average inference takes MSDNet model on Tiny ImageNet 35M FLOPs and $\sim$10ms.

A potential solution is sending the inputs from the edge to a cloud model, which then returns the prediction. Even in our optimistic estimate with a nearby AWS EC2 instance, this back-and-forth introduces $\sim$11ms latency per inference. Model partitioning alleviates this bottleneck by splitting a multi-exit model into two; deploying the small first part at the edge and the large second part at the cloud [149]. The edge part sends an input only when its prediction does not meet the stopping criteria. For example, the first early-exit of MSDNets sends only 5% and 67% of all test samples, on CIFAR-10 and Tiny ImageNet, respectively. This leads to a lower average latency per inference, *i.e.*, from 11ms down to 0.5ms and 7.4ms, respectively.

The adversary we study uses DeepSloth perturbations to force the edge part to send all the input samples to the cloud. For the victim, we deploy MSDNet models that we split into two parts at their first exit point. Targeting the first part with DeepSloth forces it to send 96% and 99.97% of all test samples to the second part. This increases average inference latency to $\sim$11ms and invalidates the benefits of model partitioning. In this scenario, perturbing each sample takes $\sim$2ms on a Tesla V-100 GPU, *i.e.*, the time adversary spends is amplified by 1.5-5$\times$ as the victim's latency increase.

Reusable Universal Perturbations. The universal attacks, although limited, are a practical as the adversary can reuse the same perturbation indefinitely to cause minor slowdowns. Table 5.2 (*third* segment) shows that they decrease the efficacy by 3–21% and the accuracy by 15–25%, over the baselines. Having a less conservative early-exit strategy, *e.g.*, RAD<15%, increases the resilience to the attack at the cost of accuracy. Further, MSDNets are fairly resilient with only 3–9% efficacy drop; whereas SDNs are more vulnerable with 12–21% drop. The attack is also slightly more effective on the more complex task, Tiny ImageNet, as the early-exits become easier to bypass. Using random noise as a baseline, i.e., $v \sim U^d(-\epsilon, \epsilon)$, we find that at most it decreases the efficacy by $\sim$3%.

In the universal attack, we observe a phenomenon: it pushes the samples towards a small subset of all classes. For example, $\sim$17% of the perturbed samples are classified into the 'bird' class of CIFAR-10; up from $\sim$10% for the clean samples. Considering certain classes are distant in the feature space, *e.g.*, 'truck' and 'bird'; we expect the class-universal variant to be more effective. The results in Table 5.2 (*fourth* segment) confirm our intuition. We see that this attack decreases the baseline efficacy by 8–50% and the accuracy by 50–65%. We report the average results across multiple classes; however, we observe that certain classes are slightly more vulnerable to this attack.

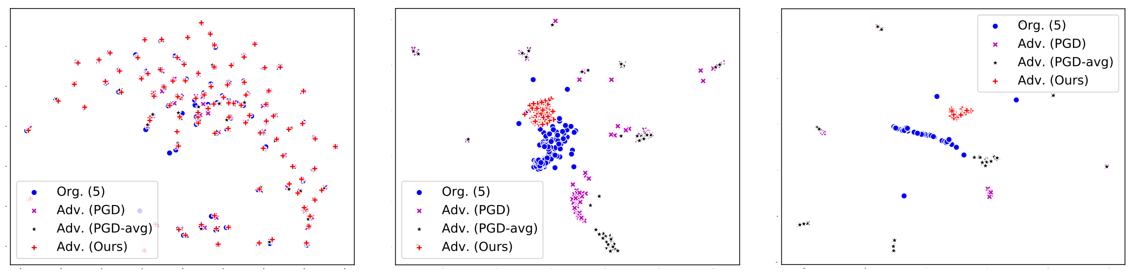


Figure 5.3: **Visualising features against attacks using UMAP.** VGG-16's 3rd (*left*), 8th (*middle*), and 14th (*right*) hidden block features on CIFAR-10's 'dog' class (Best viewed in color, zoomed in).

Feature Visualization of DeepSloth. In Figure 5.3, to shed light on how DeepSloth differs from prior attacks, *e.g.*, PGD and PGD-avg, we visualize a model's hidden block (layer) features on the original and perturbed test-time samples. We observe that in an earlier block (*left* panel), DeepSloth seems to disrupt the original features slightly more than the PGD attacks. Leaving earlier representations intact prevents PGDs from bypassing the early-exits. The behaviors of the attacks diverge in the middle blocks (*middle* panel). Here, DeepSloth features remain closer to the original features than prior attacks. The significant disruption of prior attacks leads to high-confidence misclassifications and fails to bypass early-exits. In the later block (*right* panel), we see that the divergent behavior persists.

Preserving or Hurting the Accuracy with DeepSloth. We answer whether DeepSloth can be applied when the adversary explicitly aims to cause or prevent misclassifications, while still causing slowdowns. Our main threat model has no explicit goal regarding misclassifications that hurt the user of the model, *i.e.*, who consumes the output of the model. Whereas, slowdowns additionally hurt the executor or the owner of the model through the computations and latency increased at the cloud providers. In some ML-in-the-cloud scenarios, where these two are different actors, the adversary might aim to target only the executor or both the executor and the user. To this end, we modify our objective function to push $F_i(x)$ towards a slightly non-uniform distribution, favoring either the ground truth label for preventing misclassifications or a wrong label for causing them. We test this idea on our VGG-16-based SDN model on CIFAR-10 in RAD<5% setting. We see that DeepSloth for preserving the accuracy leads to 81% accuracy with 0.02 efficacy and DeepSloth for hurting the accuracy leads to 4% accuracy with 0.01 efficacy—the original DeepSloth led to 13% accuracy with 0.01 efficacy. These results show the flexibility of DeepSloth and how it could be modified depending on the attacker's goals.

5.4.2 Towards a Black-Box Attack: Transferability of DeepSloth

Transferability of adversarial examples imply that they can still hurt a model that they were not crafted on [13, 56]. Even though white-box attacks are important to expose the vulnerability, black-box attacks, by requiring fewer assumptions, are more practical. Here, on four distinct scenarios, we investigate whether DeepSloth is transferable. Based on the scenario's constraints, we (1) train a *surrogate* model; (2) craft the DeepSloth samples on it; and (3) use these samples on the victim. We run these experiments on

CIFAR-10 in the RAD<5%.

Cross-Architecture. First, we relax the assumption that the attacker knows the victim architecture. We evaluate the transferability between a VGG-16-based SDN and an MSDNet. They are all trained using the same $\mathcal{D}$. We find that the samples crafted on the MSDNet can slowdown the SDN: reducing its efficacy to 0.63 (from 0.77) and accuracy to 78% (from 88%). Interestingly, the opposite seems not to be the case: on the samples crafted against the SDN, the MSDNet still has 0.87 efficacy (from 0.89) and 73% accuracy (from 85%). This hints that DeepSloth transfers if the adversary uses an effective multi-exit models as the surrogate.

Limited Training Set Knowledge. Second, we relax the assumption that the attacker knows the victim's training set, $\mathcal{D}$. Here, the attacker only knows a random portion of $\mathcal{D}$, i.e., 10%, 25%, and 50%. We use VGG-16 architecture for both the surrogate and victim models. In the 10%, 25% and 50% settings, respectively, the attacks reduce the victim's efficacy to 0.66, 0.5, 0.45 and 0.43 (from 0.77); its accuracy to 81%, 73%, 72% and 74% (from 88%). Overall, the more limited the adversary's $\mathcal{D}$ is, the less generalization ability the surrogate has and the less transferable the attacks are.

Cross-Domain. Third, we relax the assumption that the attacker exactly knows the victim's task. Here, the attacker uses $\mathcal{D}_f$ to train the surrogate, different from the victim's $\mathcal{D}$ altogether. We use a VGG-16 on CIFAR-100 as the surrogate and attack a VGG-16-based victim model on CIFAR-10. This transfer attack reduces the victim's efficacy to 0.63 (from 0.77) and its accuracy to 83% (from 88%). We see that the cross-domain attack might be more effective than the limited $\mathcal{D}$ scenarios. This makes DeepSloth particularly dangerous as the attacker, without knowing the victim's $\mathcal{D}$, can collect a similar dataset and still slowdown the victim. We hypothesize the transferability of earlier layer features in CNNs [168] enables the perturbations attack to transfer from one domain to another, as long as they are similar enough.

Cross-Mechnanism. Finally, we test the scenario where the victim uses a completely different mechanism than a multi-exit architecture to implement input adaptiveness, *i.e.*, SkipNet [151]. A SkipNet, a modified residual network, selectively skips convolutional blocks based on the activations of the previous layer and, therefore, does not include any internal classifiers. We use a pre-trained SkipNet on CIFAR-10 that reduces the average computation for each input sample by $\sim$50% over an equivalent ResNet and achieves $\sim$94% accuracy. We then feed DeepSloth samples crafted on a MSDNet to this SkipNet, which reduces its average computational saving to $\sim$32% (36% less effective) and its accuracy to 37%. This result suggests that the two different mechanisms have more in common than previously known and might share the vulnerability. We believe that understanding the underlying mechanisms through which adaptive models save computation is an important research question for future work.

5.5 Standard Adversarial Training Is Not a Countermeasure

In this section, we examine whether a defender can adapt a standard countermeasure against adversarial perturbations, adversarial training (AT) [54], to mitigate our attack. AT decreases a model's sensitivity to perturbations that significantly change the model's outputs. While this scheme is effective against adversarial examples that aim to trigger misclassifications; it is unclear whether using our DeepSloth samples for AT can also robustify a multi-exit model against slowdown attacks.

To evaluate, we train our multi-exit models as follows. We first take a *base* network—VGG-16—and train it on CIFAR-10 on PGD-10 adversarial examples. We then convert the resulting model into a multi-exit architecture, using the modification from [29]. During this conversion, we adversarially train individual exit points using PGD-10, PGD-10 (avg.), PGD-10 (max.), and DeepSloth; similar to [145]. Finally, we measure the efficacy and accuracy of the trained models against PGD-20, PGD-20 (avg.), PGD-20 (max.), and DeepSloth, on CIFAR-10's test-set.

ADV. TRAINING	NO ATTACK	PGD-20	PGD-20 (AVG.)	PGD-20 (MAX.)	DEEPSLOTH
UNDEFENDED	0.77 / 89%	0.79 / 29%	0.85 / 10%	0.81 / 27%	**0.01** / 13%
PGD-10	0.61 / 72%	0.55 / 38%	0.64 / 23%	0.58 / 29%	**0.33** / 70%
PGD-10 (AVG.)	0.53 / 72%	0.47 / 36%	0.47 / 35%	0.47 / 35%	**0.32** / 70%
PGD-10 (MAX.)	0.57 / 72%	0.51 / 37%	0.54 / 30%	0.52 / 34%	**0.32** / 70%
OURS	0.74 / 72%	0.71 / 38%	0.82 / 14%	0.77 / 21%	**0.44** / 67%
OURS + PGD-10	0.61 / 73%	0.55 / 38%	0.63 / 23%	0.58 / 28%	**0.33** / 70%

Table 5.3: **Evaluating adversarial training against slowdown attacks.** Each entry includes the model's efficacy score (*left*) and accuracy (*right*). Results are on CIFAR-10, in the RAD<5% setting.

Our results in Table 5.3 verify that AT provides resilience against all PGD attacks. Besides, AT provides some resilience to our attack: DeepSloth reduces the efficacy to ∼0.32 on robust models vs. 0.01 on the undefended one. However, we identify a trade-off between the robustness and efficiency of multi-exits. Compared to the undefended model, on clean samples, we see that robust models have lower efficacy—0.77 vs. 0.53∼0.61. We observe that the model trained only with our DeepSloth samples (Ours) can recover the efficacy on both the clean and our DeepSloth samples, but this model loses its robustness against PGD attacks. Moreover, when we train a model on both our DeepSloth samples and PGD-10 (Ours + PGD-10), the trained model suffers from low efficacy. Our results imply that a defender may require an out-of-the-box defense, such as flagging the users whose queries bypass the early-exits more often than clean samples for which the multi-exit network was calibrated.

5.6 Conclusions

This work exposes the vulnerability of input-adaptive inference mechanisms against adversarial slowdowns. As a vehicle for exploring this vulnerability systematically, we propose DeepSloth, an attack that introduces imperceptible adversarial perturbations to test-time inputs for offsetting the computational benefits of multi-exit inference mechanisms. We show that a white-box attack, which perturbs each sample individually, eliminates any computational savings these mechanisms provide. We also show that it is possible to craft universal slowdown perturbations, which can be reused, and transferable samples, in a black-box setting. Moreover, adversarial training, a standard countermeasure for adversarial perturbations, is not effective against DeepSloth. Our analysis suggests that slowdown attacks are a realistic, yet under-appreciated, threat against adaptive models.

Chapter 6: Conclusion

This dissertation tackles the conventional perspective on studying the security of deep learning systems and takes a step towards building secure and reliable deep learning systems. To this end, this dissertation proposes employing a systems security perspective. In contrast to the prior work that considers DNNs as a mathematical, isolated concept, we view them as a computational tool running on computing platforms. This unique perspective makes us realize: (1) DNNs can become vulnerable to adversaries who exploit the vulnerability in system components that they rely on, and (2) they have computational properties that make DNNs particularly vulnerable to such attackers. We are therefore can study new attack vectors or objectives under-examined in the prior work.

My dissertation focuses on the three desired system properties that *any* computer systems should satisfy: graceful degradation, confidentiality, and computational efficiency. In particular, I study attack vectors, such as, hardware fault attacks, side-channels, and adversarial input perturbations, that an adversary can use for jeopardizing the three desired properties of deep learning systems. I then characterize the vulnerability of DNNs to those attacks and demonstrate scenarios where an adversary exploits them in practice. From the research projects that I conducted, I reached the following conclusion:

DNNs have computational properties that traditional software does not have, *which makes them particularly vulnerable to adversaries. An adversary, owing to their computational properties, can facilitate existing attack vectors to cause unexpected damage to deep learning systems. Moreover, some of the attacks an adversary exploits have been considered ineffective against traditional software.*

6.1 Summary

Here, we re-articulate the contributions that this dissertation makes:

In Chapter 3: This dissertation exposes the *graceless degradation* of DNNs under bitwise corruptions in their memory representations. Our study examines 19 DNNs, trained on three popular benchmarks for object recognition, showing that a single bit-flip can inflict the accuracy drop of a DNN up to 99% when the bit-flip is at a specific location. We also find that $\sim$50% of a DNN's parameters contain at least one bit that degrades its accuracy over 10%—the empirical upper bound found by prior work. Furthermore, the analysis of DNNs with different datasets, architecture configurations, and training procedures allows us to conclude that this vulnerability is *prevalent*. To demonstrate the practicality of exploiting this vulnerability, we examine an adversary who facilitates a prominent hardware fault attack, Rowhammer. A Rowhammer attacker who can control the bit-flip location can inflict the accuracy drop of a DNN running inside a virtual machine (VM) from 0–99%. Moreover, the *blind* attacker who randomly flips bits in memory can also cause an accuracy drop of over 10% within a minute without any system crashes.

In Chapter 4: We show that an adversary can *steal* a unique DNN architecture from a small amount of information leakage observed from side-channel attacks. We identify the computational regularities that a DNN's computations have, which allows an adversary to estimate (1) a sequence of layers accessed by a DNN while processing input and (2) the time it takes to process the input in each layer. One can hardly expect such regularities from the conventional software; an adversary who launches the same attack may face the path explosion problem. We show that a cache side-channel attack, Flush+Reload, reliably extracts the sequence and timings while a DNN processes a single input. Based on this finding, we present an algorithm that reconstructs a DNN architecture from the computational trace extracted via Flush+Reload. In our evaluation, we consider an adversary who monitors a victim's VM running a DNN remotely via Flush+Reload and demonstrate the reconstruction of two unique architectures: (1) MalConv: a unique data pre-processing pipeline for malware detection and (2) ProxylessNAS-CPU: a unique architecture found by neural architecture search, with no reconstruction error.

In Chapter 5: We show that the computational savings promised by input-adaptive multi-exit DNNs are *a house of cards* in an adversarial setting. We first argue that this attack objective is *orthogonal* to the traditional attacker's—*i.e.*., causing misclassifications. To this end, we show that the standard adversarial examples reduce the computations required for (mis-)predictions than clean inputs. We then craft the objective of an adversarial-example crafting algorithm (PGD) to increase the computations of multi-exit architectures. We call this resulting attack DeepSloth. In evaluation, the white-box DeepSloth that individually crafting adversarial perturbations for each sample can completely offset the computational savings of a victim multi-exit model. A universal perturbation crafted with multiple samples can also reduce the computational savings up to 45%. We further show that the DeepSloth transfers to a model with different datasets, architectures, and input-adaptive mechanisms. Moreover, we examine the practicality of this attack in a typical scenario that facilitates multi-exit DNNs, *e.g.*, the Internet of Things (IoT), and demonstrate that DeepSloth can increase the latency of a victim's system by 1.5–5x. We finally examine the effectiveness of standard defenses against adversarial input perturbations, *i.e.*, adversarial training, and expose a trade-off between efficiency, robustness to adversarial examples, and robustness to DeepSloth.

6.2 Vision and Future Work

This dissertation opens up many research challenges and opportunities.

Exposing new threats to deep learning systems caused by hardware-level attacks. Recent studies [169, 170, 171] exposes various hardware-level attacks, and they pose security threats to systems. For example, Bulck *et al.* [169] showed that an adversary can exploit Linux's page-fault mechanisms to break the confidentiality guaranteed by Intel® SGX. However, it is unknown whether an adversary can utilize this threat to attack deep learning systems. As shown in Chapter 3, 4 and 5, existing attacks can pose unexpected damage, or oftentimes, can be more effective against deep learning systems. The emergence of new computing hardware, such as GPUs, Tensor Processing Units (TPUs)[1], or Quantum Processing Units (QPUs)[2], can also enlarge attack surfaces that an adversary can exploit. It is unknown how vulnerable the hardware is and how an adversary

[1]`https://cloud.google.com/tpu`
[2]`https://www.dwavesys.com`

can exploit the vulnerability; for example, a motivated attacker may use quantum-level interference to risk the availability of deep learning systems. In future work, to build secure and reliable deep learning systems, we need to study the worst-case behaviors of new hardware in adversarial settings.

Hardware-level attacks as a vehicle to understand DNN models. My dissertation exposes new security threats to under-examined attack methods such as hardware fault attacks or side-channels. However, to come up with defense mechanisms against those threats, it is essential to understand the root cause: *Why do DNNs have computational properties that an adversary can exploit with hardware attacks?* To answer this question, we can study the empirical connections between hardware attacks and their impact on generalization measures. As the initial work on adversarial examples [11], we use bitwise corruptions (bit-flip attacks) as a tool for quantifying the sensitivity of model parameters to perturbations. Such quantification enables us to define the relationship between a generalization property and the vulnerability to our attacks. Building on this connection, I expect to propose defense mechanisms against the new threats.

Hardware-software co-design for building secure and reliable deep learning systems. After we study "emerging security threats" and "their root-causes", we can ask the question: *how can we defeat them?* One way to think about the solution is to take a hardware-software co-design approach. Defense mechanisms solely depend on machine learning may fail at some point. For example, in Chapter 5, we demonstrate that a standard defense against adversarial input perturbations, *i.e.*, adversarial training, cannot provide sufficient resilience to DeepSloth. On the other hand, defenses that rely on hardware or system-level mechanisms cannot defeat our new threats. We show in Chapter 3 that hardware-level defenses against Rowhammer are not effective against the *blind* attacker. Thus, I envision characterizing the limits of existing defenses against new threats and combining both hardware- and machine learning-level defenses to close the attack surfaces

Appendix A: Experimental Details and Additional Experiments

A.1 Lines of Code Monitored via Flush+Reload

Table A.1 shows the exact lines of code we monitor in the PyTorch and TensorFlow frameworks. We use PyTorch v1.2.0[1] and Tensorflow v1.14.0[2]. In both frameworks, we can monitor a similar set of DL computations in the C++ native implementations. It allows an adversary to use one algorithm for attacking models running on both frameworks. We also can see the back-end libraries supporting the matrix multiplications are different, *i.e.*, PyTorch is compiled with OpenBLAS, whereas TensorFlow uses Eigen and MKL-DNN. However, they both implement the matrix multiplications using GOTO's algorithm [172]. Therefore, our attacker can monitor the same information—the number of iterations of for-loops—and finally, estimate the overall size of matrix multiplication.

A.2 Applicability to GPUs

Our attack is not fundamentally different for GPUs. In most deep learning frameworks, when a network performs a computation, it invokes the same function implemented in C++, and the function decides whether the back-end computation can use GPUs or not.

[1] `https://github.com/pytorch/pytorch/commit/8554416a199c4cec01c60c7015d8301d2bb39b64`

[2] `https://github.com/tensorflow/tensorflow/commit/87989f69597d6b2d60de8f112e1e3cea23be7298`

Framework		Computations	Line of Code
PyTorch	Core (C++)	Conv2d	aten/src/ATen/native/LegacyNNDefinitions.cpp:49
		Conv1d	aten/src/ATen/native/Convolution.cpp:448
		BatchNorm	aten/src/ATen/native/Normalization.cpp:428
		FC (Begin)	aten/src/TH/generic/THBlas.cpp:329
		FC (End)	aten/src/TH/generic/THBlas.cpp:499
		ReLU6	aten/src/ATen/LegacyTHFunctionsCPU.cpp:21320
		Sigmoid	aten/src/ATen/native/UnaryOps.cpp:191
		AvgPool	aten/src/ATen/native/AdaptiveAveragePooling.cpp:324
		MaxPool	aten/src/ATen/native/Pooling.cpp:50
		Embeddings	aten/src/ATen/native/Embedding.cpp:15
		add	aten/src/ATen/native/BinaryOps.cpp:46
		* (multiply)	aten/src/ATen/native/BinaryOps.cpp:88
		transpose	aten/src/ATen/natinsve/TensorShape.cpp:643
		narrow	aten/src/ATen/native/TeorShape.cpp:364
	OpenBLAS	GEMM(conv)	driver/level3/level3.c
		GEMM(oncopy)	kernel/x86_64/sgemm_ncopy_4_skylakex.c:54
TensorFlow	Core (C++)	Conv2d	core/kernels/conv_2d.h:81
		BatchNorm	core/kernels/fused_batch_norm_op.cc:254
		DepthConv2d	core/kernels/depthwise_conv_op.cc:161
		FC	core/kernels/matmul_op.cc:542
		ReLU6	core/kernels/relu_op_functor.h:68
		AvgPool	core/kernels/avgpooling_op.cc:80
		TensorAdd	core/kernels/cwise_ops_common.h:93
	Eigen	#FCNodes	src/Core/products/GeneralMatrixVector.h:155
	MKL-DNN	GEMM	src/cpu/gemm/gemm_driver.cpp:242
		(loop)M-outer	src/cpu/gemm/gemm_driver.cpp:349
		(loop)K	src/cpu/gemm/gemm_driver.cpp:355
		(loop)N	src/cpu/gemm/gemm_driver.cpp:372
		(loop)M-inner	src/cpu/gemm/gemm_driver.cpp:387

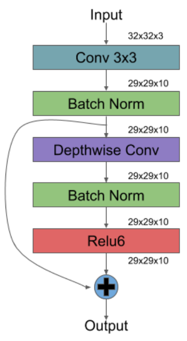
Table A.1: **Monitored lines of code in PyTorch and TensorFlow.**

This practice[3] maximizes the hardware compatibility of a framework; however, this also makes the framework vulnerable to our attacker who can still observe the common functions listed in Table A.1 by monitoring the shared cache. On GPUs, the timings would be different, so we would have to profile the computational times, *e.g.*, the time taken for the matrix multiplication with various sizes of tensor operands. However, on both CPUs and GPUs, the computation time is proportional to the size of tensor operands, which enables

[3]Customization of operations in TensorFlow: `https://www.tensorflow.org/guide/create_op`

our attacker to estimate the architecture parameters with timing observations.

A.3 ToyNet Computations Monitored in PyTorch and TensorFlow



Processed Trace (PyTorch)	**Processed Trace (TensorFlow)**
[0] Conv2d, 52230076, (1 and 2)	[0] Conv2d, 4717917, (20 and 15)
[1] BatchNorm2d, 54230076, 0	[1] BatchNorm2d, 4725816, 0
[3] Conv2d, 55600076, (10 and 10)	[2] DepthwiseConv, 4725912, 0
[4] BatchNorm2d, 59268076, 0	[3] BatchNorm2d, 4726228, 0
[5] ReLU6, 59880076, 0	[4] ReLU6, 4727308, 0
[6] add, 60270076, 0	[5] TensorAdd, 4727644, 0

Figure A.1: **The traces extracted from PyTorch and TensorFlow via Flush+Reload.** We illustrate the ToyNet architecture on the left. The sequence of computations observed from our attack are listed in the middle (PyTorch) and the right (TensorFlow).

Figure A.1 describes the reconstruction process of a small network in both the PyTorch and TensorFlow frameworks. On the left, we have the ground truth of the ToyNet architecture, which represents an example of a possible residual block in a victim network. In the middle and right, we show the observations of an adversary monitoring both PyTorch and TensorFlow code. The first entry indicates the monitored function corresponding to the desired architectural attribute. The second entry indicates the timestamp at which the adversary observes these functions, and the last entry is the number of general matrix multiplication (GEMM) function calls for the given layer observation.

Naming conventions vary slightly between the two frameworks, but the information inferred is the same. The adversary attacking both networks sees function calls that correspond to architectural attributes in the same order: Conv2d, BatchNorm2d, Conv2d / DepthwiseConv, BatchNorm2d, ReLU6, and TensorAdd. PyTorch does not distinguish between Conv2d and DepthwiseConv, but as stated in 4.3.1, we can differentiate the layers by timing data. Additionally, PyTorch and TensorFlow use different linear algebra libraries to perform matrix computation, so the implementations differ slightly. However, they both use variations on matrix multiplication algorithms that take into account system-level optimizations, such as cache size (*e.g.*, GOTO's algorithm [172]). In both cases, we observe operations in nested iterations of these implementations and are able to monitor instructions that correspond to the size of the matrices being multiplied, giving an adversary the ability to estimate the parameters of the convolution layers.

To perform the estimations of these layer parameters, the adversary can profile candidates offline on similar hardware. They can then create a data set of candidate parameters for given observation ranges. For instance, the number of observed GEMM calls in the PyTorch example for the depthwise convolution layer gives the attacker the information that there are 10 output channels, and therefore also ten output channels in the 1st convolution. Additionally, the observed GEMM calls for the 1st convolution layer give the candidate kernel sizes of 3 and 5. Likewise in TensorFlow, the observed instructions fit the candidate kernel sizes of 3 or 5, and 0-24 output channels. Therefore, these exploitable vulnerabilities exist independent of the specific deep learning framework a victim is using.

A.4 MNasNet Search Space

Tan *et al.* [137] utilize a hierarchical search space over six parameters: *ConvOp*, *KernelSize*, *SERatio*, *SkipOp*, *FilterSize*, and *#Layers*. They choose to partition a CNN into a known, finite set of blocks and then further divide these blocks into possibly repeated layers.

The number of repeats per layer in a given block i is a searchable parameter N_i, which is bounded at ± 1 the number of layers in MobileNetV2 on which block i is based. These layers are further divided into three possible network layers (*ConvOp*): regular convolution, depthwise convolution, or mobile inverted bottleneck convolution. Additionally, the network layer parameters can vary. These parameters include the convolution kernel size (*KernelSize*), the squeeze-and-excitation ratio (*SERatio*), a possible skip op (*SkipOp*), and the output filter size (*FilterSize*). The squeeze-and-excitation ration (*SERatio*) of a given layer varies between 0 and 0.025; the convolution kernel size varies between 3 and 5; the skip op is either pooling, identity residual, or no skip; and the filter size varies between 0.75, 1.0, and 1.25 the filter size of the corresponding block in MobileNetV2. Overall, this gives a claimed typical search space size of 10^{13} possibilities with 5 blocks, 3 average layers per block, and 432 options for the sub-search space of each block. This size compares to the per-layer approach with the same parameters that have a search space size 10^{39}.

A.5 Hyper-parameters for DeepSloth

We use the following hyper-parameters to craft adversarial perturbations.

ℓ_∞-based DeepSloth. We find that ℓ_∞-based DeepSloth does *not* require careful tuning. For the standard attack, we set the total number of iterations to 30 and the step size to $\alpha = 0.002$. For the modified attacks for hurting or preserving the accuracy, we set the total number of iterations to 75 and the step size to $\alpha = 0.001$. We compute the standard perturbations using the entire 10k test-set samples in CIFAR-10 and Tiny Imagenet. For the universal variants, we set the total number of iterations to 12 and reduce the initial step size of $\alpha = 0.005$ by a factor of 10 every 4 iterations. To compute a universal perturbation, we use randomly chosen 250 (CIFAR-10) and 200 (Tiny Imagenet) training samples.

ℓ_2-based DeepSloth. For both the standard and universal attacks, we set the total number of iterations to 550 and the step size γ to 0.1. Our initial perturbation has the ℓ_2-norm of 1.0. Here, we use the same number of samples for crafting the standard and universal perturbations as the ℓ_∞-based attacks.

ℓ_1-based DeepSloth. For our standard ℓ_1-based DeepSloth, we set the total number of iterations to 250, the step size α to 0.5, and the gradient sparsity to 99. For the universal variants, we reduce the total number of iterations to 100 and set the gradient sparsity to 90. Other hyperparameters remain the same. We use the same number of samples as the ℓ_∞-based attacks, to craft the perturbations.

A.6 Cost of Crafting DeepSloth Samples

ATTACKS	TIME (SEC.)
PGD-20	38
PGD-20 (AVG.)	48
PGD-20 (MAX.)	475
DEEPSLOTH	44
UNIVERSAL DEEPSLOTH	2

Figure A.2: **Time it takes to craft adversarial examples.**

In Table A.2, we compare the cost of DeepSloth with other attack algorithms on a VGG16-

based CIFAR-10 model—executed on a single Nvidia Tesla-V100 GPU. For the universal DeepSloth, we measure the execution time for crafting a perturbation using one batch (250 samples) of the training set. For the other attacks, we measure the time for perturbing the whole test set of CIFAR-10. Our DeepSloth takes roughly the same time as the PGD and PGD-avg attacks and significantly less time than the PGD-max attack. Our universal DeepSloth takes only 2 seconds (10x faster than DeepSloth) as it only uses 250 samples.

A.7 Empirical Evaluation of ℓ_1 and ℓ_2 DeepSloth

NETWORK	MSDNET		VGG16		MOBILENET	
SET.	RAD<5%	RAD<15%	RAD<5%	RAD<15%	RAD<5%	RAD<15%
			BASELINE (NO ATTACK)			
C10	0.89 / 85%	0.89 / 85%	0.77 / 89%	0.89 / 79%	0.83 / 87%	0.92 / 79%
TI	0.64 / 55%	0.83 / 50%	0.39 / 57%	0.51 / 52%	0.42 / 57%	0.59 / 51%
			DEEPSLOTH			
C10	0.36 / 51%	0.35 / 51%	0.12 / 36%	0.34 / 45%	0.18 / 41%	0.49 / 53%
TI	0.23 / 37%	0.51 / 40%	0.08 / 22%	0.15 / 25%	0.08 / 33%	0.19 / 35%
			UNIVERSAL DEEPSLOTH			
C10	0.89 / 83%	0.89 / 83%	0.75 / 85%	0.88 / 75%	0.82 / 85%	0.92 / 77%
TI	0.64 / 55%	0.83 / 50%	0.38 / 57%	0.51 / 52%	0.41 / 57%	0.59 / 51%
			CLASS-UNIVERSAL DEEPSLOTH			
C10	0.88 / 73%	0.88 / 73%	0.69 / 78%	0.86 / 67%	0.76 / 74%	0.89 / 65%
TI	0.64 / 54%	0.83 / 49%	0.39 / 59%	0.50 / 58%	0.41 / 60%	0.58 / 53%

Table A.2: **The effectiveness of ℓ_1 DeepSloth.** 'RAD<5,15%' columns list the results in each early-exit setting. Each entry includes the model's efficacy score (*left*) and accuracy (*right*). The class-universal attack's results are an average of 10 classes. 'TI' is Tiny Imagenet and 'C10' is CIFAR-10.

Table A.2 and Table A.3 shows the effectiveness of ℓ_1- and ℓ_2-based DeepSloth attacks. Our results show that the ℓ_1- and ℓ_2-based attacks are less effective than the ℓ_∞-based attacks. In contrast to the ℓ_∞-based attacks that eliminate the efficacy of victim multi-exit models, the ℓ_1- and ℓ_2-based attacks reduce the efficacy of the same models by 0.24$\sim$0.65.

NETWORK	MSDNET		VGG16		MOBILENET	
SET.	RAD<5%	RAD<15%	RAD<5%	RAD<15%	RAD<5%	RAD<15%
	BASELINE (NO ATTACK)					
C10	0.89 / 85%	0.89 / 85%	0.77 / 89%	0.89 / 79%	0.83 / 87%	0.92 / 79%
TI	0.64 / 55%	0.83 / 50%	0.39 / 57%	0.51 / 52%	0.42 / 57%	0.59 / 51%
	DEEPSLOTH					
C10	0.52 / 64%	0.52 / 64%	0.22 / 60%	0.45 / 62%	0.23 / 46%	0.48 / 55%
TI	0.24 / 42%	0.52 / 44%	0.13 / 35%	0.21 / 36%	0.12 / 38%	0.25 / 40%
	UNIVERSAL DEEPSLOTH					
C10	0.89 / 81%	0.89 / 81%	0.75 / 87%	0.88 / 76%	0.81 / 84%	0.92 / 76%
TI	0.63 / 54%	0.82 / 48%	0.38 / 56%	0.51 / 52%	0.41 / 56%	0.58 / 51%
	CLASS-UNIVERSAL DEEPSLOTH					
C10	0.88 / 73%	0.88 / 73%	0.71 / 81%	0.86 / 70%	0.76 / 76%	0.89 / 66%
TI	0.64 / 53%	0.83 / 49%	0.38 / 57%	0.50 / 57%	0.41 / 58%	0.58 / 53%

Table A.3: **The effectiveness of ℓ_2 DeepSloth.** 'RAD<5,15%' columns list the results in each early-exit setting. Each entry includes the model's efficacy score (*left*) and accuracy (*right*). The class-universal attack's results are an average of 10 classes. 'TI' is Tiny Imagenet and 'C10' is CIFAR-10.

Besides, the accuracy drops caused by ℓ_1- and ℓ_2-based attacks are in 6$\sim$21%, smaller than that of ℓ_∞-based DeepSloth (75$\sim$99%). Moreover, we see that the universal variants of ℓ_1- and ℓ_2-based attacks can barely reduce the efficacy of multi-exit models—they decrease the efficacy up to 0.08 and the accuracy by 12%.

A.8 Empirical Evaluation of DeepSloth on ResNet56

Table A.4 shows the the effectiveness of our DeepSloth attacks on ResNet56-base models. Our results show that ResNet56-based models are vulnerable to all the ℓ_∞, ℓ_2, and ℓ_1-based DeepSloth attacks. Using our ℓ_∞-based DeepSloth, the attacker can reduce the efficacy of the victim models to 0.00$\sim$0.01 and the accuracy by 39$\sim$68%. Besides, the ℓ_2, and ℓ_1-based attacks also decrease the efficacy to 0.04$\sim$0.18 and the accuracy by

NETWORK	RESNET (ℓ_∞)		RESNET (ℓ_1)		RESNET (ℓ_2)	
SET.	RAD<5%	RAD<15%	RAD<5%	RAD<15%	RAD<5%	RAD<15%
	BASELINE (NO ATTACK)					
C10	0.52 / 87%	0.69 / 80%	0.52 / 87%	0.69 / 80%	0.51 / 87%	0.69 / 80%
TI	0.25 / 51%	0.39 / 46%	0.25 / 51%	0.39 / 46%	0.25 / 51%	0.39 / 46%
	DEEPSLOTH					
C10	0.00 / 19%	0.01 / 19%	0.05 / 43%	0.18 / 47%	0.06 / 45%	0.17 / 48%
TI	0.00 / 7%	0.01 / 7%	0.04 / 27%	0.10 / 28%	0.05 / 34%	0.13 / 35%
	UNIVERSAL DEEPSLOTH					
C10	0.35 / 63%	0.59 / 60%	0.49 / 84%	0.68 / 75%	0.48 / 85%	0.67 / 76%
TI	0.25 / 25%	0.34 / 37%	0.25 / 51%	0.39 / 46%	0.25 / 51%	0.38 / 46%
	CLASS-UNIVERSAL DEEPSLOTH					
C10	0.23 / 33%	0.48 / 29%	0.39 / 70%	0.60 / 61%	0.39 / 71%	0.60 / 61%
TI	0.11 / 21%	0.23 / 18%	0.23 / 51%	0.36 / 46%	0.23 / 50%	0.36 / 46%

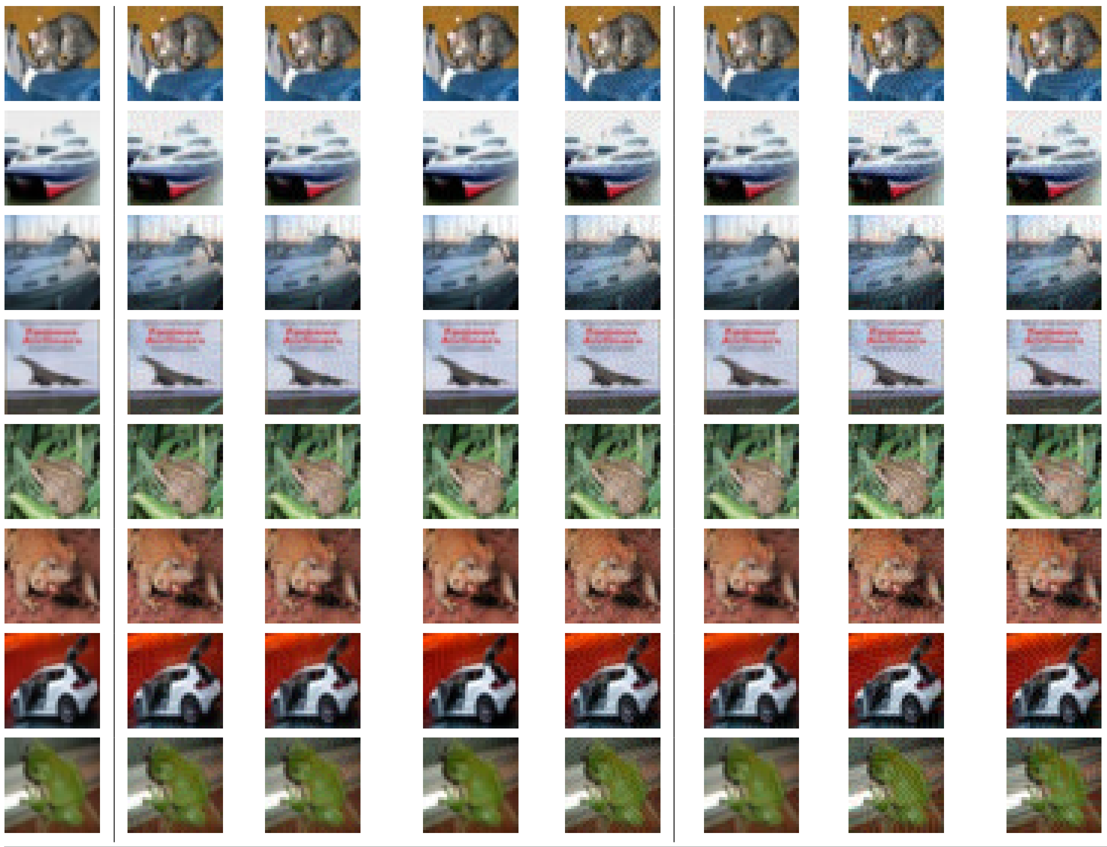
Table A.4: **The effectiveness of DeepSloth on the ResNet-based models.** 'RAD<5,15%' columns list the results in each early-exit setting. Each entry includes the model's efficacy score (*left*) and accuracy (*right*). The class-universal attack's results are an average of 10 classes.

11∼44%. Compared to the results on MSDNet, VGG16, and MobileNet in Table A.2 and A.3, the same attacks are more effective. The universal variants decrease the efficacy up to 0.21 and the accuracy up to 24%. In particular, the ℓ_2, and ℓ_1-based attacks (on C10 models) are effective than the same attacks on MSDNet, VGG16, and MobileNet-based models.

A.9 Adversarial Examples from Standard Attacks and DeepSloth

Figure A.3 shows the adversarial examples from the PGD, UAP and our DeepSloth.

	Standard Attacks				DeepSloth		
Original	PGD	PGD (avg.)	PGD (max.)	UAP	Per-sample	Universal	Class-specific



$\ell_{\inf}$ of the Perturbations (on Average)

$\ell_{\inf}$	0.03	0.03	0.03	0.03	0.03	0.03	0.03

Figure A.3: **Adversarial examples from the standard and our DeepSloth attacks.** The leftmost column shows the clean images. In the next four columns, we show adversarial examples from PGD, PGD (avg.), PGD (max.), and UAP attacks, respectively. The last four columns include adversarial examples from the three variants of DeepSloth. Each row corresponds to each sample, and the last row contains the average $\ell_{\inf}$-norm of the perturbations over the eight samples in each attack.

Bibliography

[1] Andrei Tatar, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Defeating software mitigations against rowhammer: a surgical precision hammer. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 47–66. Springer, 2018.

[2] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, page 1097–1105, Red Hook, NY, USA, 2012. Curran Associates Inc.

[3] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech Recognition with Deep Recurrent Neural Networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 6645–6649, 2013.

[4] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. *CoRR*, abs/1609.08144, 2016.

[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

[6] Tesla. Autopilot AI — Tesla. https://www.tesla.com/autopilotAI, 2015. [Accessed: 2021-06-24].

[7] Daniel A Hashimoto, Guy Rosman, Daniela Rus, and Ozanan R Meireles. Artificial intelligence in surgery: promises and perils. *Annals of surgery*, 268(1):70, 2018.

[8] Elon Musk's Neuralink 'shows monkey playing Pong with mind'. https://www.bbc.com/news/technology-56688812. [Accessed: 2021-06-24].

[9] Uber's self-driving operator charged over fatal crash. https://www.bbc.com/news/technology-56688812. [Accessed: 2021-06-25].

[10] Researchers Bypass Apple FaceID Using Biometrics 'Achilles Heel'. https://threatpost.com/researchers-bypass-apple-faceid-using-biometrics-achilles-heel/147109/. [Accessed: 2021-06-25].

[11] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. In *International Conference on Learning Representations*, 2014.

[12] Florian Tramèr, Nicolas Papernot, Ian Goodfellow, Dan Boneh, and Patrick McDaniel. The space of transferable adversarial examples. *arXiv*, 2017.

[13] Yanpei Liu, Xinyun Chen, Chang Liu, and Dawn Song. Delving into transferable adversarial examples and black-box attacks. In *Proceedings of 5th International Conference on Learning Representations*, 2017.

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

[15] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. In *International Conference on Learning Representations*, 2018.

[16] Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Logan Engstrom, Brandon Tran, and Aleksander Madry. Adversarial examples are not bugs, they are features. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[17] Preetum Nakkiran. A discussion of 'adversarial examples are not bugs, they are features': Adversarial examples are just bugs, too. *Distill*, 2019. https://distill.pub/2019/advex-bugs-discussion/response-5.

[18] Joern-Henrik Jacobsen, Jens Behrmann, Richard Zemel, and Matthias Bethge. Excessive invariance causes adversarial vulnerability. In *International Conference on Learning Representations*, 2019.

[19] Florian Tramer, Jens Behrmann, Nicholas Carlini, Nicolas Papernot, and Joern-Henrik Jacobsen. Fundamental tradeoffs between invariance and sensitivity to adversarial perturbations. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 9561–9571. PMLR, 13–18 Jul 2020.

[20] Battista Biggio, Blaine Nelson, and Pavel Laskov. Poisoning attacks against support vector machines. In *Proceedings of the 29th International Coference on International Conference on Machine Learning*, ICML'12, page 1467–1474, Madison, WI, USA, 2012. Omnipress.

[21] Jacob Steinhardt, Pang Wei W Koh, and Percy S Liang. Certified defenses for data poisoning attacks. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[22] Ali Shafahi, W. Ronny Huang, Mahyar Najibi, Octavian Suciu, Christoph Studer, Tudor Dumitras, and Tom Goldstein. Poison frogs! targeted clean-label poisoning attacks on neural networks. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 6106–6116, Red Hook, NY, USA, 2018. Curran Associates Inc.

[23] W. Ronny Huang, Jonas Geiping, Liam Fowl, Gavin Taylor, and Tom Goldstein. Metapoison: Practical general-purpose clean-label data poisoning. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 12080–12091. Curran Associates, Inc., 2020.

[24] Jonas Geiping, Liam H Fowl, W. Ronny Huang, Wojciech Czaja, Gavin Taylor, Michael Moeller, and Tom Goldstein. Witches' brew: Industrial scale data poisoning via gradient matching. In *International Conference on Learning Representations*, 2021.

[25] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks Are Practical. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, page 605–622, USA, 2015. IEEE Computer Society.

[26] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, San Diego, CA, August 2014. USENIX Association.

[27] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[28] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[29] Yiğitcan Kaya, Sanghyun Hong, and Tudor Dumitraş. Shallow-Deep Networks: Understanding and mitigating network overthinking. In *Proceedings of the 2019 International Conference on Machine Learning (ICML)*, Long Beach, CA, Jun 2019.

[30] Sanghyun Hong, Varun Chandrasekaran, Yigitcan Kaya, Tudor Dumitras, and Nicolas Papernot. On the Effectiveness of Mitigating Data Poisoning Attacks with Gradient Shaping. *CoRR*, abs/2002.11497, 2020.

[31] Yann LeCun, John Denker, and Sara Solla. Optimal brain damage. In D. Touretzky, editor, *Advances in Neural Information Processing Systems*, volume 2. Morgan-Kaufmann, 1990.

[32] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. *CoRR*, abs/1608.08710, 2016.

[33] Ron Banner, Itay Hubara, Elad Hoffer, and Daniel Soudry. Scalable methods for 8-bit training of neural networks. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[34] Guanpeng Li, Siva Kumar Sastry Hari, Michael Sullivan, Timothy Tsai, Karthik Pattabiraman, Joel Emer, and Stephen W. Keckler. Understanding error propagation in deep learning neural network (dnn) accelerators and applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, New York, NY, USA, 2017. Association for Computing Machinery.

[35] Minghai Qin, Chao Sun, and Dejan Vucinic. Robustness of neural networks against storage media errors. *CoRR*, abs/1709.06173, 2017.

[36] Sanghyun Hong, Pietro Frigo, Yigitcan Kaya, Cristiano Giuffrida, and Tudor Dumitras. Terminal Brain Damage: Exposing the Graceless Degradation in Deep Neural Networks Under Hardware Fault Attacks. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 497–514, Santa Clara, CA, August 2019. USENIX Association.

[37] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. In *Concurrency: the Works of Leslie Lamport*, pages 203–226. 2019.

[38] Stephen R Mahaney and Fred B Schneider. Inexact agreement: Accuracy, precision, and graceful degradation. In *Proceedings of the fourth annual ACM symposium on Principles of distributed computing*, pages 237–249, 1985.

[39] Maurice P Herlihy and Jeannette M Wing. Specifying graceful degradation. *IEEE Transactions on Parallel and Distributed Systems*, 2(1):93–104, 1991.

[40] Per Brinch Hansen. *Operating system principles*. Prentice-Hall, Inc., 1973.

[41] Andreas Berl, Erol Gelenbe, Marco Di Girolamo, Giovanni Giuliani, Hermann De Meer, Minh Quan Dang, and Kostas Pentikousis. Energy-efficient cloud computing. *The computer journal*, 53(7):1045–1051, 2010.

[42] Minghai Qin, Chao Sun, and Dejan Vucinic. Robustness of neural networks against storage media errors, 2017.

[43] Yan Zhou, Murat Kantarcioglu, and Bowei Xi. Breaking transferability of adversarial samples with randomness, 2018.

[44] Edward Raff, Jon Barker, Jared Sylvester, Robert Brandon, Bryan Catanzaro, and Charles K Nicholas. Malware detection by eating a whole exe. In *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[45] Han Cai, Ligeng Zhu, and Song Han. ProxylessNAS: Direct neural architecture search on target task and hardware. In *International Conference on Learning Representations*, 2019.

[46] Gao Huang, Danlu Chen, Tianhong Li, Felix Wu, Laurens van der Maaten, and Kilian Weinberger. Multi-scale dense networks for resource efficient image classification. In *International Conference on Learning Representations*, 2018.

[47] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014.

[48] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 448–456, Lille, France, 07–09 Jul 2015. PMLR.

[49] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[50] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[51] Ian Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In *International Conference on Learning Representations*, 2015.

[52] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik, and Ananthram Swami. Practical black-box attacks against machine learning. In *Proceedings of the 2017 ACM on Asia conference on computer and communications security*, pages 506–519, 2017.

[53] N. Carlini and D. Wagner. Towards evaluating the robustness of neural networks. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 39–57, 2017.

[54] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. In *International Conference on Learning Representations*, 2018.

[55] Anish Athalye, Nicholas Carlini, and David Wagner. Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 274–283, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

[56] Florian Tramèr, Nicolas Papernot, Ian Goodfellow, Dan Boneh, and Patrick McDaniel. The space of transferable adversarial examples. *arXiv preprint arXiv:1704.03453*, 2017.

[57] Nathan Inkawhich, Wei Wen, Hai (Helen) Li, and Yiran Chen. Feature space perturbations yield more transferable adversarial examples. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[58] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. Adversarial machine learning at scale. *arXiv preprint arXiv:1611.01236*, 2016.

[59] Weilin Xu, David Evans, and Yanjun Qi. Feature squeezing: Detecting adversarial examples in deep neural networks. *arXiv preprint arXiv:1704.01155*, 2017.

[60] Yang Song, Taesup Kim, Sebastian Nowozin, Stefano Ermon, and Nate Kushman. Pixeldefend: Leveraging generative models to understand and defend against adversarial examples. In *International Conference on Learning Representations*, 2018.

[61] Fangzhou Liao, Ming Liang, Yinpeng Dong, Tianyu Pang, Xiaolin Hu, and Jun Zhu. Defense against adversarial attacks using high-level representation guided denoiser. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[62] M. Lecuyer, V. Atlidakis, R. Geambasu, D. Hsu, and S. Jana. Certified robustness to adversarial examples with differential privacy. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 656–672, 2019.

[63] Octavian Suciu, Radu Marginean, Yigitcan Kaya, Hal Daume III, and Tudor Dumitras. When does machine learning FAIL? generalized transferability for evasion and poisoning attacks. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1299–1316, Baltimore, MD, 2018. USENIX Association.

[64] Chen Zhu, W. Ronny Huang, Hengduo Li, Gavin Taylor, Christoph Studer, and Tom Goldstein. Transferable clean-label poisoning attacks on deep neural nets. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 7614–7623. PMLR, 09–15 Jun 2019.

[65] Yuzhe Yang, Guo Zhang, Dina Katabi, and Zhi Xu. Me-net: Towards effective adversarial robustness with matrix estimation. *arXiv preprint arXiv:1905.11971*, 2019.

[66] Sun Tzu, Sun Tzu, Wu Sun, Sun Cu Vu, et al. *The art of war*, volume 361. Oxford University Press, USA, 1971.

[67] G. An. The effects of adding noise during backpropagation training on a generalization performance. *Neural Computation*, 8(3):643–674, April 1996.

[68] Brandon Reagen, Udit Gupta, Lillian Pentecost, Paul Whatmough, Sae Kyu Lee, Niamh Mulholland, David Brooks, and Gu-Yeon Wei. Ares: A framework for quantifying the resilience of deep neural networks. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2018.

[69] Chenyi Chen, Ari Seff, Alain Kornhauser, and Jianxiong Xiao. Deepdriving: Learning affordance for direct perception in autonomous driving. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2722–2730, 2015.

[70] Nikolai Smolyanskiy, Alexey Kamenev, Jeffrey Smith, and Stan Birchfield. Toward low-flying autonomous mav trail navigation using deep neural networks for environmental awareness. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4241–4247. IEEE, 2017.

[71] Mark Seaborn and Thomas Dullien. Exploiting the dram rowhammer bug to gain kernel privileges.

[72] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[73] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 448–456, Lille, France, 07–09 Jul 2015. PMLR.

[74] 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 *ACM SIGARCH Computer Architecture News*, volume 42, pages 361–372. IEEE Press, 2014.

[75] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. CLKSCREW: Exposing the perils of security-oblivious energy management. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1057–1074, Vancouver, BC, 2017. USENIX Association.

[76] Jakub Breier, Xiaolu Hou, Dirmanto Jap, Lei Ma, Shivam Bhasin, and Yang Liu. Practical fault attack on deep neural networks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 2204–2206, New York, NY, USA, 2018. ACM.

[77] Yannan Liu, Lingxiao Wei, Bo Luo, and Qiang Xu. Fault injection attack on deep neural network. In *Proceedings of the 36th International Conference on Computer-Aided Design*, pages 131–138. IEEE Press, 2017.

[78] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. Flip feng shui: Hammering a needle in the software stack. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 1–18, Austin, TX, 2016. USENIX Association.

[79] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. One bit flips, one cloud flops: Cross-vm row hammer attacks and privilege escalation. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 19–35, Austin, TX, 2016. USENIX Association.

[80] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer. js: A remote software-induced fault attack in javascript. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 300–321. Springer, 2016.

[81] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Dedup est machina: Memory deduplication as an advanced exploitation vector. In *2016 IEEE symposium on security and privacy (SP)*, pages 987–1004. IEEE, 2016.

[82] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Grand pwning unit: accelerating microarchitectural attacks with the gpu. In *Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU*, page 0. IEEE, 2018.

[83] Victor Van Der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic rowhammer attacks on mobile platforms. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 1675–1689. ACM, 2016.

[84] Moritz Lipp, Misiker Tadesse Aga, Michael Schwarz, Daniel Gruss, Clémentine Maurice, Lukas Raab, and Lukas Lamster. Nethammer: Inducing rowhammer faults through network requests. *arXiv preprint arXiv:1805.04956*, 2018.

[85] Andrei Tatar, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Throwhammer: Rowhammer attacks over the network and defenses.

[86] 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, Austin, TX, 2016. USENIX Association.

[87] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[88] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009.

[89] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[90] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[91] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[92] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *International Conference on Learning Representations (ICLR)*, 2015.

[93] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[94] Forrest Iandola, Matt Moskewicz, Sergey Karayev, Ross Girshick, Trevor Darrell, and Kurt Keutzer. Densenet: Implementing efficient convnet descriptor pyramids. *arXiv preprint arXiv:1404.1869*, 2014.

[95] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the Inception Architecture for Computer Vision. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2818–2826, 2016.

[96] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[97] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical evaluation of rectified activations in convolutional network, 2015.

[98] Jidong Xiao, Zhang Xu, Hai Huang, and Haining Wang. Security implications of memory deduplication in a virtualized environment. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12. IEEE, 2013.

[99] Bolun Wang, Yuanshun Yao, Bimal Viswanath, Haitao Zheng, and Ben Y. Zhao. With great training comes great vulnerability: Practical attacks against transfer learning. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1281–1297, Baltimore, MD, 2018. USENIX Association.

[100] Johannes Stallkamp, Marc Schlipsing, Jan Salmen, and Christian Igel. Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition. *Neural networks*, 32:323–332, 2012.

[101] Maria-Elena Nilsback and Andrew Zisserman. Automated flower classification over a large number of classes. In *Computer Vision, Graphics & Image Processing, 2008. ICVGIP'08. Sixth Indian Conference on*, pages 722–729. IEEE, 2008.

[102] Sanjay Ghemawat. TCMalloc : Thread-Caching Malloc. `https://gperftools.github.io/gperftools/tcmalloc.html`, 2018.

[103] Jemalloc manual. Jemalloc: general purpose memory allocation functions. `http://jemalloc.net/jemalloc.3.html`, 2019.

[104] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. Another flip in the wall of rowhammer defenses. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 245–261. IEEE, 2018.

[105] Daniel Gruss, Erik Kraft, Trishita Tiwari, Michael Schwarz, Ari Trachtenberg, Jason Hennessey, Alex Ionescu, and Anders Fogh. Page cache attacks. *arXiv preprint arXiv:1901.01161*, 2019.

[106] Ferdinand Brasser, Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. Can't touch this: Software-only mitigation against rowhammer attacks targeting kernel memory. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 117–130, Vancouver, BC, 2017. USENIX Association.

[107] 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 *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 697–710, Carlsbad, CA, 2018. USENIX Association.

[108] Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Rui Qiao, Reetuparna Das, Matthew Hicks, Yossi Oren, and Todd Austin. Anvil: Software-based protection against next-generation rowhammer attacks. *ACM SIGPLAN Notices*, 51(4):743–755, 2016.

[109] Barry L Kalman and Stan C Kwasny. Why tanh: Choosing a sigmoidal function. In *Neural Networks, 1992. IJCNN., International Joint Conference on*, volume 4, pages 578–581. IEEE, 1992.

[110] Alex Krizhevsky and Geoff Hinton. Convolutional deep belief networks on cifar-10. *Unpublished manuscript*, 40(7), 2010.

[111] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *ECCV*, 2016.

[112] Yan Wang, Wei-Lun Chao, Divyansh Garg, Bharath Hariharan, Mark Campbell, and Kilian Q. Weinberger. Pseudo-lidar from visual depth estimation: Bridging the gap in 3d object detection for autonomous driving. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[113] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning transferable architectures for scalable image recognition. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[114] David So, Quoc Le, and Chen Liang. The evolved transformer. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 5877–5886, Long Beach, California, USA, 09–15 Jun 2019. PMLR.

[115] Ambra Demontis, Marco Melis, Maura Pintor, Matthew Jagielski, Battista Biggio, Alina Oprea, Cristina Nita-Rotaru, and Fabio Roli. Why do adversarial attacks transfer? explaining transferability of evasion and poisoning attacks. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 321–338, Santa Clara, CA, August 2019. USENIX Association.

[116] Qixue Xiao, Yufei Chen, Chao Shen, Yu Chen, and Kang Li. Seeing is not believing: Camouflage attacks on image scaling algorithms. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 443–460, Santa Clara, CA, August 2019. USENIX Association.

[117] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy*, pages 605–622, May 2015.

[118] Lejla Batina, Shivam Bhasin, Dirmanto Jap, and Stjepan Picek. CSI NN: Reverse engineering of neural network architectures through electromagnetic side channel. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 515–532, Santa Clara, CA, August 2019. USENIX Association.

[119] Weizhe Hua, Zhiru Zhang, and G. Edward Suh. Reverse engineering convolutional neural networks through side-channel information leaks. In *Proceedings of the 55th Annual Design Automation Conference*, DAC '18, pages 4:1–4:6, New York, NY, USA, 2018. ACM.

[120] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. Scattercache: Thwarting cache attacks via cache set randomization. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 675–692, Santa Clara, CA, August 2019. USENIX Association.

[121] Yuval Yarom and Katrina Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, San Diego, CA, August 2014. USENIX Association.

[122] Mengjia Yan, Christopher W. Fletcher, and Josep Torrellas. Cache telepathy: Leveraging shared resource attacks to learn DNN architectures. *CoRR*, abs/1808.04761, 2018.

[123] Vasisht Duddu, Debasis Samanta, D. Vijay Rao, and Valentina E. Balas. Stealing neural networks via timing side channels. *CoRR*, abs/1812.11720, 2018.

[124] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. *CoRR*, abs/1611.01578, 2016.

[125] Weizhe Hua, Zhiru Zhang, and G. Edward Suh. Reverse engineering convolutional neural networks through side-channel information leaks. In *Proceedings of the 55th Annual Design Automation Conference*, DAC '18, pages 4:1–4:6, New York, NY, USA, 2018. ACM.

[126] Sanghyun Hong, Michael Davinroy, Yigitcan Kaya, Stuart Nevans Locke, Ian Rackow, Kevin Kulda, Dana Dachman-Soled, and Tudor Dumitras. Security analysis of deep neural networks operating in the presence of cache side-channel attacks. *CoRR*, abs/1810.03487, 2018.

[127] Soumith Chintala Sam Gross Adam Paszke Francisco Massa Adam Lerer Gregory Chanan Zeming Lin Edward Yang Alban Desmaison Alykhan Tejani Andreas Kopf James Bradbury Luca Antiga Martin Raison Natalia Gimelshein Sasank Chilamkurthy Trevor Killeen Lu Fang Junjie Bai Benoit Steiner, Zachary DeVito. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, 2019.

[128] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard,

Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, Savannah, GA, November 2016. USENIX Association.

[129] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 199–212, New York, NY, USA, 2009. ACM.

[130] Y. Zhang, A. Juels, A. Oprea, and M. K. Reiter. Homealone: Co-residency detection in the cloud via side-channel analysis. In *2011 IEEE Symposium on Security and Privacy*, pages 313–328, May 2011.

[131] Adam Bates, Benjamin Mood, Joe Pletcher, Hannah Pruse, Masoud Valafar, and Kevin Butler. Detecting co-residency with active traffic analysis techniques. In *Proceedings of the 2012 ACM Workshop on Cloud Computing Security Workshop*, CCSW '12, pages 1–12, New York, NY, USA, 2012. ACM.

[132] T. Kohno, A. Broido, and K. C. Claffy. Remote physical device fingerprinting. *IEEE Transactions on Dependable and Secure Computing*, 2(2):93–108, April 2005.

[133] Venkatanathan Varadarajan, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. A placement vulnerability study in multi-tenant public clouds. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 913–928, Washington, D.C., Aug 2015. USENIX Association.

[134] Yuval Yarom. Mastik: A micro-architectural side-channel toolkit. *Retrieved from School of Computer Science Adelaide: http://cs. adelaide. edu. au/˜ yval/Mastik*, 16, 2016.

[135] Zeina Abu-Aisheh, Romain Raveaux, Jean-Yves Ramel, and Patrick Martineau. An exact graph edit distance algorithm for solving pattern recognition problems. 2015.

[136] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[137] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V. Le. Mnasnet: Platform-aware neural architecture search for mobile. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[138] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. STEALTHMEM: System-level protection against cache-based side channel attacks in the cloud. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 189–204, Bellevue, WA, 2012. USENIX.

[139] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 406–418, March 2016.

[140] C. Szegedy, Wei Liu, Yangqing Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, 2015.

[141] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[142] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal processing magazine*, 29(6):82–97, 2012.

[143] Mingxing Tan and Quoc Le. EfficientNet: Rethinking model scaling for convolutional neural networks. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 6105–6114, Long Beach, California, USA, 09–15 Jun 2019. PMLR.

[144] S. Teerapittayanon, B. McDanel, and H. T. Kung. Branchynet: Fast inference via early exiting from deep neural networks. In *2016 23rd International Conference on Pattern Recognition (ICPR)*, pages 2464–2469, 2016.

[145] Ting-Kuei Hu, Tianlong Chen, Haotao Wang, and Zhangyang Wang. Triple wins: Boosting accuracy, robustness and efficiency together by enabling input-adaptive inference. In *International Conference on Learning Representations*, 2020.

[146] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[147] Ben Taylor, Vicent Sanz Marco, Willy Wolff, Yehia Elkhatib, and Zheng Wang. Adaptive deep learning model selection on embedded systems. *ACM SIGPLAN Notices*, 53(6):31–43, 2018.

[148] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami. The limitations of deep learning in adversarial settings. In *2016 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 372–387, 2016.

[149] Elias De Coninck, Tim Verbelen, Bert Vankeirsbilck, Steven Bohez, Pieter Simoens, Piet Demeester, and Bart Dhoedt. Distributed neural networks for internet of things: The big-little approach. In *International Internet of Things Summit*, pages 484–492. Springer, 2015.

[150] 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.

[151] Xin Wang, Fisher Yu, Zi-Yi Dou, Trevor Darrell, and Joseph E. Gonzalez. SkipNet: Learning Dynamic Routing in Convolutional Networks. In *The European Conference on Computer Vision (ECCV)*, September 2018.

[152] Michael Figurnov, Maxwell D. Collins, Yukun Zhu, Li Zhang, Jonathan Huang, Dmitry Vetrov, and Ruslan Salakhutdinov. Spatially Adaptive Computation Time for Residual Networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[153] Mirazul Haque, Anki Chauhan, Cong Liu, and Wei Yang. Ilfo: Adversarial attack on adaptive neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.

[154] Tiny. Tiny ImageNet Visual Recognition Challenge. `http://tiny-imagenet.herokuapp.com/`. Accessed: 2020-09-28.

[155] Florian Tramèr and Dan Boneh. Adversarial training and robustness for multiple perturbations. In *Advances in Neural Information Processing Systems*, pages 5866–5876, 2019.

[156] Pin-Yu Chen, Yash Sharma, Huan Zhang, Jinfeng Yi, and Cho-Jui Hsieh. EAD: Elastic-Net Attacks to Deep Neural Networks via Adversarial Examples. *arXiv preprint arXiv:1709.04114*, 2017.

[157] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'17, page 615–629, New York, NY, USA, 2017. Association for Computing Machinery.

[158] En Li, Zhi Zhou, and Xu Chen. Edge intelligence: On-demand deep learning model co-inference with device-edge synergy. In *Proceedings of the 2018 Workshop on Mobile Edge Communications*, MECOMM'18, page 31–36, New York, NY, USA, 2018. Association for Computing Machinery.

[159] Li Zhou, Hao Wen, Radu Teodorescu, and David H.C. Du. Distributing deep neural networks with containerized partitions at the edge. In *2nd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 19)*, Renton, WA, July 2019. USENIX Association.

[160] Wangchunshu Zhou, Canwen Xu, Tao Ge, Julian McAuley, Ke Xu, and Furu Wei. BERT Loses Patience: Fast and Robust Inference with Early Exit. In *Advances in Neural Information Processing Systems*, 2020.

[161] Lu Hou, Zhiqi Huang, Lifeng Shang, Xin Jiang, Xiao Chen, and Qun Liu. DynaBERT: Dynamic BERT with Adaptive Width and Depth. In *Advances in Neural Information Processing Systems*, 2020.

[162] C. Hu, W. Bao, D. Wang, and F. Liu. Dynamic Adaptive DNN Surgery for Inference Acceleration on the Edge. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, pages 1423–1431, 2019.

[163] Weiwen Jiang, Edwin H.-M. Sha, Xinyi Zhang, Lei Yang, Qingfeng Zhuge, Yiyu Shi, and Jingtong Hu. Achieving super-linear speedup across multi-fpga for real-time dnn inference. *ACM Trans. Embed. Comput. Syst.*, 18(5s), October 2019.

[164] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, Omar Fawzi, and Pascal Frossard. Universal adversarial perturbations. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[165] Jérôme Rony, Luiz G Hafemann, Luiz S Oliveira, Ismail Ben Ayed, Robert Sabourin, and Eric Granger. Decoupling direction and norm for efficient gradient-based l2 adversarial attacks and defenses. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4322–4330, 2019.

[166] Se Jin Park, Murali Subramaniyam, Seoung Eun Kim, Seunghee Hong, Joo Hyeong Lee, Chan Min Jo, and Youngseob Seo. Development of the elderly healthcare monitoring system with iot. In *Advances in Human Factors and Ergonomics in Healthcare*, pages 309–315. Springer, 2017.

[167] Jiasi Chen and Xukan Ran. Deep learning with edge computing: A review. *Proceedings of the IEEE*, 107(8):1655–1674, 2019.

[168] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How Transferable Are Features in Deep Neural Networks? In *Advances in neural information processing systems*, pages 3320–3328, 2014.

[169] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A practical attack framework for precise enclave execution control. In *2nd Workshop on System Software for Trusted Execution (SysTEX)*, pages 4:1–4:6. ACM, October 2017.

[170] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval

Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[171] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[172] Kazushige Goto and Robert A. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, 34(3):12:1–12:25, May 2008.