

UC Davis

UC Davis Electronic Theses and Dissertations

Title

On-device Deep Learning For Security

Permalink

<https://escholarship.org/uc/item/1np674qd>

Author

Din, Zainul Abi

Publication Date

2021

Peer reviewed|Thesis/dissertation

On-device Deep Learning For Security

By

ZAINUL ABI DIN
DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Samuel T. King, Chair

Matthew K. Farrens

Hao Chen

Committee in Charge

2021

To my parents for their unflinching love and support.

Contents

Abstract	v
Acknowledgments	vii
Chapter 1. Introduction	1
1.1. Overview of Thesis	2
Chapter 2. In-browser visual ad blocking with on-device deep learning	7
2.1. Motivation	7
2.2. Introduction	7
2.3. Contributions	9
2.4. Overview	10
2.5. Design and Implementation	13
2.6. Deep Learning Pipeline	17
2.7. Evaluation	20
2.8. Limitations	28
2.9. Related Work	29
2.10. Future Work	31
Chapter 3. Preventing credit card fraud with on-device deep learning	32
3.1. Motivation	32
3.2. Introduction	33
3.3. Threat model, assumptions, and goal	35
3.4. Boxer design principles and overview	35
3.5. Image analysis	37
3.6. Secure counting	45

3.7. Implementation	49
3.8. Evaluation	50
3.9. Related work	56
Chapter 4. Wide scale measurement study with on-device deep learning	58
4.1. Motivation	58
4.2. Measurement study	59
Chapter 5. Ethical anti-fraud systems for mobile payments	65
5.1. Motivation	65
5.2. Overview	67
5.3. Design	70
5.4. Evaluation	78
5.5. Related work	91
5.6. Future Work	92
Chapter 6. Conclusion	94
Bibliography	96

Abstract

Deep learning has facilitated human-level performance on several tasks spanning a multitude of domains such as computer vision, natural language processing, medical analysis, gaming, retail, and marketing, just to name a few. The ability to solve a problem end-to-end, learn self-supervised high-level features from the data, and minimal hand-engineering have been key contributing factors in its success.

Due to the success of deep learning in related areas, it is also making in-roads into security. While a complete automation of a practical security system may be a remote prospect, we have seen many security sub-systems being upgraded with deep learning capabilities. For its self-learning capabilities, deep learning has been successfully used for enterprise-level network intrusion detection, malware detection and analysis, spam and phishing detection, and data privacy protection.

This work brings deep learning for security closer to the end-user. In addition to providing latency and scalability benefits, it enables a path away from privacy-invasive training and inference procedures.

In the first part of this work, I introduce Percival, an in-browser, deep learning powered native perceptual ad blocker implemented in two browsers. Percival advances the state of ad blocking and defends against a variety of attacks published against ad blockers, all while running purely client-side without any server intervention.

In the second part of this work, I present Boxer, a client-side Software Development Kit (SDK) and a server that can be used to prevent credit card fraud. Boxer's client-side SDK scans user's credit card and extracts high-level privacy-preserving features, which it then sends to the server for further processing. Boxer runs the entire deep learning inference client-side which ensures privacy-sensitive user data never leaves the user's device.

While Percival and Boxer respect end-user privacy and run machine learning inference client-side, the performance differences in running these models on end-devices could result in the compromise of the system utility or introduce bias into the decision process. Percival could degrade the browsing experience and Boxer could unfairly block a user with a low-end device.

In the third part of this work, I quantify the performance differences of running Boxer on the breadth of the devices one can see in distribution. Specifically, I perform a large-scale in-field study of running Boxer on front-end devices and quantify the impact of hardware diversity on the performance and reliability of Boxer’s machine learning pipeline. I identify the key performance metrics and design strategies that are critical for any on-device machine learning application.

Finally, in the last part of this work, I present a new anti-fraud payment card scanning system called Daredevil. Daredevil incorporates insights from the in-field measurement study and works well across the broad range of performance characteristics and hardware configurations found on modern mobile devices.

Acknowledgments

I am incredibly grateful and fortunate to have had Professor Sam King as my advisor. His passion for research, inspirational leadership, attention to detail, continued support and guidance have made this thesis possible. Thank you for believing in this problem and believing in me when I did not. Sam has not only advised this research but also worked alongside me like any graduate student. I have learned why *“numbers are better than theory”* and how fulfilling real-world impact can be. The guidance I have received, and the lessons learned in working with him will continue to remain an important influence for the rest of my life.

I would also like to thank my thesis committee members Professor Matt Farrens and Professor Hao Chen for their insightful comments and valuable feedback on this research. I was fortunate enough to take courses with both during the initial stages of my graduate studies which shaped my research worldview. I distinctly remember Professor Farrens’ golden nugget *“you are a researcher and not a salesman”*, a question I since ask myself every time I draft a paper or present my work. Special thanks to Professor Yong Jae Lee for in part co-advising this research and keeping us abreast of the latest developments in the field of computer vision.

I would like to thank my lab mates and collaborators who have been a significant part of this work. Thank you, Hari, for countless meetings, all-nighters, and endless collaborative writing sessions. Thank you, Jaime, Andy, Gary, and Sven for all your help implementing everything, however vague, we threw at you. Thank you, Henry, Adam, and Steven for helping us bridge the gap between academic research and industry which made sure that millions of people could use the systems we built.

I had the opportunity to intern at Brave Software during the summer of 2018. My work on user-facing AI at Brave, shepherded by Dr Ben Livshits, my mentor Panos and collaborator Pete served as the foundation to the work I did later in my research career. Thank you, Ben, Panos, Pete, Steven, Umar, Mo, Jan, Yan, Karen, Asad, Brad, Brian, David, Ailin, Slava, Michael, Jen, Mandar, Sabina, Ross, and Mihai for a wonderful summer of 2018.

Lastly, I would like to thank my parents who have been perfect role models for me. They have provided me with a lot of freedom and every opportunity to pursue what interests me the most. Thank you for your unconditional love and support and all the sacrifices you have made for me.

CHAPTER 1

Introduction

Deep learning is the principal engine and the most successful manifestation of AI, widely adopted in academia and enterprise alike. Its success can be ascribed to its inherent generalization capability, coupled with synchronous availability of data and computational resources, all with minimal overall human supervision. With deep learning, it is now possible to build systems that have better than human-level performance for tasks like image classification, since these models can learn these datasets more effectively.

However, the state-of-the-art results in deep learning come at the price of intensive use of computing resources. These developments in deep learning are accompanied with a trend towards massive and resource-hungry models that require specialized hardware (like GPUs and TPUs). The leading deep learning frameworks, TensorFlow and PyTorch, run models on GPUs or high-end servers in datacenters.

Little effort has been invested in building models that are suitable for tasks on edge devices, like mobile phones and laptops. It is estimated that by the year 2025, more than 75 billion devices [89] will be connected to the Internet. Most of these devices will be either hand-held or IoT devices with limited compute power, small memories, and the potential to generate a huge amount of data.

To meet the computational requirements of these compute-intensive models, applications use the cloud resources to offload all the computation from end devices to the server. For instance, voice processing services like Alexa and Siri only process wake words like “Alexa” or “Hey Siri” on the device, and then the following voice recording is sent to the server for further processing.

Offloading computation to the server has major drawbacks; for instance, it can take up to 200-ms round trip time for an AWS server to process an image frame [111]. This is close to 10 times slower than the rate at which modern smartphone cameras produce frames (15-ms to 30-ms per frame) [109]. Sending data from the end devices to the servers also introduces scalability issues since as the number of end devices increases, the propagation and queuing delays increase

significantly as well. Privacy is another major concern with server-based computation since the data is owned by the user or may contain user behavior or identifying characteristics.

On-device deep learning can provide latency, scalability, and privacy benefits to end users and apps alike. With a guarantee that user’s data never leaves their device, end-users are more likely to use these services freely and it also presents opportunities for app developers and researchers to provide a more secure ecosystem.

In this thesis, I propose and implement solutions to address the challenges of realizing deep learning for security at the end-devices. The substantial growth and complexity of security threats in recent times needs a reactive, autonomous and closer-to-the source response, and on-device deep learning can provide the necessary framework for the appropriate defenses. Accordingly, in this body of work, the core idea is that:

On-device deep learning is practical for improving client-side security and privacy.

This claim is validated as follows:

- A new system powered by deep learning is implemented for blocking advertisements in browsers. The system is dynamic and can adjust and generalize in the dynamic landscape of web advertisements. The system respects end-user privacy by running purely client-side on commodity computers, obviating the need to stream data out of the user’s device.
- A payment-card verification system is implemented that mobile apps can use to verify a user’s payment method. The system requires users to scan their payment cards to prove that they possess the real physical card. The system extracts privacy-preserving, high-fidelity features from the client-side and correlates this information with the rule-engine server-side to provide a holistic solution to payment-card verification. The payment-card verification system is evaluated with a large-scale in-field measurement study, spanning over 5 million real devices and 496 production apps to demonstrate its practical nature.

1.1. Overview of Thesis

In this section, I present a high-level description of the major aspects of this thesis. I describe the conceptualization of each system and how ideas from one system inform the design of the next.

The overarching goal of this thesis is to use client-side deep learning for security. This goal rests on the success of deep learning in many areas - particularly in computer vision, which has benefited the most from deep learning algorithms.

A natural application of computer vision to security is web advertisement blocking, due to the visual nature of advertisements. If a software agent capable of understanding images can identify which elements in a web-page correspond to advertisements, it can facilitate a more robust and automated approach to ad blocking. The first system presented in this work, Percival, is an illustration of taking the most basic deep learning algorithm (classification) and applying it to automate a traditional computer security task (blocking web advertisements). While the problem formulation is straightforward, the system design and the data required to train the system need significant engineering effort and innovation. The key ideas of streamlined data acquisition and efficient system design underlie all the subsequent systems and form a recurring theme in most of this thesis.

The second system presented in this work, Boxer, is an illustration of formulating a real world security challenge as a composition of multiple deep learning algorithms. From a high level, Boxer verifies payment cards by requiring users to scan their cards using their smartphone cameras and then processes the captured frames. Boxer answers the question of how to convert a sequence of payment card images scanned by a user to a security decision. In addition to data acquisition and system design, Boxer adds problem formulation as an important part of the system.

The third system presented in this work, Daredevil, demonstrates what can go wrong with a client side deep learning powered security system. If developers aren't careful, deep learning-based security checks can create structural biases in the system which can lead to disproportionate outcomes in at-risk populations. Daredevil illustrates the need for extensive measurements, particularly when the improper functioning of a system can lead to blocking of an end-user. Daredevil combines algorithmic optimizations, efficient system design, novel data generation and extensive measurement and evaluation to make deep learning powered payment card security challenges practical at the edge.

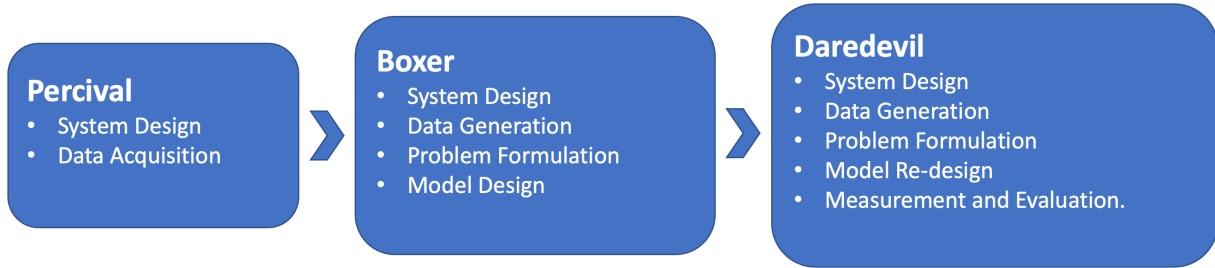


FIGURE 1.1. Major components of each system described in this thesis. Successive systems borrow and build off ideas from the previous systems.

In essence, the three systems describe the logical progression of ideas and detail the necessary parts of creating a workable security solution for end-devices (Figure 1.1). In the next sub-sections, I briefly describe the motivation, characteristics and evaluation of each system.

1.1.1. Percival: In-browser visual ad blocking with on-device deep learning. Online advertising has been a long-standing concern for user privacy and overall web experience. Several techniques have been proposed to block ads, mostly based on filter-lists and manually written rules. While a typical ad blocker relies on manually curated block lists, these inevitably get out-of-date, thus compromising the ultimate utility of this ad blocking approach.

In the first work, I present Percival, a browser-embedded, lightweight, deep learning-powered ad blocker. Percival embeds itself within the browser’s image rendering pipeline, which makes it possible to intercept every image obtained during page execution and to perform blocking based on applying deep learning for image classification to flag potential ads.

Percival’s implementation inside the Chromium browser shows only a minor rendering performance overhead of 4.55%, demonstrating the feasibility of deploying traditionally heavy models (i.e. deep neural networks) inside the critical path of the rendering engine of a browser.

To show the versatility of Percival’s approach case studies are presented that demonstrate that Percival does surprisingly well on ads in languages other than English and it also performs well on blocking first-party Facebook ads, which have presented issues for other ad blockers. Percival proves that image-based perceptual ad blocking is an attractive complement to today’s dominant approach of block lists.

1.1.2. Boxer: Preventing credit card fraud with on-device deep learning. Card-not-present credit card fraud costs businesses billions of dollars a year [72]. Combating card-not-present fraud is difficult since card-not-present transactions only require the card information and *not* the physical card, and fraudsters can easily acquire stolen credit card information online [14].

In the second work, I present Boxer, a mobile Software Development Kit (SDK) and a server that enables apps to combat card-not-present fraud by scanning physical cards and verifying that they are genuine. Boxer analyzes the images from these scans, looking for telltale signs of attacks (such as forgeries and images rendered on false media like screens), and introduces a novel abstraction on top of modern security hardware for complementary protection.

As of December 2020, over 450 apps have integrated Boxer and dozens of them have deployed it to production, including some large, popular, and international apps, resulting in Boxer scanning over 100 million real cards. The evaluation of Boxer from one of these deployments shows ten cases of real attacks that the novel hardware-based abstraction detects. Additionally, from the same deployment, without letting in any fraud, Boxer’s card scanning recovers 89% of the legitimate transactions, which the traditional fraud-prevention system used prior to Boxer would have otherwise blocked.

1.1.3. What can go wrong with Percival and Boxer? Bringing deep-learning-based security checks closer to the end-user improves privacy, scalability, and latency; however, running these checks on end-devices is non-trivial. The intense computation demands of running deep models coupled with the vastly different performance characteristics of end-devices can result in a stark difference in the security guarantees and utility provided by deep-learning-based checks. If not done carefully, Percival can be unusable on lower tier devices, while Boxer can potentially flag users simply for not being able to afford high-end phones, since these checks require substantial compute capabilities to run to completion. Considering the hardware and software differences between lower tier and higher tier devices, it is imperative to quantify the performance differences of the deep learning workloads on the distribution of devices one is likely to see in the wild.

To this end, an in-field measurement study of Boxer is conducted by running it on end-devices in real apps with the aim to quantify its performance across the device spectrum. The goal is to identify critical performance metrics and key system design strategies that can improve the performance

and reliability of Boxer. Data is collected from over 3,505,184 devices, and it demonstrates the stark difference in performance characteristics of resource-constrained and well-provisioned devices when running deep learning workloads for security. It is found that the same deep learning models and algorithms perform 41% worse on resource-constrained devices. Boxer is unable to run its checks fast enough on low-performing devices, which leads to poor user experience where end-users give up before completing these checks. Hence, Boxer is unable to verify the users of low-end devices.

1.1.4. Daredevil: Ethical anti-fraud systems for mobile payments. Using the insights gained from the measurement study on poor performance of low-end mobile devices for deep learning workloads, a new anti-fraud system, Daredevil, is designed. Daredevil works equally well across a broad range of performance characteristics and hardware configurations of modern mobile devices. Using data from 1,580,260 devices that run Daredevil, it is shown that Daredevil reduces the performance gap between resource-constrained and well-provisioned devices to less than 1%. Daredevil reduces the number of devices that run at less than one Frames Per Second (FPS) by an order of magnitude compared to Boxer, providing a more equitable system for fighting fraud.

CHAPTER 2

In-browser visual ad blocking with on-device deep learning

2.1. Motivation

The goal of this chapter is to determine whether the recent advances in the field of computer vision powered by deep learning can be leveraged to create a robust, practical, and generalizable solution to a traditional client-side security task of ad blocking. This chapter seeks to answer if it is practical to bring deep learning-based evaluation into an optimized, latency-sensitive task such as browser rendering.

Given the ubiquitous nature of web browsers, the techniques developed in this chapter can be utilized for optimizing and deploying other deep learning based tools within the browser pipeline. In essence, this chapter serves as a blueprint for any client-side deep learning system that seeks to improve security without affecting the overall usability of the system. The design principles, evaluation metrics and data acquisition strategies presented in this chapter serve as foundation for the chapters that follow.

In the following sections, I first motivate the need for ad blocking and why deep learning-based ad blockers can enable a better solution to this problem. This is followed by a description of some of the challenges of running deep learning-based models in the browser and how it informs the design and implementation of this work. Finally, I present a detailed evaluation of the system in both online (in-browser) and offline settings.

2.2. Introduction

Web advertising provides the financial incentives necessary to support most of the free content online, but it comes at a security and privacy cost. To make advertising effective, ad networks or publishers track user browsing behavior across multiple sites to generate elaborate user profiles for targeted advertising.

Users find that ads are intrusive [102] and cause disruptive browsing experience [1, 66]. In addition, studies have shown that advertisements impose privacy and performance costs to users, and carry the potential to be a malware delivery vector [18, 26, 61, 81, 82, 131].

Ad blocking is a software capability for filtering out unwanted advertisements to improve user experience, performance, security, and privacy. At present, ad blockers either run directly in the browser [48, 60] or as browser extensions [46].

Current ad blocking solutions filter undesired content based on “handcrafted” filter lists such as EasyList [126], which contain rules matching ad-carrying URLs and DOM elements. Most widely-used ad blockers, such as uBlock Origin [65] and Adblock Plus [46] use these block lists for content blocking. While useful, these approaches fail against adversaries who can change the ad-serving domain or obfuscate the web page code and metadata.

In an attempt to find a more flexible solution, researchers have proposed alternative approaches to ad blocking. One such approach is called Perceptual ad blocking, which relies on “visual cues” frequently associated with ads like the AdChoices logo or a sponsored content link. Storey et al. [118] built the first perceptual ad blocker that uses traditional computer vision techniques to detect ad-identifiers. Recently, Adblock Plus developers built filters into their ad blocker [70] to match images against a fixed template in order to detect ad labels. Due to the plethora of ad-disclosures, AdChoices logo and other ad-identifiers, it is unlikely that traditional computer vision techniques are sufficient and generalizable to the range of ads one is likely to see in the wild.

A natural extension to traditional vision-based blocking techniques is deep learning. Adblock Plus recently proposed SENTINEL [116] that detects ads in web pages using deep learning. SENTINEL’s deep learning model takes as input the screenshot of the rendered webpage to detect ads. However, this technology is still in development.

To this end, this chapter presents Percival, a native, deep learning-powered *perceptual ad blocker*, which is built into the browser image rendering pipeline. Percival intercepts every image obtained during the execution sequence of a page and blocks images that it classifies as ads. Percival is small (half the average webpage size [19]) and fast, and as part of this work implemented in two commercial browsers to block and detect ads at real-time.

Percival can be run *in addition* to an existing ad blocker, as a last-step measure to block whatever slips through its filters. However, Percival may also be deployed *outside* the browser, for example, as part of a crawler, whose job is to construct comprehensive block lists to supplement EasyList.

2.3. Contributions

This chapter makes the following contributions:

- **Perceptual ad blocking in Chromium-based browsers.** Percival is deployed in two Chromium-based browsers: Chromium and Brave. Two deployment scenarios are demonstrated; first, Percival blocks ads synchronously as it renders the page, with a modest performance overhead. Second, Percival classifies images *asynchronously* and memoizes the results, thus speeding up the classification process¹.
- **Lightweight and accurate deep learning models.** Percival embeds its machine learning framework as part of the Blink rendering engine of Chromium, which required significant strides in making the deep convolutional neural networks deployable with a light footprint on system resources. Percival shows that ad blocking can be done effectively using highly-optimized deep neural network-based models for image processing. Previous studies suggest that models over 5MB in size become hard to deploy on mobile devices [108]; because of the focus on low-latency detection, Percival consists of a compressed in-browser model that occupies 1.76MB² on disk, which is smaller by factor of 150 compared to other models of this kind [116], while maintaining similar accuracy results.
- **Accuracy and performance overhead measurements.** Percival’s perceptual ad blocking model can replicate EasyList rules with the accuracy of 96.76%, making Percival a viable and complementary ad blocking layer. The implementation within Chromium shows an average overhead of 178.23ms for page rendering. This overhead shows the feasibility of deploying deep neural networks inside the critical path of the rendering engine of the browser.

¹Source code, pre-trained models and data is available at <https://github.com/dxaen/percival>

²The in-browser model is 3.2MB due to a less efficient serialization format. Still, the weights are identical to the 1.76MB model

- **First-party ad blocking.** While the focus of traditional ad blocking is primarily on third-party ad blocking, Percival blocks first-party ads as well, such as those found on Facebook. Specifically, the experiments show that Percival blocks ads on Facebook (often referred to as “sponsored content”) with a 92% accuracy, with precision³ and recall⁴ of 78.4% and 70.0%.
- **Language-agnostic blocking.** Percival also blocks images that are in languages it did not train on. Percival is evaluated on Arabic, Chinese, Korean, French and Spanish image-based ads. It achieves an accuracy of 81.3% on Arabic, 95.1% on Spanish, and 93.9% on French datasets, with moderately high precision and recall. However, results from Chinese and Korean ads are less accurate.

2.4. Overview

This chapter presents Percival, a novel deep-learning based system for blocking ads. The primary goal is to build a system that blocks ad images that could escape detection by current techniques, while remaining small and efficient enough to run in a mobile browser.

Percival blocks rendering of ads using two key ideas. First, Percival runs in the browser image rendering pipeline. By running in the image rendering pipeline, Percival can inspect all images before the browser shows them to the user. Second, Percival uses a deep convolutional neural network (CNN) for detecting ad images. Using CNNs enables Percival to detect a wide range of ad images, even if they are in a language that Percival was not trained on.

This section discusses Percival’s architecture overview, possible alternative implementations and the detection model. Section 2.5 discusses the detailed design and implementation for the browser modifications.

2.4.1. Percival’s Architecture Overview. Percival’s detection module runs in the browser’s image decoding pipeline after the browser has decoded the image into pixels, but before it displays these pixels to the user. Running Percival after the browser has decoded an image takes advantage of the browser’s mature, efficient, and extensive image decoding logic, while still running at a choke

³Precision is defined as the percentage of blocked images that were originally ads.

⁴Recall is defined as the percentage of ad images blocked.

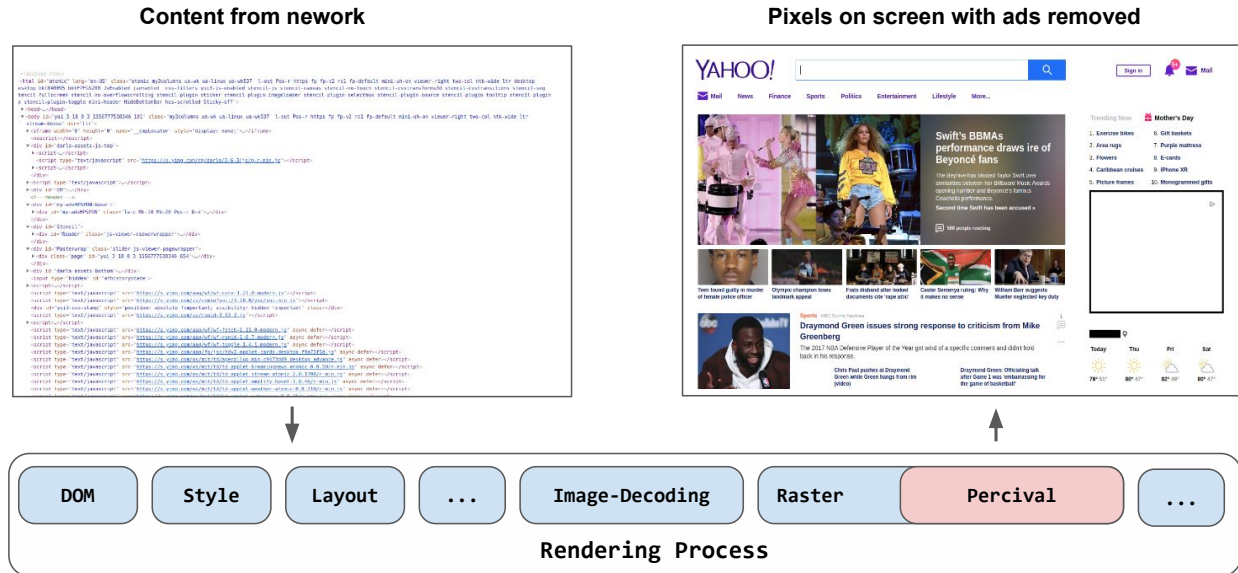


FIGURE 2.1. Overall architecture of Percival. Percival is positioned in the render process—which is responsible for creating rasterized pixels from HTML, CSS, Javascript. As the renderer process creates the Document Object Model (DOM) and decodes and rasterizes all image frames, these are first passed through Percival. Percival blocks the frames that are classified as ads. The corresponding output with ads removed is shown above (right).

point before the browser displays the decoded pixels. Simply put, if a user sees an image, it goes through this pipeline first.

More concretely, as shown in Figure 2.1 Percival runs in the render process of the browser engine. The render process on receiving the content of the web page proceeds to create the intermediate data structures to represent the web page. These intermediate representations include the Document Object Model (DOM), which encodes the hierarchical structure of the web page, the layout-tree, which consists of the layout information of all the elements of the web page, and the display list, which includes commands to draw the elements on the screen. If an element has an image contained within it, it needs to go through the *Image Decoding Step* before it can be rasterized. Percival is executed after the *Image Decoding Step* during the *raster* phase which helps run Percival in parallel for multiple images at a time. Images that are classified as ads are blocked from rendering. The web page with ads removed is shown in Figure 2.1 (right). The detailed design and implementation are presented in Section 2.5.

2.4.2. Alternative Possible Implementations and Advantages of Percival. One alternative to running Percival directly in the browser could have been to run Percival in the browser’s JavaScript layer via an extension. However, this would require scanning the DOM to find image elements, waiting for them to finish loading, and then screenshotting the pixels to run the detection model. The advantage of a JavaScript-based system is that it works within current browser extensibility mechanisms, but recent work has shown how attackers can evade this style of detection [122].

Ad blockers that inspect web pages based on the DOM such as Ad Highlighter [118] are prone to DOM obfuscation attacks. They assume that the elements of the DOM strictly correspond to their visual representation. For instance, an ad blocker that retrieves all `img` tags and classifies the content contained in these elements does not consider the case where a rendered image is a result of several CSS or JavaScript transformations and not the source contained in the tag. These ad blockers are also prone to resource exhaustion attacks where the publisher injects a lot of dummy elements in the DOM to overwhelm the ad blocker. Additionally, a native implementation is much faster than a browser extension implementation with the added benefit of having access to the unmodified image buffers.

2.4.3. Detection Model. Percival runs a detection model on every image loaded in the document’s main frame, a sub-document such as an `iFrame`, as well as images loaded in JavaScript to determine if the image is an ad.

Although running directly within the browser provides Percival with more control over the image rendering process, it introduces a challenge: how to run the model efficiently in a browser? The goal is to run Percival in browsers that run on laptops or even mobile phones. This requires that the model be small to be practical [108]. This design also requires that the model run directly in the image rendering pipeline, so overhead remains low. Any overhead adds latency to rendering for all images it inspects. Percival uses a modified SqueezeNet network [41] optimized for ad blocking. This results in a model size that is less than 2MB and detects ad images in 11ms per image (on a MacBook Pro’17 with IntelCore i7 3.1GHz processor).

A second challenge in using small CNNs is how to provide enough training data. In general, smaller CNNs can have suitable performance but require more training data. What is more, the labels are highly imbalanced making the training procedure even more challenging.

Gathering ad images is non-trivial; most ads are programmatically inserted into the document through `iFrames` or `JavaScript`, and so simple crawling methods that work only on the initial HTML of the document will miss most of the ad images.

To crawl ad images, other researchers [116, 122] propose screenshotting `iFrames` or `JavaScript` elements. This data collection method leads to problems with synchronizing the timing of the screenshot and when the element loads. Many screenshots end up with white-space instead of the image content. Also, this method only makes sense if the input to the classifier is the rendered content of the web page.

To address these concerns and to provide ample training data, a custom crawler was built into Blink⁵ to handle dynamically-updated data and eliminate the race condition between the browser displaying the content and the screenshot used to capture the image data. The custom-crawler fetches ad and non-ad images directly from the rendering pipeline and uses the model trained during the previous phase as a labeler. This amplifies the dataset to fine-tune the model further.

2.5. Design and Implementation

This section covers the design and implementation of the browser portion of Percival. First, the high-level design principles are discussed. This is followed by the discussion on the rendering and image handling in Blink - the rendering engine of Chromium-based browsers. Finally, the end-to-end implementation within Blink is described.

2.5.1. Design Goals. There are two main goals in the design of Percival:

Run Percival at a choke point: Advertisers can serve ad images in different formats, such as JPG, PNG, or GIF. Depending on the format of the image, an encoded frame can traverse different paths in the rendering pipeline. Also, a wide range of web constructs can cause the browser to load images, including HTML image tags, JavaScript image objects, HTML Canvas elements, or CSS background attributes. The goal is to find a single point in the browser to run Percival, such that it inspects all images, operates on pixels instead of encoded images, but does so before the user sees the pixels on the screen, enabling Percival to block ad images cleanly. **Note:** If individual pixels are drawn programmatically on canvas, Percival will not block it from rendering.

⁵Blink <http://www.chromium.org/blink> is the rendering engine used by Chromium.

In Blink, the raster task within the rendering pipeline enables Percival to inspect, and potentially block, all images. Regardless of the image format or how the browser loads it, the raster task decodes the given image into raw pixels, which it then passes to the GPU to display the content on the screen. Percival runs at this precise point to abstract different image formats and loading techniques, while still retaining the opportunity to block an image before the user sees it.

Run multiple instances of Percival in parallel: Running Percival in parallel is a natural design choice because Percival makes all image classification decisions independently based solely on the pixels of each individual image. When designing Percival, it is important to look for opportunities to exploit this natural parallelism to minimize the latency added due to the addition of the ad blocking model.

2.5.2. Rendering and Percival : Overview. Percival is integrated into Blink, the rendering engine for Google Chrome and Brave. From a high level, Blink’s primary function is to turn a web page into the appropriate GPU calls [53] to show the user the rendered content.

A web page can be thought of as a collection of HTML, CSS, and JavaScript code, which the browser fetches from the network. The rendering engine parses this code to build the DOM and layout tree, and to issue OpenGL calls via Skia, Google’s graphics library [64].

The layout tree contains the locations of the regions the DOM elements will occupy on the screen. This information together with the DOM element is encoded as a `display item`.

The browser then proceeds with rasterization, which takes the display items and turns them into bitmaps. Rasterization issues OpenGL draw calls via the Skia library to draw bitmaps. If the display list items have images in them (a common occurrence), the browser must decode these images before drawing them via Skia.

Percival intercepts the rendering process at this point, after the `Image Decode Task` and during the `Raster Task`. As the render process creates the DOM and decodes and rasterizes all image frames, these are first passed through Percival. Percival blocks the frames that are classified as ads.

2.5.3. End-to-End Implementation in Blink. Percival is implemented inside the Chromium rendering engine Blink, where it uses the functionality exposed by the Skia library. Skia uses a set of image decoding operations to turn `SkImages`, which is the internal type within Skia that

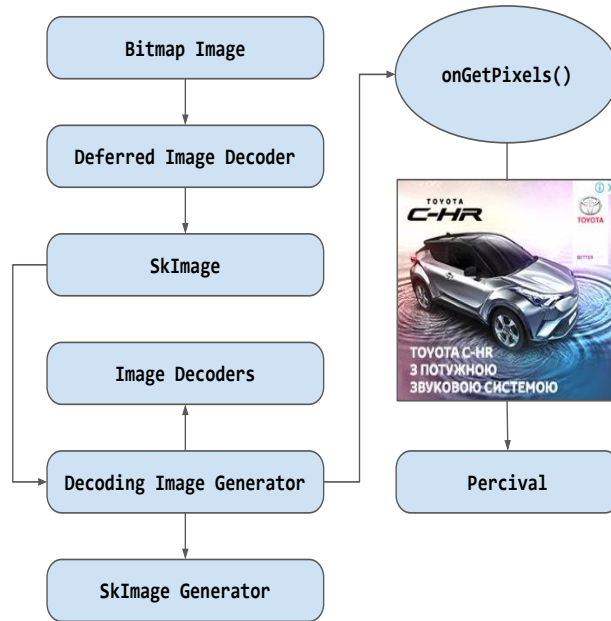


FIGURE 2.2. Percival in the image decoding pipeline. SkImage Generator allocates a bitmap and calls the `onGetPixels()` of `DecodingImageGenerator` to populate the bitmap. This bitmap is then passed to the network for classification and cleared if it contains an ad.

encapsulates images, into bitmaps. Percival reads these bitmaps and classifies their content accordingly. If Percival classifies the bitmap as an ad, it blocks it by removing its content. Otherwise, Percival lets it pass through to the next layers of the rendering process. When content is cleared, there are several ways to fill up the surrounding white-space; either collapsing it by propagating the information upwards or displaying a predefined image (user’s spirit animal) in place of the ad.

Figure 2.2 shows an overview of the Blink integration. Blink class `BitmapImage` creates an instance of `DeferredImageDecoder` which in turn instantiates a `SkImage` object for each encoded image. `SkImage` creates an instance of `DecodingImageGenerator` (blink class) which will in turn decode the image using the relevant image decoder from Blink. Note that the image hasn’t been decoded yet since chromium practices deferred image decoding.

Finally, `SkImageGenerator` allocates bitmaps corresponding to the encoded `SkImage`, and calls `onGetPixels()` of `DecodingImageGenerator` to decode the image data using the proper image decoder. This method populates the buffer (pixels) that contain decoded pixels, which are passed to Percival along with the image height, width, color channel information (`SKImageInfo`) and other

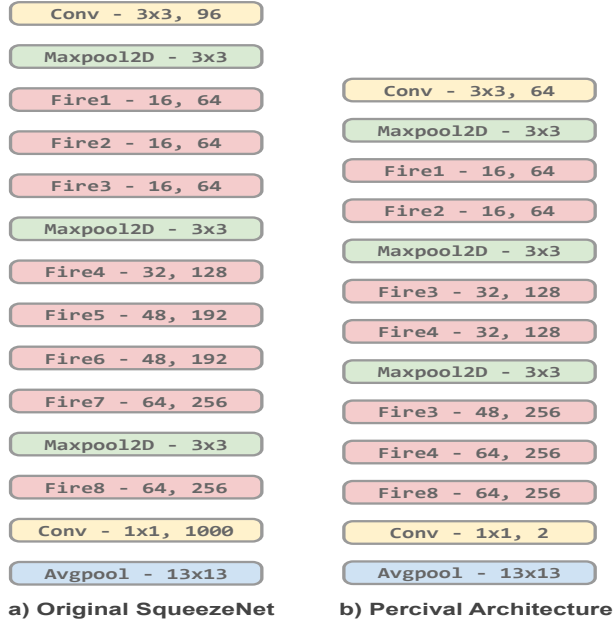


FIGURE 2.3. Original SqueezeNet (left) and Percival’s fork of SqueezeNet (right). For Conv, Maxpool2D, and Avgpool blocks $a \times b$ represents the dimensions of the filters used. For fire blocks a, b represents the number of intermediate and output channels. Extraneous blocks are removed and feature maps are downsampled at regular intervals to reduce the classification time per image.

image metadata. Percival reads the image, scales it to $224 \times 224 \times 4$ (default input size expected by SqueezeNet), creates a tensor, and passes it through the CNN. If Percival determines that the buffer contains an ad, it clears the buffer, effectively blocking the image frame.

Rasterization, image decoding, and the rest of the processing happen on a raster thread. Blink rasters on a per tile basis and each tile is like a resource that can be used by the GPU. In a typical scenario there are multiple raster threads each rasterizing different raster tasks in parallel. Percival runs in each of these worker threads after image decoding and during rasterization, which runs the model in parallel.

As opposed to Sentinel [116] and Ad Highlighter [118] the input to Percival is not the rendered version of web content; Percival takes in the Image pixels directly from the image decoding pipeline. This is important since Percival has access to unmodified image buffers and it helps prevent attacks where publishers modify content of the webpage (including iFrames) with overlaid masks (using CSS techniques) meant to fool the ad blocker classifier.

2.6. Deep Learning Pipeline

This section covers the design of Percival’s deep neural network and the corresponding training workflow.

2.6.1. Percival’s CNN Architecture. Percival casts ad detection as a traditional image classification problem, where images are fed into the model and it classifies them as either being an ad, or not an ad. CNNs are the current standard in the computer vision community for classifying images.

Due to the prohibitive size and speed of standard CNN based image classifiers, a small network, SqueezeNet [41], is used as the starting point for the in-browser model. The SqueezeNet authors show that SqueezeNet achieves comparable accuracy to much larger CNNs, like AlexNet [77], and boasts a final model size of 4.8 MB.

SqueezeNet consists of multiple *fire modules*. A *fire module* consists of a “squeeze” layer, which is a convolution layer with 1×1 filters and two “expand” convolution layers with filter sizes of 1×1 and 3×3 , respectively. Overall, the “squeeze” layer reduces the number of input channels to larger convolution filters in the pipeline.

A visual summary of Percival’s network structure is shown in Figure 2.3. The modified network consists of a convolution layer, followed by 6 *fire modules* and a final convolution layer, a global average pooling layer and a SoftMax layer. As opposed to the original SqueezeNet, feature maps are downsampled at regular intervals in the network. This helps reduce the classification time per image. Additionally, max-pooling is performed after the first convolution layer and after every two *fire modules*.

2.6.2. Training and Fine-Tuning. Before training the network in Percival, network blocks Conv 1 and Fire1 to Fire4 are initialized using the weights of a SqueezeNet model pre-trained with ImageNet [77]. The goal of using the weights from a previously-trained model is to reuse the feature extraction mechanism also known as representation learning [6].

Percival is trained with stochastic gradient descent, momentum ($\beta = 0.9$), learning rate 0.001, and batch size of 24. For training, step learning rate decay is employed and the learning rate is decayed by a multiplicative factor 0.1 after every 30 epochs. To be able to convert the model

weights to a format that can be deployed in the browser, for every MaxPool layer, the output shape was calculated by using the floor value and not ceiling as in the original model.

2.6.3. Data Acquisition. In Percival two different systems are used to collect training image data. Initially, a traditional crawler with traditional ad-blocking rules (EasyList [51]) is employed to identify ad images. Subsequently, the browser instrumentation from Percival is used to collect images, improving on some of the issues encountered with the traditional crawler.

2.6.3.1. *Crawling with EasyList.* At first, a traditional crawler matched with a traditional rule-based ad blocker is employed to identify ad content for the first dataset. In particular, to identify ad elements which could be iFrames or complex JavaScript constructs, Percival uses EasyList, which is a set of rules that identify ads based on the URL of the elements, location within the page, origin, class or id tag, and other hand-crafted characteristics known to indicate the presence of ad content.

The crawler is built using Selenium [62] for browser automation. The crawler is then used to visit Alexa top-1,000 web sites, waiting for 5 seconds on each page, and then randomly selecting 3 links and visiting them, while waiting on each page for a period of 5 seconds as before. For every visit, the crawler applies every EasyList network, CSS and exception rule.

For every element that matches an EasyList rule, the crawler takes a screenshot of the component, cropped tightly to the coordinates reported by Chromium, and then stores it as an ad sample. The non-ad samples are captured by taking screenshots of the elements that do *not* match any of the EasyList rules. Using this approach, around 22,670 images are extracted out of which 13,741 are labelled as ads, and 8,929 as non-ads. This automatic process was followed by a semi-automated post-processing step, which includes removing duplicate images, as well as manual spot-checking for misclassified images.

Eventually, 2,003 ad images and 7,432 non-ad images are identified. The drop in the number of ad images from 13,741 to 2,003 is due to a lot of duplicates and content-less (single-color) images because of the asynchrony of iFrame-loading and the timing of the screenshot. These shortcomings motivated the creation of the new crawler. To balance the positive and negative examples in the dataset so the classifier doesn't favor one class over another, the number of non ad and ad images is limited to 2,000.

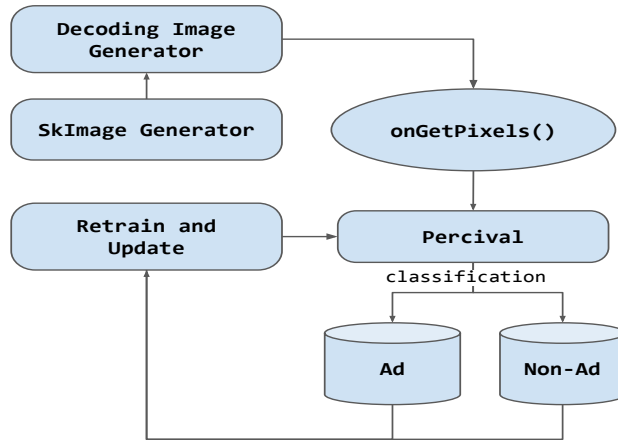


FIGURE 2.4. Crawling, labelling and re-training with Percival. Every decoded image frame is passed through Percival and it downloads the image frame into the appropriate bucket. This data is then used to re-train Percival. Every time onGetPixels() populates a buffer corresponding to the decoded image frame, the buffer is passed through the model. If the model classifies the buffer contents as an ad, the bitmap file is saved in the ad bucket else it gets saved to the non-ad bucket. This helps to capture ads that go through complex transformations(JavaScript, iFrames etc) before being displayed. The model is then re-trained with new data.

2.6.3.2. *Crawling with Percival.* While traditional crawling is good enough to bootstrap the ad classification training process, it has the fundamental disadvantage that for dynamically-updated elements, the meaningful content is often unavailable at the time of the screenshot, leading to screenshots filled with white-space.

More concretely, the *page load* event is not very reliable when it comes to loading iFrames. Oftentimes when a screenshot of the webpage is taken after the *page load* event, most of the iFrames do not appear in the screenshots. Even if the crawler waits a fixed amount of time before taking the screenshot, iFrames constantly keep on refreshing, making it difficult to capture the rendered content within the iFrame consistently.

To handle dynamically-updated data, Percival’s browser architecture is leveraged to read all image frames after the browser has decoded them, eliminating the race condition between the browser displaying the content and the screenshot used to capture the image data. This way it is guaranteed to capture all the iFrames that were rendered, independently of the time of rendering or refresh rate.

Instrumentation: Figure 2.4 shows how Percival’s browser instrumentation is used to capture image data. Each encoded image invokes an instance of `DecodingImageGenerator` inside Blink, which in turn decodes the image using the relevant image decoder (PNG, GIFs, JPG, etc.). The buffer passed to the decoder is used to store pixels in a bitmap image file, which contains exactly what the rendering engine sees. Additionally, the browser passes this decoded image to Percival, which determines whether the image contains an ad. This way, every time the browser renders an image, it is automatically stored and labeled using the initially trained network, resulting in a much cleaner dataset.

Crawling: To crawl for ad and non-ad images, the Percival-based crawler is executed with a browser automation tool called Puppeteer [55]. In each phase, the crawler visits the landing page of each Alexa top-1,000 websites, waits until `networkidle0` (when there are no network connections for at least 500 ms) or 60 seconds. This is done to give the ads enough time to load. Then the crawler finds all internal links embedded in the page. Afterwards, it visits 20 randomly selected links for each page, while waiting for a `networkidle0` event or 60 seconds time out on each request.

In each phase, the crawler downloads between 40,000 to 60,000 ad images. The images are then post processed to remove duplicates, leaving around 15-20% of the collected results as useful. The crawler runs for a total of 8 phases, retraining Percival after each stage with the data obtained from the current and all the previous crawls.

This process was spread-out in time over 4 months, repeated every 15 days for a total of 8 phases, where each phase took 5 days. The final dataset contains 63,000 unique images in total with a balanced split between ad and non-ad images.

2.7. Evaluation

2.7.1. Accuracy Against EasyList. To evaluate whether Percival can be a viable shield against ads, a comparison is conducted against the most popular crowd-sourced ad blocking list, EasyList [51], currently being used by extensions such as Adblock Plus [46], uBlock Origin [65] and Ghostery [54].

Methodology: For this experiment, Alexa top 500 news websites are crawled as opposed to Alexa top 1000 websites, which were used in the crawl for training. This is because the news websites

Images	Ads Identified	Accuracy	Precision	Recall
6,930	3466	96.76%	97.76%	95.72%

FIGURE 2.5. Summary of the results obtained by testing the dataset gathered using EasyList with Percival. Precision is defined as the percentage of images correctly classified as ads to the total number of images classified as ads. Recall is defined as the percentage of ad images correctly identified to the total number of ad images.

Ads	No-ads	Accuracy	False Positives	False Negatives	Precision	Recall
354	1,830	92.0%	68	106	78.4%	70.0%

FIGURE 2.6. Online evaluation of Facebook ads and sponsored content. False positives are the number of non-ads incorrectly blocked and false negatives are the number of ads Percival failed to block.

are an excellent source of advertisements [79] and the crawl can be completed relatively quickly. Also, Alexa top 500 news websites serves as a test domain different from the train domain used previously.

For the comparison two data sets are created: First, EasyList rules are applied to select DOM elements that potentially contain ads (iFrames, div tag elements, etc.); then, screenshots of the contents of these elements are captured. Second, resource-blocking rules from EasyList are used to label all the images of each page according to their resource URL. After crawling, the images are manually inspected (to check for any mistakes) resulting in a total of 6,930 images.

Performance: On the evaluation dataset, Percival is able to replicate the EasyList rules with accuracy 96.76%, precision 97.76% and recall 95.72%. Precision is defined as the percentage of images correctly classified as ads to the total number of images classified as ads, and recall is defined as the percentage of ad images correctly identified to the total number of ad images present in the dataset. These results are summarized in Figure 2.5.

2.7.2. Blocking Facebook Ads. Facebook obfuscates the “signatures” of ad elements (e.g. HTML classes and identifiers) used by filter lists to block ads since its business model depends on serving first-party ads. Over the years, the ad blocking community and Facebook have been playing a game of cat and mouse, where the ad blocking community identifies the ad signatures to block the ads and Facebook responds by changing the ad signatures or at times even the website [101].

Images	Ads	No-ads	False Positives	False Negatives
100	16	84	3-4	4-5

FIGURE 2.7. Average reporting of evaluation of Facebook ads and sponsored content per visit. It is assumed that each Facebook visit consists of browsing through 100 total images. Based on the numbers reported in Figure 2.6, a user will find roughly 16 and 84 ad and non-ad images, respectively. Given Percival’s recall of 70.0%, it will correctly block on average 11 - 12 of the 16 ad images, while also incorrectly blocking 4 - 5 non-ad images (given precision 78.4%).

Adblock plus was able to block all ads and sponsored content for more than a year [100]; however, as of late 2018 Facebook ads successfully manage to evade Adblock plus as the ad post code now looks identical to normal posts [107, 120] and rule-based filtering cannot detect these ads and sponsored content any more. As of 2021, Facebook does not obfuscate the content of sponsored posts and ads due to the regulations regarding misleading advertising [16, 107]. Even though this requirement favors perceptual ad blockers over traditional ones, a lot of the content on Facebook is user-created which complicates the ability to model ad and non-ad content.

In this section, the accuracy of Percival is assessed on blocking Facebook ads and sponsored content.

Methodology: To evaluate Percival’s performance on Facebook, Percival is used to browse Facebook for a period of 35 days using two non-burner accounts that have been in use for over 9 years. Every visit is a typical Facebook browsing session like browsing through the feed, visiting friends’ profiles, and different pages of interest. For desktop computers, the two most popular places to serve ads is the right-side columns and within the feed (labelled sponsored) [52].

For this work, content served in these elements is considered ad content and everything else as non-ad content. A false positive (FP) is defined as a non-ad image incorrectly blocked and a false negative (FN) is defined as an ad image Percival failed to block. For every session, these numbers are manually computed. Figure 2.6 shows the aggregate numbers from all the browsing sessions undertaken.

Results: The experiments show that Percival blocks ads on Facebook with a 92% accuracy and 78.4% and 70.0% precision and recall, respectively. Figure 2.6 shows the complete results from this experiment.

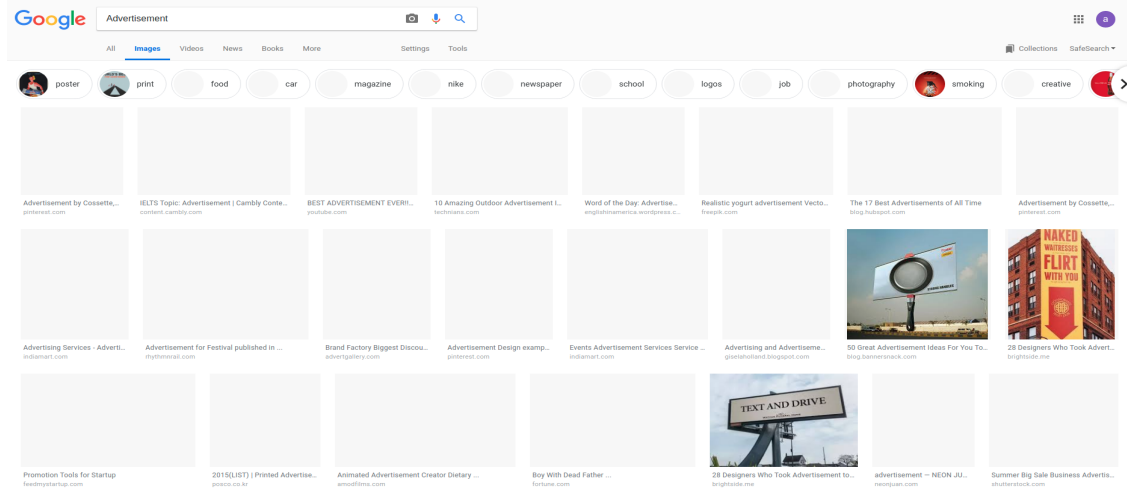


FIGURE 2.8. Search results from searching for “Advertisement” on Google images, using Percival. Percival blocks most of the images from this query since these images have high ad intent.

Even though Percival achieves the accuracy of 92%, there is a considerable number of false positives and false negatives, and as such, precision and recall are lower. The classifier always picks out the ads in the right-columns but struggles with the ads embedded in the feed. This is the source of the majority of the false negatives. False positives come from high “ad intent” user-created content, as well as content created by brand or product pages on Facebook.

Discussion: False Positives and False Negatives: To put Figure 2.6 into perspective since it might appear to have an alarming number of false positives and false negatives, it is worthwhile to consider an average scenario. If each facebook visit on average consists of browsing through 100 images, then by the above experiments (Figure 2.6), a user will find roughly 16 ad images and 84 non-ad images, out of which Percival will block on average 11 to 12 ad images (given 70.0% recall - Figure 2.6) while also blocking 3 to 4 non-ad images (given 78.4% precision - Figure 2.6). This is shown in Figure 2.7.

In addition to the above mentioned experiments which evaluate the out of box results of using Percival, a version of Percival was trained on a particular user’s ad images. The model achieved higher precision and recall of 97.25% and 88.05%, respectively.

Query	# blocked	# rendered	False Positives	False Negatives
Obama	12	88	12	0
Advertisement	96	4	0	4
Coffee	23	77	-	-
Detergent	85	15	10	6
iPhone	76	24	23	1

FIGURE 2.9. Percival blocking image search results. For each search only the first 100 images returned are considered (“-” represents cases where it is hard to determine whether the content served is an ad or non-ad). False positives are the number of non-ad images incorrectly blocked and false negatives are the number of ad images Percival failed to block.

Language	# crawled	# Ads	Accuracy	Precision	Recall
Arabic	5008	2747	81.3%	83.3%	82.5%
Spanish	2539	309	95.1%	76.8%	88.9%
French	2414	366	93.9%	77.6%	90.4%
Korean	4296	506	76.9%	54.0%	92.0%
Chinese	2094	527	80.4%	74.2%	71.5%

FIGURE 2.10. Accuracy of Percival on ads in non-English languages. The second column represents the number of images crawled, while the third column is the number of images that were identified as ads by a native speaker. The remaining columns indicate how well Percival is able to reproduce these labels.

2.7.3. Blocking Google Image Search Results. To improve the understanding of the misclassifications of Percival, Google Images were used as a way to fetch images from distributions that have high or low ad intent. For example, image results were retrieved with the query “Advertisement” and then Percival was used to classify and block these results. As can be seen in Figure 2.8, out of the top 23 images, 20 of them were successfully blocked. Additionally, for testing examples of low ad intent distribution, the query “Obama” was used. Other keywords, such as “Coffee”, “Detergent”, etc. were also searched. The detailed results are presented in Figure 2.9. As shown, Percival can identify a significant percentage of images on a highly ad-biased content.

2.7.4. Language-Agnostic Detection. Percival is also tested against images with language content different than the one it was trained on. In particular, a data set of images in Arabic, Chinese, French, Korean and Spanish is used to evaluate Percival’s language agnostic detection capability.

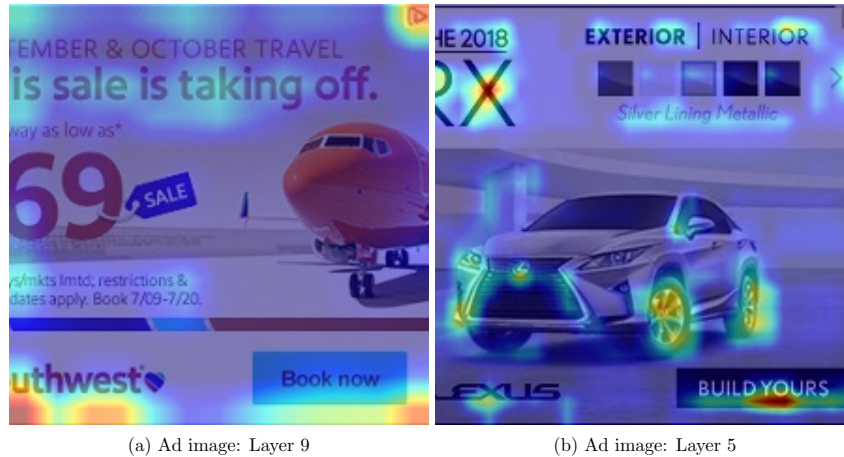


FIGURE 2.11. Saliency map of the network on a sample ad images. Each image corresponds to the output of Grad-CAM [115] for the layer in question.

Crawling: To crawl for ad and non-ad images, ExpressVPN [50] is used to VPN into major world cities where the above mentioned languages are spoken. Then, in that region, the top 10 websites as mentioned in the SimilarWeb [63] list are manually visited. The author engages with the ad-networks by clicking on ads, as well as closing the ads (icon at the top right corner of the ad) and then choosing random responses like content not relevant or ad seen multiple times. This is done to ensure ads are served from the language of the region.

Afterwards, the Percival-based crawler is executed with the browser automation tool Puppeteer [55]. The crawler visits the landing page of each top 50 SimilarWeb websites for the given region, waits until `networkidle0` (when there are no network connections for at least 500 ms) or 60 seconds. Then, the crawler finds all internal links embedded in the page. Afterwards, it visits 10 randomly selected links for each page, while waiting for a `networkidle0` event or a 60 second time out on each request. As opposed to Section 2.6.3.2, every image frame is downloaded to a single bucket.

Labeling: For each language, around 2,000–6,000 images are crawled. A native speaker of the language under consideration is hired to label the data crawled for that language, and Percival is tested with this labeled dataset to determine how accurately can it reproduce these human annotated labels. Figure 2.10 shows the detailed results from all languages Percival was tested on.

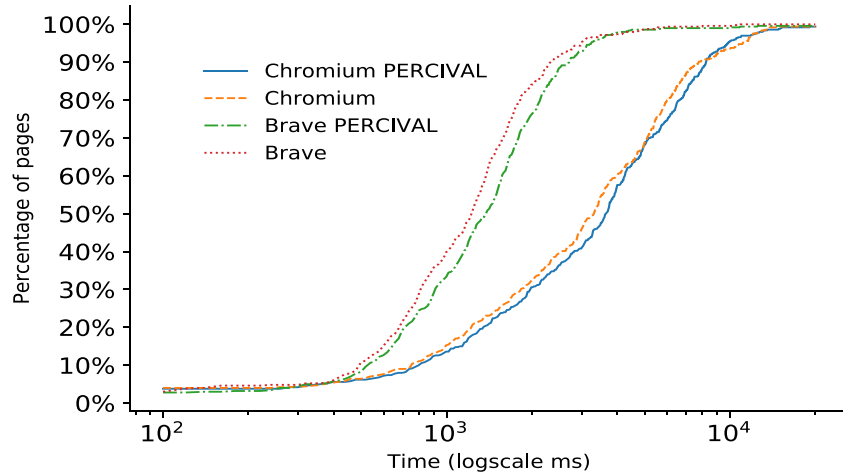


FIGURE 2.12. **Render** time evaluation in Chromium and Brave browser. The plot shows the Cumulative Distribution Function (CDF) of percentage of pages against **Render** time. A point on the curve represents the percentage of pages in the evaluation that rendered in the time given by the time axis.

Results: These experiments show that Percival can generalize to different languages with high accuracy (81.3% for Portuguese, 95.1% for Spanish, 93.9% for French) and moderately high precision and recall (83.3%, 82.5% for Arabic, 76.8%, 88.9% for Spanish, 77.6%, 90.4% for French). This illustrates the out-of-the box benefit of using Percival for languages that have much lower coverage of EasyList rules, compared to the English ones. The model does not perform as well on Korean and Chinese datasets.

2.7.5. Saliency Map of the CNN. To visualize which segments of the image are influencing the classification decision, Grad-CAM [115] network saliency mapping is used to highlight the important regions in the image that caused the prediction. As can be seen in Figure 2.11, the network is focusing on ad visual cues (*AdChoice* logo) when this is present (case (a)), also it follows the outlines of text (signifying existence of text between white space) or identifies features of the object of interest (wheels of a car).

2.7.6. Runtime Performance Evaluation. The impact of Percival-based blocking on the browser performance is evaluated next. This latency is a function of the number and complexity of the images on the page and the time the classifier takes to classify each of them. In this evaluation, the rendering time impact to classify each image *synchronously* is measured.

Baseline	Treatment	Overhead (%)	(ms)
Chromium	Chromium + Percival	4.55	178.23
Brave	Brave + Percival	19.07	281.85

FIGURE 2.13. Performance evaluation of Percival on **Render** metric. Percival leads to an increase of 178.23ms in the median render time when running in the Chromium browser and an increase of 281.85ms when running inside the Brave browser with ad blocker and shields on.

To evaluate the performance of Percival, the top 5,000 URLs from Alexa are used to test against Chromium compiled on Ubuntu Linux 16.04, with and without Percival activated. Percival is also tested in Brave, a privacy-oriented Chromium-based browser, which blocks ads using block lists by default. For each experiment, **render** time (which is defined as the difference between **domComplete** and **domLoading** events timestamps) is measured. These evaluations are conducted sequentially on the same Amazon m5.large EC2 instance to avoid interference with other processes and make the comparison fair. Also, all the experiments were conducted using **xvfb** for rendering, an in-memory display server which enabled running the tests without a display.

In the evaluation, there is an increase of 178.23ms in the median render time when running Percival in the rendering critical path of Chromium and an increase of 281.85ms when running inside the Brave browser with ad blocker and shields on. Figures 2.12 and 2.13 summarize the results.

To capture the rendering and perceptual impact better, a micro-benchmark that measures the **firstMeaningfulPaint** is created to illustrate overhead. In the new experiment, a static html page containing 100 images is constructed. Afterwards, the event **firstMeaningfulPaint** is measured with Percival classifying images synchronously and asynchronously. In synchronous classification, Percival adds on average 120ms to Chrome and 140ms to Brave. In asynchronous classification, Percival adds on average 6ms to Chrome and 3ms to Brave. Although asynchronous classification nearly eliminates overhead, it opens up the possibility of showing an image to the user that is later removed after flagging it as an ad because the rasterization of the image runs in parallel with classification in this mode of operation.

To determine why Brave with Percival is slower than Chromium with Percival, events inside the decoding process are traced using **firstMeaningfulPaint**. The results show that there is

Model	Size	False Positives	False Negatives
Sentinel [116] Clone	256 MB	0/20	5/29
ResNet [40]	242 MB	0/20	21/39
Percival	1.76 MB	2/7	3/33

FIGURE 2.14. Tramer’s [122] evaluation of various deep learning based perceptual ad blockers. The difference in the number of images used for evaluation stem from the kind of images the ad blocker is expecting. False positives are the number of non-ad images incorrectly blocked and false negatives are the number of ad images the ad blocker failed to block.

no significant deviation between the two browsers. The variance observed initially is due to the additional layers in place like Brave’s ad blocking shields.

2.7.7. Comparison With Other Deep Learning Based Ad Blockers. Recently, researchers evaluated the accuracy of three deep-learning based perceptual ad blockers including Percival [122]. They used real website data from Alexa top 10 news websites to collect data which is later manually labelled. In this evaluation, Percival outperformed models 150 times bigger than Percival in terms of recall. The results are shown in Figure 2.14.

2.8. Limitations

Deployment Concerns: This chapter’s primary focus was on deploying Percival as a native in-browser ad-blocker. However, it is important to note that this is not the only way to use this approach. Percival can also be used to build and enhance block lists for traditional ad blockers. This would require setting up a crawling infrastructure to find URLs and potentially DOM XPath expressions to block.

How to properly orchestrate crawling is not entirely clear: the simple approach of crawling a subset of the most popular sites such as those provided by Alexa will likely miss the long tail — unpopular sites that are not reached by such crawls but are reached by the long tail of users. However, such techniques can still be used to frequently update block lists automatically.

Yet a third approach is to collect URLs (and possibly XPath expressions) in the browser that are not already blocked by existing block lists, and then to crowd-source these from a variety of users. However, all these techniques come with different user privacy trade-offs. Blocking too late in the pipeline (e.g. during rendering) provides context that opens the doors for machine learning

based blocking techniques, but this sacrifices the privacy of the user, since the tracking mechanisms might have already run at this stage. On the other hand, blocking too early allows for higher privacy guarantees, but the blocking accuracy will depend on the effectiveness of the filter lists. A hybrid approach of rendering time and filter lists based blocking can help the creation of effective shields against ads that balance the trade-offs between accuracy and privacy.

Dangling Text: Testing Percival integrated into Chromium demonstrated the following limitations: Many ads consist of multiple elements, which contain images and text information layered together. Percival is positioned in the rendering engine, and therefore it has access to one image at a time. This leads to situations where the image is blocked, but the text is left dangling. Although this is rare, it can be mitigated by retraining the model with ad image frames containing just the text. Alternatively, a non-machine learning solution would be to memorize the DOM element that contains the blocked image and filter it out on consecutive page visitations. Although this might provide an unsatisfying experience to the user, it is of the benefit to the user to eventually have a good ad blocking experience, even if this is happening on a second page visit.

Small Images: Currently, images that are below 100×100 are not intercepted by Percival to reduce the processing time. This limitation can be alleviated by deferring the classification and blocking of small images to a different thread, effectively blocking *asynchronously*.

False Positives: With purely image-based blocking false-positives are a concern, though highly dependent on the context. For some websites removing a non-ad image is a big issue, whereas for others it is not.

To account for this, Percival could set different thresholds per website to pick a trade-off in the FP/FN spectrum. Of course, setting this would require user studies and detailed empirical evaluation, but is well within Percival’s capabilities. Also, Percival adds efficient CNNs to the broader ad-blocking ecosystem and as such Percival can be used in concert with other forms of ad-blocking.

2.9. Related Work

Filter lists: Popular ad blockers like Adblock Plus [46], uBlock Origin [65], and Ghostery [54] use a set of rules, called filter-list, to block resources that match a predefined crowd-sourced list

of regular expressions (from lists like EasyList and EasyPrivacy). On top of that, CSS rules are applied to prevent DOM elements that are potential containers of ads. These filter-lists are crowd-sourced and updated frequently to adjust to the non-stationary nature of the online ads [118]. For example, EasyList, the most popular filter-list, has a history of 9 years and contains more than 60,000 rules [126]. However, filter-list based solutions enable a continuous cat-and-mouse game: their maintenance cannot scale efficiently, as they depend on the human-annotator and they do not generalize to “unseen” examples.

Machine Learning Based Ad Blockers: Lately, techniques based on machine learning have started to emerge. AdGraph [67] proposes a supervised machine-learning solution on the multi-layered graph representing the interaction of the HTML, HTTP and JavaScript of the page to identify ads and trackers. This work is complementary to Percival since it exploits the latent interactions that happen inside the browser, to identify the resources of interest.

Perceptual Ad Blocking: Perceptual ad blocking is the idea of blocking ads based solely on their appearance. Storey et al. [118] uses the rendered image content to identify ads. More specifically, they use Optical Character Recognition (OCR) and fuzzy image search techniques to identify *visual cues* such as ad disclosure markers or sponsored content links. Unlike Percival, this work assumes that the ad provider is complying with the Federal Trade Commission’s guidelines on deceptive advertising [22] and is using visual cues like *AdChoices*.

Sentinel [116] proposes a solution based on convolutional neural networks (CNNs) to identify Facebook ads. This work is closer to Percival; however, their model is not deployable in mobile devices or desktop computers because of its large size (>200MB). Also, of relevance are the works of [2, 40, 134], where they use deep neural networks to identify the represented signifiers in the Ad images. This is a promising direction in semantic and perceptual ad blocking.

Adversarial attacks: In computer-vision, researchers have demonstrated attacks that can cause prediction errors by near-imperceptible perturbations of the input image. This poses risks in a wide range of applications in which computer vision is a critical component (e.g. autonomous cars, surveillance systems) [28, 93, 94, 95]. Percival is also susceptible to such attacks, since Percival’s model is deployed client side and can be used to create adversarial samples that can evade it.

Similar attacks have been demonstrated in speech to text [11], malware detection [31] and reinforcement-learning [37]. To defend from adversarial attacks, a portfolio of techniques has been proposed [13, 73, 76, 78, 87, 123]; whether these solve this open research problem remains to be seen.

2.10. Future Work

Percival opens several avenues of future work that can improve the current state of ad blocking. While Percival only focuses on running inference client-side, an efficient client-side training pipeline can facilitate a decentralized approach to ad blocking. Every user’s browsing behavior is different and unique to their browsing experience and as such a single model or a global approach of using a single block-list for all users may be sub-optimal. Client-side model generation can create custom, efficient, and privacy preserving avenues for ad blocking. It can also address the bloat of ad block-lists where less than 10% of rules are useful. Creating custom models for end-users can also aid in defense against adversarial machine learning attacks, since every user can potentially have a different copy of the model, making evasion harder. Furthermore, this technique can also be leveraged to create client-side block lists per-user and in concert with deep learning-based blocking can facilitate more robust and efficient content blocking solutions.

CHAPTER 3

Preventing credit card fraud with on-device deep learning

3.1. Motivation

In the last chapter, I showed how deep learning can improve the performance and reliability of a client-side security task (ad blocking). The focus was more on the implementation, since the problem formulation was straightforward; image-based classification. The challenge was to improve ad blocking capabilities while keeping the browser responsive.

This chapter focuses more on the problem formulation aspect, essentially answering whether it is possible to create new security solutions using client-side deep learning, particularly for the mobile platform. Given that cameras have become an integral part of the mobile app ecosystem, the rich image-data captured by them can be used to improve both the security as well as the user-experience of the application. This work leverages this capability to reduce online credit card fraud.

Just like a teller at a store physically inspects a credit card before authorizing a purchase, this work describes a system that can do the same task for online purchases. Since physical access to the card is not possible, a smartphone camera is used to inspect the card. Every time a user makes a purchase, instead of manually typing in their card details they are asked to scan their physical card with their smartphone camera. These images are then inspected by a collection of deep learning models that automatically extract all the information as well as determine whether the user's card is real or fraudulent. In essence, this work describes the process of composing multiple deep learning algorithms to convert a sequence of images into a security decision.

In the following sections, I first introduce online credit card fraud, also known as card-not-present fraud. This is followed by the problem formulation and implementation of making security decisions based on image-data. Finally, I present the detailed evaluation and real-world impact of this work.

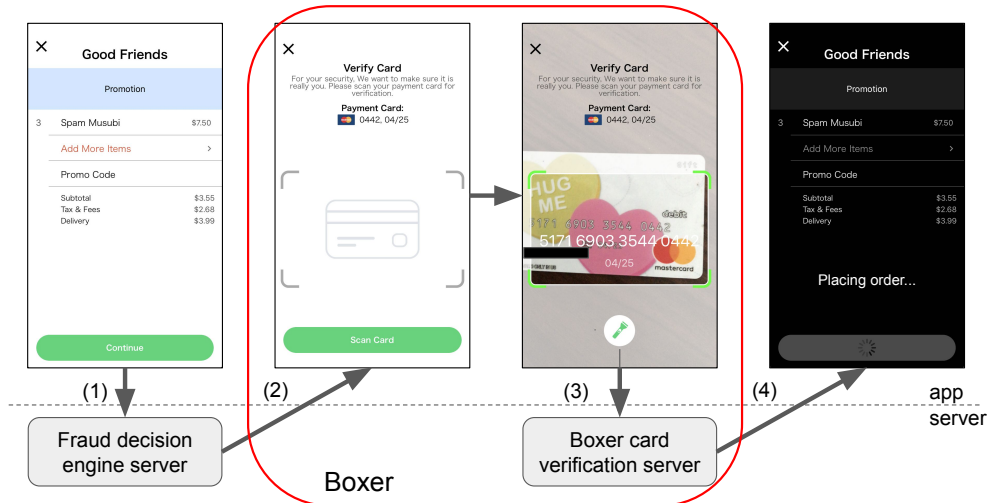


FIGURE 3.1. This figure shows how a food delivery app can use Boxer to verify a credit card for a suspicious transaction. In this example, (1) the app detects a suspicious transaction. Rather than blocking it, (2) they forward the user to Boxer’s card scanner. Boxer’s card scanner scans the user’s card, performs OCR, analyzes video frames to detect telltale signs of attacks, and collects signals from the device before (3) sending this data to Boxer’s server. Boxer’s server then decides if the card is genuine, and if it is (4) instructs the app to allow the transaction to proceed.

3.2. Introduction

Credit card card-not-present fraud is on the rise. Card-not-present fraud happens when fraudsters make purchases online or via an app with stolen credit card credentials. They enter the number, CVV, and expiration date into the app to complete the transaction, without ever needing to use the physical card itself. Industry estimates put losses from card-not-present fraud between \$6.4B to \$8.1B in 2018 [12, 72], more than twice the losses from 2015.

Two trends have pushed attackers in this direction. First, Europay, Mastercard, and Visa (EMV) chips have improved the security of traditional point-of-sale transactions where the credit card is physically present [128]. Second, financial technology (fintech) innovations have made it easy for apps to integrate payments directly, with popular apps such as Coinbase, Venmo, Lyft, Uber, Didi, Lime, and *booking.com* including payments as a core part of their user experience, providing attackers with more options to use stolen credit card numbers.

App builders are responsible for stopping card-not-present fraud themselves. When a consumer spots a suspicious charge on their credit card statement, they can dispute this charge with their credit card company. The credit card company will investigate, and if they deem the charge to be fraudulent, will file a *chargeback* with the app company. The chargeback forces the company to pay back the money from the transaction, even if they had delivered the service, and credit card companies assess apps an additional dispute fee (e.g., \$15 [119]). Thus, credit card companies financially incentivize app builders to curb this type of fraud in their apps.

One strawman technique that app builders could use to combat card-not-present fraud is to ask suspicious users to scan their physical card with the camera on their phone to prove possession of the payment method. Intuitively, scanning the card makes sense as attackers typically buy credit card numbers and not physical cards [10]. Additionally, several major apps for e-commerce, ride sharing, coupons, food delivery, and payments already use card scanning for a different purpose: as a user-friendly way to enter credit and debit card details. Thus, repurposing this basic user-experience and using card scanning as a security measure, if it can stop attacks, has the potential to easily verify legitimate users.

This chapter presents Boxer, a client-side SDK and a server (Figure 3.1) that can be used to deter credit card fraud. The first part of Boxer is a card scanner that is designed from the ground up for security. The wide deployment of card scanning suggests that it already provides a good user experience, thus the focus is on the techniques to verify that a card scanned by a user is in fact a genuine physical credit card.

The second part of Boxer is a secure counter that is based on security hardware found on modern smartphones. The secure counter is a novel abstraction where Boxer tracks events, like cards added, on a per-device basis. These events help app builders detect attacks and track devices that attackers have used previously. However, as a first-class design consideration the secure counters maintain end-user privacy.

These defensive techniques work in concert, where they are designed specifically to complement each other and to fight against card-not-present fraud. The contribution of this work, in addition to each individual defensive technique, lies in their composition as a practical defensive system to fight against a wide range of stolen card attacks.

3.3. Threat model, assumptions, and goal

In the threat model, the attacker commits credit card fraud using stolen credit card information, such as the card number (PAN), cardholder’s name, expiration date, billing address, etc. Although the card information available to the attacker is complete and accurate, the attacker does *not* have access to the physical card itself. The attacker’s goal is to authorize transactions using the stolen information.

This work considers attacks that vary across a broad range of sophistication, where the key differences lie in the technical sophistication, physical and monetary resources available, and knowledge of the banking system. This work considers attackers who are technologically savvy (e.g., can train and deploy novel machine learning algorithms) and who know how credit and debit cards work to be *sophisticated attackers*, capable of carrying out large scale automatic attacks. Other attackers use humans and real devices to carry out credit card fraud, relying on human scale (cheap labor) to attempt fraudulent transactions one at a time; these are considered to be *unsophisticated attackers*.

The goal is to stop attacks from both sophisticated and unsophisticated attackers. However, the goal is *not* to stop *all* fraudulent transactions, but rather to make credit card fraud economically infeasible across this broad spectrum of attacker sophistication

3.4. Boxer design principles and overview

This section discusses the principles that underlay the design and gives a brief overview of the technology.

The first general defensive philosophy of this work is to compose complementary defenses. Financial fraud is diverse, ranging from groups of humans carrying out attacks manually using real iPhones to attacks employing full-blown automation, bots, and machine learning. Rather than try to devise a single defense to stop them all, this work composes several complementary pieces to make an overall defensive system.

The second general defensive philosophy is to strive to never block good users. While the constraints imposed by Boxer inconveniences fraudulent users, they are designed such that they do not hamper the experience of good users.

3.4.1. Boxer design principles. This section describes the general design for scanning credit cards to verify that they are genuine. Although the focus is on scanning credit cards, these general principles can apply to similar problems, such as scanning IDs, liveness checks, or verifying utility bills. The design has five general principles that guide the implementation.

Principle 1: Scan the card to extract relevant details and check them against what the app has on record. Boxer scans the credit card number using optical character recognition (OCR, Section 3.5.5) and checks it against the card number that the app has on record for that user.

Principle 2: Inspect the card image for telltale signs of tampering. Boxer uses a visual consistency check of the card image against the card’s Bank Identification Number (BIN), which is the first six digits of the card number, and identifies the issuing bank of the card (e.g., Chase) (Section 3.5.6). For example, if a scanned card has a BIN from Chase but the model does not detect the Chase logo, then the scan is likely to be an attack.

Principle 3: Detect cards rendered on false media. Although modern machine learning and computer vision algorithms enable attackers to create modified images that are difficult to detect, the attacker still needs to render these altered images to scan them. Boxer detects the presence of a screen when it scans a card (Section 3.5.7). By detecting a screen, Boxer prevents one simple avenue for producing and scanning fake card images.

Principle 4: Associate attacker activities with items that are expensive. Boxer tracks activities and increments a secure counter when they occur on the same device (Section 3.6). This counting mechanism is important because it cuts to the core of a broad range of attack behavior: attackers will use a small set of real phones over and over to carry out attacks. By providing apps with the ability to count key events (like adding a credit card to an account) on a per device basis, it allows them to limit the damage done by large scale attacks.

Principle 5: Respect end-user privacy. Boxer puts a premium on end-user privacy by only using device identifiers that users can reset (Section 3.6) and by running the machine learning models on the client (Section 3.7).

3.4.2. Overview. Together, the card scanning system and secure counting abstraction make up Boxer, where both mechanisms complement each other to prevent damage from card-not-present

Defense	Man.	Text	Photoshop	Phys.
OCR	●	○	○	○
BIN consistency	●	●	○	○
Screen detection	●	●	●	○
Secure counting	◐	◐	◐	◐

FIGURE 3.2. Comparing defensive techniques. This table compares OCR, BIN consistency checks, screen detection, and secure counting and how they prevent attacks. The attacks are attackers entering card details manually (Man.), a text-based card image (Text), a photoshopped image scanned off a computer (Photoshop), and a physical card printed to look like a real card (Phys.). The full circle ● shows complete detection, the half circle ◐ shows detection but may let some fraud through, and the empty circle ○ shows attacker evasion.

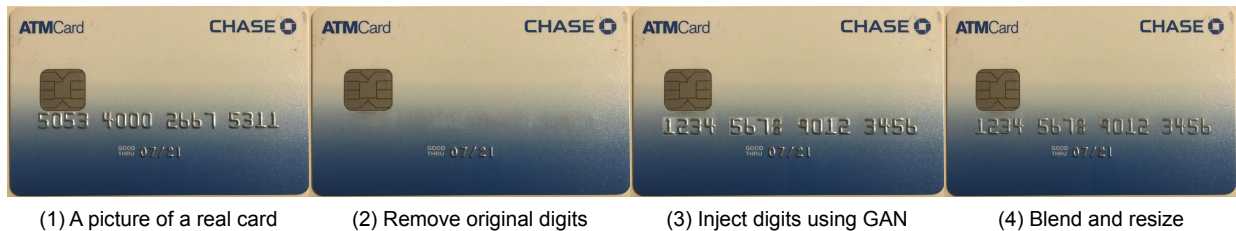


FIGURE 3.3. Fugazi’s basic process for creating fake card images. The process uses four steps and combines traditional image manipulation techniques with deep learning.

fraud (Figure 3.2). The image analysis techniques behind card scanning (OCR, BIN consistency, and screen detection) detect common ways that attackers could create fake cards with stolen card numbers. The advantage of these techniques is that when they work, they stop the attack completely. The disadvantage is that attackers who create sophisticated fake cards (e.g., physically prints cards) can evade them. On the other hand, the secure counting abstraction can effectively deter even technologically sophisticated attackers. However, it will let through a limited number of fraudulent transactions - thus, Boxer uses both card scanning and secure counting together to help make up for the shortcomings of each.

3.5. Image analysis

The purpose of Boxer’s image analysis pipeline is to verify whether a scanned image provided by a user came from a real, physical card. This verification helps distinguish between legitimate

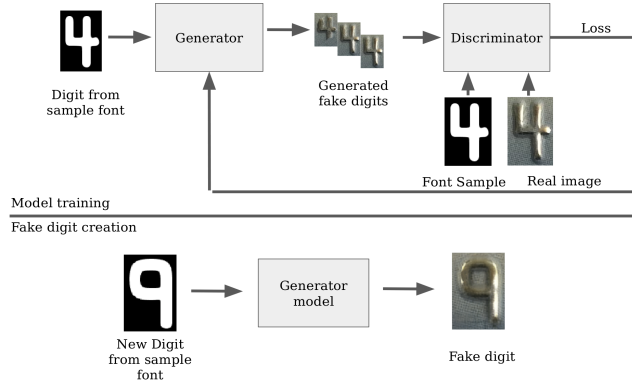


FIGURE 3.4. Fugazi digit generation process. In the above example the goal is to inject digit 9 in place of digit 4 in the original card. Fugazi takes the digit 4 in the sample font. It then trains the model to reproduce the original digit from the sample font digit. Afterwards, it use the trained model to create the textured version of digit 9 from the corresponding font digit. The model reproduces lighting, shade, and orientation close to the original digit.

and fraudulent users. A legitimate user can produce a real image by scanning their real card while an attacker, possessing only stolen credit card information, would have to doctor one. A doctored image leads to possible avenues for inconsistencies, and Boxer’s image analysis pipeline tries to spot these inconsistencies.

Although there has been work on synthetic image generation [84, 97, 121], nothing has been published about creating fake credit and debit cards. To answer if creating realistic fake card images is possible and whether existing methods can detect them, this work introduces Fugazi, an automatic system for creating realistic fake card images. The inability of the current state-of-the-art image tampering detection techniques to detect Fugazi influences the eventual design of the image analysis pipeline described later.

3.5.1. Fugazi. From a high level, Fugazi creates fake credit card images by injecting a different credit card number into an existing credit card image, automatically. Being able to create fake credit cards at scale helps devise and evaluate image-based defenses to understand their abilities and limitations.

Figure 3.3 shows Fugazi’s overall four step process for creating fake card images. (1) Fugazi starts with a picture of a real card, then (2) using hole filling and cloning computer vision algorithms removes the digits from the card, leaving only the background texture. Next (3) Fugazi uses



FIGURE 3.5. Fake credit cards that were generated using Fugazi.

a modified generative adversarial network (GAN) system pix2pix [69] to inject the digits from the new credit card number, while still respecting the lighting conditions, font wear, shape, and shading from the original card (Figure 3.4). Finally (4) Fugazi uses Poisson blending and Lanczos resampling to minimize artifacts that indicate digital tampering.

This version of Fugazi represents the fourth iteration on its design, where the informal goal was to keep working on it until the author of this work could not distinguish between fake and real cards. The first iteration used traditional computer vision algorithms and there were always clear artifacts. The second iteration used image-to-image translation deep learning systems to generate the entire card, and this approach worked well for simple textures but always produced clear visual artifacts for more complicated textures. The third iteration also used image-to-image translation, but only for the region of the card that contained the number. This technique produced great looking numbers but had a clear bounding box around the number. To remove the bounding box, yet another image-to-image translation step was used specifically to smooth out the bounding box. This third iteration was the first to produce card images that the author was unable to distinguish between fake and real, but it was too complex and took too long to create new fakes. This motivated the ultimate use of traditional vision algorithms combined with small and well defined image-to-image translation tasks. Figure 3.5 shows examples of fake cards generated by Fugazi.

3.5.2. Is machine learning sufficient to detect tampered images? While machine learning can detect images containing clear signs of forgery, researchers acknowledge that image tampering detection in general is more nuanced and requires learning richer features [136]. To answer the

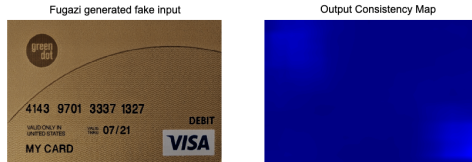


FIGURE 3.6. Fake sample and corresponding consistency map generated by the self-consistency based deep learning model [39]. Since the image is fake, and the map is uniform, it can be seen that Fugazi is able to overcome the proposed model.



FIGURE 3.7. Tampering regions determined by the Faster R-CNN based model [136] on a real image (left), and Fugazi fake (right). Not only does the model fail to detect the tampered regions in the Fugazi fake, it also mistakenly detects untampered regions on both images.

question of whether or not machine learning is sufficient to detect tampered images, in this section fakes generated by Fugazi (which can be considered as a proxy for high quality fakes) are evaluated against general image tampering detection models that achieve state-of-the-art performance on benchmark image manipulation datasets [39], [136].

3.5.2.1. *Evaluating Fugazi with state-of-the-art methods.* For evaluation, some of the existing state-of-the-art deep learning and traditional image forensics algorithms were employed to detect Fugazi generated samples.

Fugazi was first tested on self-consistency for detecting fake images [39]. This model produces a consistency map that indicates regions within the card that attackers have tampered with. Figure 3.6 shows the result of running Fugazi’s fakes through this model. As shown in the Figure, the model produces a uniform consistency map, indicating that it believes that Fugazi’s fake sample is real.

Fugazi was also evaluated against a modified Faster R-CNN model [105] proposed in recent work by Zhou, et al. [136]. Figure 3.7 shows the tampered regions detected by this model on a real image and a Fugazi image of the same texture. This figure shows that the model detects similar

regions in both the fake and real cards, suggesting that the technique is ineffective at detecting Fugazi fakes.

Additionally, Fugazi was also tested on traditional computer vision techniques, an in-house binary classifier, and an autoencoder-based anomaly detector. These techniques were also unable to detect Fugazi fakes reliably.

3.5.2.2. Further difficulties with practical deployments. All the fake image detection techniques tested try to detect tampered digital images, but in a practical deployment the user would scan the image using a phone camera. This resampling of the image runs through camera sensors and the full image processing pipeline. This layer of indirection between the tampered image and the detection algorithm has the potential to make direct detection even more difficult. So even if a user has a fake card image with possible imperfections, this layer of indirection has the potential of masking them.

3.5.3. Where to go from here? Previous sections have shown that using pure machine learning based image analysis to detect fake cards is difficult. However, attackers are *not* trying to misclassify images, they are trying to commit credit card fraud. So, in this work, the machine learning techniques are augmented with rule-based assertions to enforce checks on what passes as a valid scan. More concretely, Boxer’s image analysis pipeline uses machine learning to extract high-level features from images and enforces rules on them based on the knowledge of the design of credit cards (Section 3.5.6). Since the rules are designed to validate scans based on the design of actual credit cards, the approach serves as a form of image tampering detection. The scans blocked are those that do not conform to valid credit card designs, indicating the presence of image tampering. While this approach does not catch the most sophisticated fakes, when it works it proactively stops attacks.

The secure counting abstraction (Section 3.6) minimizes fraud from more sophisticated attacks by limiting the number of cards a user can add to a single device. This hardware-based limiting is key for technologically sophisticated attackers because to make money they need to use many stolen cards, so tying cards to relatively expensive hardware will make their attacks more expensive at scale and provide a signal that the detection system can use to identify bad actors. Boxer’s screen

detection model (Section 3.5.7) detects card images that attackers scan off screens, a common technique employed by attackers who use real phones running the app to carry out fraud.

3.5.4. Image analysis design. This section describes Boxer’s image analysis pipeline, which consists of three stages: OCR, BIN consistency and expectation check, and screen detection. Each stage collects different signals from the image and relays them to Boxer’s server. Boxer’s server enforces rules on these signals as well as those obtained from Boxer’s secure counting abstraction (Section 3.6) to determine the validity of a transaction. The stages in the image analysis pipeline along with the secure counting together realize Boxer’s general principles that are outlined in Section 3.4.1. Section 3.7 discusses the implementation.

3.5.5. Optical character recognition. OCR is how Boxer extracts a card number out of a video stream when a user scans their card. In Boxer, OCR serves as the baseline of the defense where it uses this scanned card number to match against the card number that the app has on record¹. Although unsophisticated fake cards can bypass OCR alone (Section 3.8.7), it will deter some attackers and acts as a first line of defense, feeding the card’s BIN into the more advanced image analysis stages.

Perhaps ironically, Boxer uses Fugazi fakes (Section 3.5.1) to train its OCR system. The design of Fugazi makes generating synthetic labelled data to use for training straightforward. In Boxer, OCR is cast as a special case of object detection, where a smaller more constrained version of a traditional object detector is trained, tailored specifically for credit and debit cards.

3.5.6. BIN consistency and expectation check. The BIN consistency and expectation check uses the BIN and the visual design elements of the card to check if they match. The BIN is the first six digits of the card number and identifies the issuing bank (e.g., Capital One). The goal is to train a model that can be used to verify that a card “looks” like a card corresponding to the correct BIN and the issuing bank. A card that does *not* have its BIN consistent with the visual design elements does not exist in the real world, and hence, is a telltale sign of image tampering.

¹For security reasons, the app generally has the first 6 and/or last 4 digits of the card number on record and not the full number.



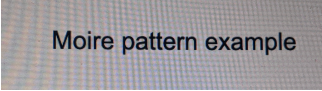
FIGURE 3.8. Output of the object detector of the BIN consistency and expectation check. The model correctly identifies issuing bank, the card network (Visa), card type, chip, name, and card number. These extracted features are correlated with the data of the card BIN to identify any inconsistencies.

The first iteration of BIN consistency was a BIN/Texture check where the model identifies issuing banks from the card appearance (texture). The key insight being that since a BIN uniquely identifies a bank, for a given BIN, there can only exist a limited number of textures.

However, from a practical perspective, it is difficult to source enough data to get realistic coverage of global credit and debit cards. First, card designs change constantly, meaning that new samples would be needed often. Given the principle of respecting end-user privacy and the sensitivity of this data, collecting card samples from users would not work. Second, the BIN database that Boxer uses contains 348,925 unique BINs worldwide [91], which from a practical perspective would make sourcing enough data from each BIN to train a model difficult.

To be able to model all possible card images given a BIN, BIN consistency check is cast as an object detection problem where the model identifies different objects and their corresponding locations on a card image. Objects such as the logo of the issuing bank, the payment network (Visa, Mastercard, etc.), and the type of card (debit or credit) are finite and persistent regardless of the background texture used to print the card. This ensures that the BIN can be uniquely identified from a combination of these objects independent of the background texture.

Boxer's BIN consistency check consists of a client-side object detector that detects objects on a card image (like the issuing bank, network, type of card) and a server side rule-aggregator that correlates the information from the features extracted by the object detector with the knowledge of card BINs to identify fraud. Figure 3.8 shows the output of Boxer's client-side object detector on a regular card image. As this figure shows, the object detector successfully detects the issuing bank, card type, payment network, name, and other features of the payment card. The Boxer SDK



Moiré pattern example

FIGURE 3.9. Moiré patterns observed when capturing a laptop screen on a mobile phone. These patterns are an inherent aliasing effect that arise from differences in spatial frequency of the laptop screen and the mobile camera.

sends an encoding of the name of each extracted feature, coordinates with respect to the card, the confidence of each detection, and the card BIN to Boxer’s server.

Boxer’s server-side rule-aggregator has built up an extensive BIN identification database and correlates the card BIN information from this database with the extracted features to identify fraud. As a simple example, if the OCR system detects the BIN of a Chase Visa debit card but the BIN consistency check detects a Bank of America logo or a Mastercard logo, Boxer flags the scan as inconsistent. Additionally, if Boxer does *not* detect a subset of the expected number of objects from a card scan, Boxer flags the scan as inconsistent.

By focusing the analysis on higher level and common features, Boxer can train an effective object detection model using less data. Also, Boxer can use the server to collect BIN and object data mappings to serve as the ground truth for improving the server-side rule-aggregator.

3.5.7. Screen detection. Boxer includes a screen detection module to detect cards scanned from computer, phone, or tablet screens. With this check, an attacker would have to physically print credit card information before scanning, which increases both the time taken and the cost required to commit fraud, particularly when done at a large scale. The general principle is to detect any false medium rendering an image, but this work focuses on screens since the author observed attackers attempt to do so in the wild (Section 3.8.4).

When cards are scanned off screens, there are distinct telltale signs and Boxer seeks to capitalize on them. These signs include screen edges or reflections (that attackers can carefully avoid) and more intrinsic signs such as Moiré patterns [99] which are much harder to circumvent.

Moiré patterns, as shown in Figure 3.9, are an aliasing effect arising from an overlay of two different patterns on top of each other, resulting in new patterns. In the context of screens, the patterns come from differences in spatial frequency of the screen containing the image and that of

the camera used to capture the image [96]. Boxer detects these signs by training a binary image classifier.

3.6. Secure counting

Boxer enables app builders to count events that it associates with hardware devices. This section describes the design of the secure counting abstraction by motivating why app builders would want to count and some of the limitations of current approaches, in addition to describing the basics of how counting works.

3.6.1. Why counting? Before describing how Boxer counts, it is important to explain *why* one would want to count events. One key observation about modern attackers is that they tend to use real hardware devices to carry out their attacks. Hardware-based mechanisms from Apple [44] and Google [56] provide app builders with solid mechanisms for ensuring that a request comes from a legitimate iOS or Android device.

Given that app builders can push attackers into using legitimate hardware devices, attackers try to repeat the same attacks using the same physical and relatively expensive hardware. App builders, knowing this, will try to count events associated with a device that indicate the existence of an attack. For example, credit card fraudsters will add many cards to accounts using the same device and will login to several different accounts from the same device. If app builders can count these events on a per-device basis, they can detect the attacks, as shown in Section 3.8.3.

Unfortunately, app builders have a difficult tradeoff that they need to make to be able to detect these events. They can either use privacy-friendly device IDs, which attackers can reset by uninstalling the app or performing a factory reset of the device. Or app builders can use persistent device IDs, which violate the privacy of their end users and Apple’s App Store policy prohibits [17, 68]. Existing industry solutions to counting also suffer from these problems. Boxer’s secure counter is novel because it respects end-user privacy while still empowering apps to maintain counts even across resets.

3.6.2. Secure counting basics. At the heart of the secure counting abstraction is Apple’s DeviceCheck abstraction [44]. DeviceCheck uses hardware-backed tokens stored on a device, which Boxer’s server uses to query two bits per device from Apple’s servers. DeviceCheck is supported

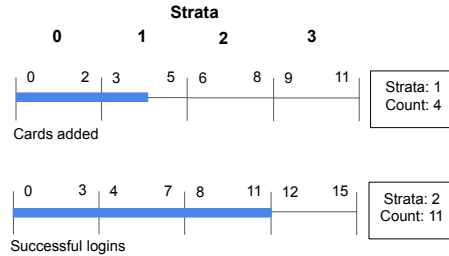


FIGURE 3.10. Counts and their associated strata. This figure shows counts for cards added and successful logins and their corresponding strata. A strata can be thought of as a bucket. In this case, the user has added 4 cards to their account and Boxer has advanced them to strata (bucket) 1 for cards added. Similarly, the user has also attempted 11 successful logins and Boxer has advanced them to strata 2 for logins.

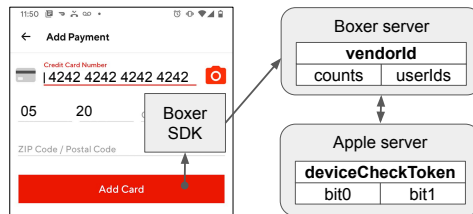


FIGURE 3.11. The architecture for the secure counting abstraction. This figure shows how Boxer updates counts after an app adds a card. The app calls into the Boxer SDK, which calls the Boxer server, where Boxer maintains a database of counts. The Boxer server manages the DeviceCheck bits by accessing Apple’s servers on behalf of the app.

on all devices running iOS 11.0 and above, which accounts for 98.3% of all iOS devices. However, two bits are not enough for app builders who want to count directly arbitrary events.

Instead of using DeviceCheck’s two bits to encode values directly, Boxer uses them to define a range of possible counts. Figure 3.10 shows a device where Boxer is tracking cards added and successful logins. For this app, the app builder expects a maximum of 11 cards added and 15 successful logins (to limit the number of accounts linked to a device), which Boxer divides into four buckets, or strata. Boxer divides the counts by four so that it can represent each of the four strata using Apple’s two hardware bits. In the example, this device has a count of four cards added and eleven successful logins, which map to strata one and two respectively.

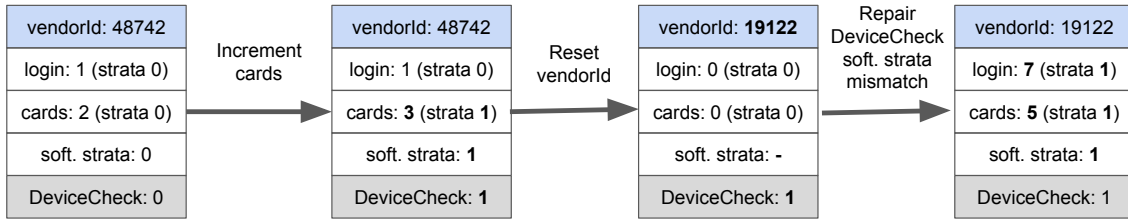


FIGURE 3.12. Example of Boxer’s secure counting system. This figure shows the counting system in four different states with three transitions between them. In this example, Boxer is counting cards added and logins, and tracking these on a per vendorId basis.

The software counts and hardware strata complement each other. Boxer uses software counts in the common case where the user maintains the same *vendorId* (Apple’s privacy-friendly deviceId abstraction [45]), but hardware strata to recover lost count values when it sees a device reset. For attackers that reset their device, the counting abstraction provides monotonically increasing count values, but for legitimate users who reset their device, by dividing the counts up into four strata Boxer limits the amount that the counts will increase on a device reset to avoid falsely flagging good users.

Figure 3.11 shows the overall architecture for how app builders use the secure counting abstraction and how Boxer keeps track of counts. From a high level, the app invokes an `increment` function in the SDK to increment the count for cards added, successful logins, or any events they want to track. The app includes an anonymous, but consistent, `userId` along with the request. The SDK then retrieves a fresh `DeviceCheck` token from the device and the `vendorId` and passes these along with the `userId` to the Boxer server. The Boxer server maintains a device database indexed via the `vendorId` to keep track of counts and `userIds` for this device. The Boxer server also accesses Apple’s servers, on behalf of the app, to query and set `DeviceCheck` bits. Apps need to register a `DeviceCheck` private key with Boxer to enable it to access Apple’s servers. Subsequently, the app can query counts from Boxer’s server.

3.6.3. Counting and inconsistencies. Figure 3.12 shows an example of how the counting system state advances as three different events occur while counting cards added and logins. The system starts with two cards added and one login counts. Boxer defines the strata for each of these

counts by dividing the maximum expected number of events by four, and each range represents a stratum. If the system increments the cards added count, it causes the count to cross a stratum as it moves the count from two to three, putting that count into stratum 1. The system defines the overall software stratum for a device as the maximum of all counter strata, so Boxer advances the software stratum to 1 and the DeviceCheck stratum to 1 as well to match the software stratum. At this point, the cards added, and login counts are in different stratum, which is acceptable as long the app continues to use the same vendorId.

If the user resets their vendorId, then subsequent requests will appear to come from a new device with all counts set to 0 and the software stratum set to an initial state (“-” in the figure). However, the DeviceCheck stratum is 1, causing an inconsistency. As a result of this inconsistency, Boxer sets *all* counts to the maximum value for their strata, which in this case is five for the cards added count and seven for the login count when they are in stratum 1. *By setting the counts to the maximum value within their strata as defined by the DeviceCheck stratum, it is guaranteed that all counts are equal to or greater than what they were before the inconsistency, thus maintaining monotonically increasing counts even after a vendorId reset.*

The rules for counting are:

- The strata for a count = $\text{floor}(\text{count} * 4 / \text{max_count})$.
- The software strata = $\text{max}(\text{strata for all counts})$.
- If DeviceCheck strata $>$ software strata, set all counts to the maximum count according to the DeviceCheck strata.
- If DeviceCheck strata $<$ software strata, set the DeviceCheck strata = software strata.

Boxer also handles counts where the maximum count is less than four and the case where the attacker moves vendorIds between attacker-controlled devices (these details are omitted for brevity).

Initialization edge case: Boxer handles the case where a user resets their device before advancing strata with the help of an uninitialized state. For a fresh device, both hardware and software strata are set to this state before the app is used for the first time, after which both advance to stratum 0. For a user who resets their device at this point, the software state goes back to being uninitialized, while the hardware state is still in stratum 0. When Boxer sees such

a configuration, it pushes the user to the maximum count of stratum 0 in software to match the hardware stratum. When the user adds their next card, both hardware and software strata will go to stratum 1, thereby ensuring monotonically increasing counts.

3.6.4. Applying stratified counting to Android. Like Apple, Google also has mechanisms in their Android systems that have the potential to serve as the backbone for Boxer’s rate limiting abstraction. In particular, Android’s Key Attestation system [56] provides a hardware-backed `uniqueId` that cycles every 30 days. However, Google restricts the use of the `uniqueId` to system level processes only, and as a third-party SDK, Boxer is unable to use it. As such, Boxer doesn’t support secure counting on Android.

3.7. Implementation

Boxer includes libraries that run on Android and iOS that app builders can put into their apps. For Android, Boxer uses the standard `jCenter` repository to deploy the library, and for iOS it uses the ubiquitous `Cocoapods` for distribution. The net result is that app builders can install these libraries using standard tools that they are almost certainly already using, with only a single line configuration change.

Boxer also includes a server portion that consumes the output of the client-side libraries to make the ultimate decision about whether a scan is genuine. The server portion runs as a Google App Engine app and uses Google’s `Cloud Datastore` as the underlying database for both iOS and Android.

The goal of the machine learning pipeline is to simultaneously pull the card number off the card to match what’s on record, look at the visual elements of a card to verify that the card design matches what is expected for card from that BIN, and detect any cards scanned off computer screens. In the current implementation, Boxer uses four different machine learning models to glean this information from a video stream: two models for OCR, one for object detection and BIN matching, and one for image classification to detect screens.

Boxer runs models client-side because it provides stronger privacy by virtue of *not* sending images of cards to the server. Also, running models client-side puts the models close to the video

stream, allowing Boxer to process more frames and with lower latency than if they were sent to a server.

3.8. Evaluation

This section seeks to answer six primary questions about Boxer and its impact in combating card-not-present fraud.

- Does Boxer recover false positives in a real deployment?
- Can Boxer’s secure counting catch real attacks?
- How does screen detection fare against real attackers?
- How viable is the BIN consistency and expectation check?
- What types of attacks are currently being employed by fraudsters, and how does Boxer stop them?
- Do existing card scanners detect fake cards?

Several international apps have already deployed Boxer, leading to over 10 million cards scanned already. Boxer is evaluated on its performance when encountering real attacks against these deployments (Sections 3.8.2, 3.8.3, 3.8.4, 3.8.6) as well as a more rigorous and controlled in-house evaluation against anticipated attacks (Sections 3.8.4, 3.8.5).

3.8.1. Handling production data. Boxer uses real data from production systems to train its defensive models and the results are reported based on real people using the apps that use Boxer. As such, for any data Boxer uses, it employs access control, stores it in an encrypted loopback device, and only uses end-to-end encrypted file systems when it is necessary to access the encrypted loopback device.

3.8.2. Does Boxer recover false positives in a real deployment? To evaluate Boxer’s ability to recover false positives, the results from an app that shipped with Boxer installed are presented. In this deployment the app allowed users flagged by its existing fraud systems to verify their cards using Boxer instead of completely blocking them.

From January 22nd, 2020 to February 5th, 2020 45 users whom the app’s systems flagged as fraudulent were sampled. Of these 45 users, 35 left without scanning. Of the ten users who did

scan, eight scanned their cards successfully and passed Boxer’s security checks, while the other two failed. Of these two users, one exceeded the *cards added* count from Boxer’s secure counting system and the other failed the screen detection check. All eight users who completed their transactions did not have chargebacks on their accounts as of February 12th, indicating that these were good users who would have otherwise been blocked (i.e., false positives).

Based on a manual analysis of the users in this dataset, the app confirmed that all 35 users who left without scanning were indeed fraudsters, as was the user caught by the secure counter. However, the user caught by screen detection appeared to be a good user. Although Boxer was unable to verify this user, they were in the same state that they would have been in without Boxer : their transaction was blocked.

Accordingly, the total number of good users in this dataset is 9, of which Boxer successfully recovers 8. Thus in this evaluation, Boxer recovers 89% of false positives without incurring any additional fraud.

3.8.3. Can Boxer’s secure counting catch real attacks? To evaluate secure counting, data from an app that shipped with Boxer in their production system is presented. They ran the system for two weeks in November 2019 in production but did *not* use the results actively to stop attacks, but rather passively recorded information. The company does have other rules that they use to block transactions, so they do actively block transactions from suspicious users. Having a passive count is advantageous because it gives the ability to inspect the data more deeply before attackers attempt to evade Boxer.

In their setup they count cards that users add to an account for each device and set the maximum count to six per month. The following evaluation consists of a random sample of ten users who hit this maximum count.

The first question to ask is whether attackers reset their devices. Boxer tracks device resets by observing an inconsistency between the software count and hardware stratum and records a timestamp for when the reset happens. In the above sample, 7/10 attackers did reset their device, presumably as a countermeasure to the other security rules that the company used. For these reset devices, the company would have been unable to count any per-device events, including cards

added, without using Boxer. Boxer was able to recover the cards added count after resets and maintain monotonically increasing counts.

The second question to ask is whether counting cards added to a device would be useful for stopping fraud. To answer this question, the userIDs from the database were examined for all users who added a card to one of the devices that hit the six-card limit and all their transactions were inspected manually.

Fraudsters used all ten devices for attacks that the company would like to prevent, and the attacks fell into three categories. First, 4/10 devices took part in traditional stolen card fraud where the users of that device added cards from a broad range of zip codes (e.g., across multiple states), indicating that the cards were coming from a list of stolen credentials. Second, 3/10 devices took part in a credit-card specific version of credential stuffing, where they added 12, 42, and 100 unique cards to a device, presumably to check if the card data they had was valid. Interestingly, the devices that they used to check cards did *not* have any transactions on them. Third, 3/10 devices took part in a scheme where they abuse the pre-authorization system.

For the ten devices that were inspected manually, there were no false positives – fraudsters used all the devices inspected for attacks. The attacks fell into three different categories, but they were all attacks.

Although the recall (recall is the fraction of the fraudulent transactions where the card limit is also exceeded) of the Boxer card added count is not known, which would be a measure of how much of the total fraud problem this signal catches, it can be confirmed that the 7/10 devices used for stolen card fraud and failed transaction fraud had charges on them that the company’s other systems had missed. As such, the company plans to start using the Boxer card added count in production to block suspicious transactions.

Finally, of the ten devices that were inspected, one device had three unique users, and another had six unique users all of whom added cards on the same device, suggesting that tracking unique logins per device could be another useful signal.

3.8.4. Can screen detection catch real attackers scanning card images from screens?

From a production dataset, 63 images are randomly sampled where attackers scanned cards rendered on screens as the validation set. Boxer’s screen detection model caught all 63 attacks. All

Accuracy	Precision	Recall
96.25%	98.25%	94.25%

FIGURE 3.13. Screen detection results on a dataset of 800 images having an even split of samples containing and not containing screens. Precision is defined as the percentage of the images correctly classified as containing screens to the total number of image classified as containing screens. Recall is defined as the percentage of the total number of images containing screens correctly classified by the screen detector.



FIGURE 3.14. A BIN inconsistent fake card image caught by the BIN consistency and expectation check. The card shown in the image starts with a 4 and should thus have the Visa logo. The BIN (first 6 digits) of the card is also not from Chase, and thus, a Chase logo should not be present. The BIN check detects both inconsistencies, showing that it would flag such a card as fake.

these images, however, clearly showed the edge of the screen that the attacker was using to display the card.

Since a careful attacker can avoid screen edges while scanning, a more extensive internal evaluation is conducted. 400 images of different credit cards captured across multiple screens were collected manually. These 400 samples had credit cards displayed on multiple screens, and were captured using multiple devices, showing the screen edge in some cases, and not showing in others. These were combined with 400 images clearly not having screens obtained from the same mobile payment app to build a test set of 800 images for evaluation. Of these 800 images, the screen detection model was able to correctly label 770 images, giving an accuracy of 96.25%. Screen detection incorrectly labeled only 7 out of 400 images that did not contain screens, thereby resulting in a precision of 98.25%, and missed 23 out of 400 images that contained screens resulting in a recall of 94.24%. Figure 3.13 summarizes these results.

In Boxer, the screen detector is run on three frames for each scan, so there are multiple opportunities to detect a screen and some flexibility in balancing false positives and false negatives.



FIGURE 3.15. Samples for the case study. This figure shows the original card and four different fake versions of the real card. The fake cards include a Google doc with the number in the middle scanned off a computer screen, a Fugazi fake where the BIN mismatches and scanned off a phone, the real card scanned off a phone, and a Fugazi fake that was printed out using a high quality printer and plastic. All of these fake cards were added successfully to all the apps that were tested using their card scanner.

3.8.5. How viable is the BIN consistency and expectation check? The BIN consistency check is designed to catch attackers who create card images that look like cards, perhaps using a stock card image, but do not match what is expected for a card from that BIN. Since this style of attack has not been seen in the wild so far, this defense can be considered a proactive defense that anticipates future attacks. Thus, it is evaluated on valid cards to check for false positives and see if it can detect a purposely crafted BIN inconsistent Fugazi fake.

A validation dataset containing 2000 legitimate production credit card images is used. On evaluation, the BIN check had a false positive rate of 0 on this dataset, showing that it will not affect the experience of good users.

A BIN inconsistent fake card was created using Fugazi. This card has the BIN of a GreenDot card, but it is rendered in the form of a Chase card. While existing apps were unable to detect this fake card (Section 3.8.7), the BIN check correctly classifies this as a BIN inconsistent image by detecting the presence of a Chase logo as shown in Figure 3.14.

3.8.6. What types of attacks are currently being employed by fraudsters, and how does Boxer stop them? Following are a random sample of attacks observed in the wild:

- 23% were users who did not produce a picture of a card in their scan.
- 74% were users whose scanned cards did not match the card that the app had on record for these users.
- 3% were users who scanned card images rendered on mobile and computer screens.
- One user with a clearly photoshopped card.

In this dataset, the overwhelming majority of attacks were from users who scanned something other than a card and users who scanned a card that did not match what the app had on record for the user. Boxer's OCR stops the users who were unable to produce a card image and those who produced card images that did not match the card number on record. A few users scanned cards rendered on mobile devices or monitors, which the screen detector detects. There was a single example of a user using a photoshopped card. While Boxer was unable to detect this card, it had an incorrect font and it is expected that future systems will be able to detect this style of attack.

3.8.7. Do existing card scanners detect fake cards? Many apps include card scanning as a better user experience for adding cards when compared to entering the card details manually into an app. However, this extra data from the scan also presents an opportunity to detect signs of attacks. To test if existing apps use this data, several fake cards were generated and added to a ride sharing app, a food deliver app, an e-commerce app, and a security SDK using their card scanning features. The ethical considerations of these attacks is discussed in Section 3.8.7.1.

Figure 3.15 shows both the real card and the fake cards that are used for this experiment. The original card is a GreenDot card, and the fakes include a Google Doc with the card details typed on it, a Fugazi fake with a Chase card containing the GreenDot number on a phone screen, the original GreenDot card on a phone screen, and a physical card printed on plastic of the Chase card. The physical card was printed at a local print shop, and it cost \$35 per card.

The fake cards were added to all the apps successfully, suggesting that these apps are not looking at the card for signs of tampering. Fraud systems are complicated, and cards were added to existing accounts for the ride-sharing app and the e-commerce app. Thus, it is possible that

even if they had detected signs of abuse, they may have let it pass due to the good standing of the accounts that added the cards.

However, with the food delivery app, a new account was created and the cards added were fake, a classic pattern for financial fraud. After adding the card, a purchase was made, showing that this card bypassed all fraud checks. The security SDK that was evaluated is an anti-fraud library as opposed to an app itself, but they claim to provide confidence that the user possesses the physical card, which the experiment shows to be false.

3.8.7.1. *Ethical considerations.* In the experiments, fake cards were added to accounts on real apps. In one experiment, a purchase was made using the fake card. However, the credit card number that was used in this experiment was from a pre-paid debit card that the person making the purchase owned. Thus, the merchant was appropriately paid for the food delivered.

3.9. Related work

In addition to the papers already mentioned, this chapter is related to other works in detection of physical and cyber payment card fraud or card not present fraud, digital image forensics, image synthesis and multi-factor authentication.

Payment cards are vulnerable to skimming attacks where data is stolen and sold online [14]. Researchers did a study of card skimmer technology and used it to develop a card skimmer detector [113], which exploits the physical constraints required for a card skimmer to work properly. Scaife et al. [112] surveyed various gas pump point-of-sale skimmer detection techniques like Bluetooth based skimmer detection mobile apps. The authors reverse engineer all the available apps to determine the common skimmer detection characteristics. In another work, Nishant et al. [8] evaluate the effectiveness of using Bluetooth scans to detect card skimmers. They demonstrate several discriminating signals between regular Bluetooth devices and card skimmers which can be used to detect the skimmers.

Researchers have also bolstered the security of gift cards, an increasingly popular payment method considerably different in design from both credit and debit cards [114]. They devise a reliable method to detect counterfeit gift cards by picking up high variance in the bit lengths in the encoding of information on counterfeit gift cards.

Stapleton and Poore explain in detail the standards maintained by the Payment Card Industry (PCI) Security Standards Council (SSC) to protect credit card holder data [117]. Researchers have shown how BIN can be used in conjunction with the IP address for a BIN/IP check [47] to identify fraud. The device location is correlated with the country of issuance of the bank to identify fraud.

Data mining has been used to propose solutions to card not present fraud. Akhilomen used features like geolocation of the transaction, email address or phone number used in the transaction, goods purchased, and shipping address to train a neural network based fraud detection anomaly system [3]. More recently, Zanin proposed a combination of data mining and parenclitic network analysis to ascertain the validity of credit card transactions [135].

The area of digital image forensics looks at the broad area of detecting fake images. Farid outlines this area in a survey of the topic [21]. Techniques, such as cloning [24] and JPEG quantization [20], use the fact that the underlying statistics of any digitally forged alteration would not match that of a real image, although they look indistinguishable to a human being. Such techniques have also been incorporated into deep learning algorithms, where the model learns the distribution of only real images and uses anomaly detection to detect fake images [133].

Detecting screens has also been explored previously. Patel et al. seek to use Moiré patterns to detect replay attacks aiming to evade facial recognition systems [96]. More recently, Gracia and Queiroz also use Moiré pattern analysis to detect replay attacks [25].

Multi-factor authentication focuses on how to use additional mechanisms to prove the identity of the individual interacting with an app. Recently researchers have proposed novel factors to empower people to authenticate explicitly via voice recognition [5, 125]. Researchers have also proposed a number of systems to enable login systems to verify additional factors implicitly [74, 80, 88]. Finally, researchers have shown how to use statistical methods to be smart about when to even ask for additional factors [23].

CHAPTER 4

Wide scale measurement study with on-device deep learning

4.1. Motivation

To protect end-user privacy, Boxer (presented in the previous chapter) runs the deep learning inference that directly interfaces with user-data, entirely client side. Owing to the differences in sensor quality and compute capabilities, there is a stark difference in the performance of running image processing machine learning tasks on low-end and high-end phones. Thus, Boxer has to deal with orders of magnitude difference in machine learning capabilities across various devices.

For high-end iOS devices, third party developers can use hardware accelerators with vendor specific APIs such as CoreML, which uses Metal (Apple's GPU programming language) to leverage the GPU, CPU and Neural Engines to provide efficient neural network inference. For low-end iOS devices, however, third party developers have to fall back to running inference on the CPU, which is significantly slower.

For Android devices, third party developers find optimizing for a particular chipset difficult since there are over 2000 unique System on Chips (SoCs) in distribution. While there are vendors such as Qualcomm, HiSilicon etc., that provide proprietary Software Development Kits (SDKs) for accessing Digital Signal Processors (DSPs), GPUs, etc. on different mobile platforms, all of these SDKs provide access to only some chipsets and are incompatible with each other. Phone vendors such as Google and Samsung provide specific AI chips mainly used by the cameras for efficient image processing - however, no SDKs or drivers are provided for those chips and as such they can't be used by third party developers. Hardware acceleration for deep learning algorithms continues to be a challenge on Android devices, and this is the primary reason that nearly all mobile inference is run on the CPU which forms the common denominator for the vast majority of Android phones.

At best, the result of the range in performance inconveniences users by making them wait longer to verify their cards, and at worst, prevents users from successful verification. In either case, users

attempting to verify themselves are being penalized simply for not possessing a high end phone. This chapter demonstrates the impact of these differences first-hand in the public deployment of Boxer and serves to quantify the impact of end-user compute capabilities on the performance and reliability of Boxer’s machine learning pipeline.

4.2. Measurement study

This section presents the first large-scale measurement study of a security challenge using deep learning on mobile devices. It describes the practical characteristics and limitations of credit card scanning using real apps running on end-user devices with real people and credit cards, and all the idiosyncrasies inherent in large-scale software with live deployments.

This study is the first of its kind and has implications for deep learning engineers, app developers, and hardware vendors. The closest to this study in terms of scale is presented by Ignatov et. al [43] - however, their study is limited to only 10,000 Android devices and runs pre-defined images through pre-trained models loaded on each device to benchmark the hardware.

This is an in-field correlation study and represents a realistic usage scenario for end-users since it benchmarks the usage of a deep learning driven application where the user, the phone sensor, image processing, ambient lighting, device surface temperature, the compute capability of the device and other production variables determine the performance of the system. In this study, while the end-user privacy is protected by limiting the amount and nature of the statistics that are recorded, the metrics have enough fidelity that they can inform the end-design.

4.2.1. Measurement study goals and questions. The high-level goal is to understand the practical performance and limitations of camera-based mobile security challenges in real-world conditions. The study is performed using Boxer, a widely deployed credit card scanning system described in the previous chapter, and its success rate is measured as the primary metric for evaluation. The success rate is defined as the ratio of the number of users where the Boxer scanner successfully extracted the card number to the total number of users using the scanner. To understand the performance and limitations, the correlation study is focused on three primary questions:

How does the speed of machine learning (ML) predictions influence end-to-end metrics for success? The research community and industry have put a heavy emphasis on performance for ML

predictions through machine learning models designed specifically for mobile devices [35,36,42,110] and hardware support for fast inference [4,30]. This study measures the impact of these efforts on high-level metrics for success.

How widely do the ML capabilities on modern phones vary in the field? This study measures the range of ML capabilities one is likely to see in practice. By understanding the range of capabilities, one can anticipate the performance differences for security challenges in realistic settings. This study also quantifies the number of devices that are unable to run Boxer ML effectively, which for a security check blocks the user.

How long are people willing to wait when they try to scan documents with their phone? As there are many forms of scanning documents that apps use for security checks, understanding how long people are willing to wait as they try to scan informs the overall design of a security check. Security check designers will benefit from knowing how long they have to capture relevant information before someone gives up.

4.2.2. Measurement Platform. To measure Boxer’s performance, Boxer’s SDK is instrumented and made available to third-party app developers. Afterwards, the success rate for the users of their live production apps is measured. This study reports the results from anonymous statistics sent by 496 apps that deployed and ran the instrumented library from July 2019 to late November 2020.

4.2.3. Testbed. The instrumented Android SDK ran on a total of 329,272 Android devices spanning a total of 611 Android device types. This included 168,658 Samsung devices spanning 281 Samsung device types, 49,329 Huawei devices spanning 91 Huawei device types, 80,351 Xiaomi devices spanning 64 Xiaomi device types, 5,464 LG devices spanning 63 LG device types, 2,939 Google devices spanning 11 Google device types, 2,501 Motorola devices spanning 27 Motorola device types, 2,560 OnePlus devices spanning 18 OnePlus device types and tail of 17,470 devices, spanning 56 device types and 23 vendors. The instrumented iOS SDK ran on a total of 3,175,912 iOS devices spanning a total of 27 iOS device types.

4.2.4. Task. Boxer uses a two-stage OCR similar to Deep Text Spotter [9], consisting of a detection phase to detect groups of digits in credit card images and a recognition phase to extract

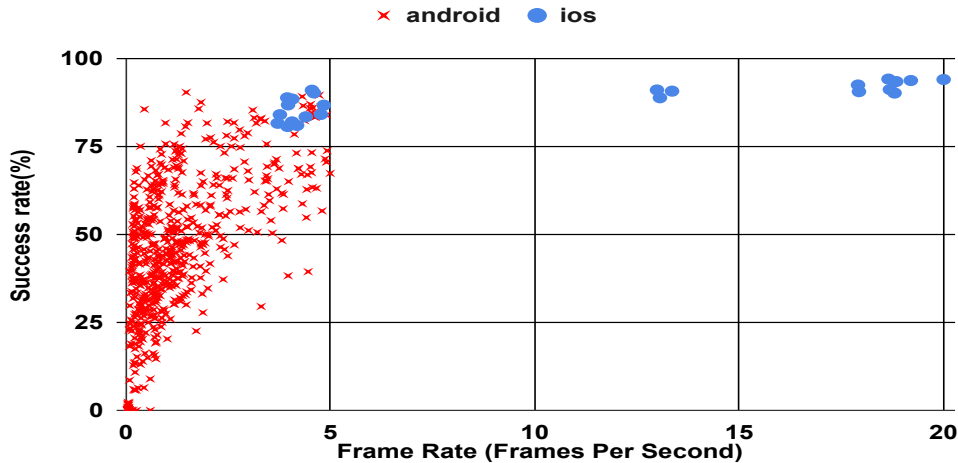


FIGURE 4.1. Boxer OCR success rate vs frame rate on Android and iOS. Each point is the average success rate and frame rate for a specific device type. This figure shows that when using the same machine learning model, end-to-end success rates drop off as the frame rate declines. It can also be seen that the same machine learning models and inference pipeline exhibit different performance characteristics on Android and iOS.

individual digits from these groups. The input image from the camera is processed and sent to the detection model, which outputs a set of proposals. These proposals are then passed to the second model that outputs the digits contained in the proposals. Afterwards, a set of post-processing methods are run to aggregate the information.

Both models of Boxer OCR use a modified version of MobileNet [36], where the detection model occupies 1.75MB and the recognition model occupies 1.19MB on disk. The detection model processes an input image of size 480x302x3 and generates a total of 1,734 proposals. It has a total of 910,379 parameters of which 901,379 are trainable. There are 16 2D convolution blocks and 10 Depthwise convolution blocks, each followed by a batch-norm and an activation layer.

The recognition model processes input images of size 36x80x3, each corresponding to the proposal generated by the detection model, and generates a feature map of 17x11 for each proposal. It has a total of 618,540 parameters of which 611,754 are trainable. There are 14 2D convolution blocks and 8 Depthwise convolution blocks, each followed by a batch-norm and an activation layer.

For inference, the iOS SDK uses the vendor specific CoreML, which runs the models on the CPU, the GPU and the Neural engine depending upon the availability and usage at any time. The

Android FPS	Count	Success rate
< 1 FPS	146,890 (44.61%)	31.87%
1–2 FPS	97,798 (29.70%)	49.97%
>= 2 FPS	84,584 (25.68%)	68.72%

FIGURE 4.2. Success rates for Android devices running Boxer by the frame rate. It can be seen that a significant portion of devices operate at frame rates less than 1 FPS.

Android SDK uses a generic interpreter TFlite, where the inference primarily runs on the CPU. The models are quantized using 16-bit floating point weights.

The instrumented Boxer SDK prompts users to scan their credit cards. When invoked, it starts the camera and prompts users to place their card in the center of the viewport. The OCR processes the frames obtained from the camera and attempts to extract the card number and the expiry from the card. Upon success, the card number and the expiry are displayed to the user and the SDK sends the scan statistics to the server. If the OCR is unable to extract the number, the flow doesn't time-out; instead, it lets the user cancel the scan which provides an additional user-level metric that can guide a new design, since this time can inform how long users wait on average when scanning entities with their phones. After a user cancels the scan, they can either try again or manually type in the number to proceed.

4.2.5. Results.

4.2.5.1. *Key Performance Metrics.* **Success rate:** The success rate is defined as the ratio of the number of users where the scanner successfully extracted the card number to the total number of users using the scanner.

Frame rate: The frame rate is defined as the number of frames from the camera processed by the OCR pipeline (detection and recognition) per second.

Figure 4.1 shows the variation in success rate against the frame rate for different devices and Figure 4.2 shows the success rates for Android devices running Boxer grouped by frame rate. Data from Figure 4.1 and Figure 4.2 suggests:

- Both the frame rate and the success rate are higher on iOS than on Android when using the same machine learning models and same inference pipeline.

Platform	Count	Avg Success Rate	Avg FPS	Avg Scan Duration (s)
Android	329,272	46.72%	1.303	14.45
iOS	3,175,912	88.60%	10.00	10.02

FIGURE 4.3. Aggregate results of Boxer on Android and iOS. It can be seen that both average success and frame rates are significantly lower on Android than on iOS. The average scan duration is the average time for which a user had to scan their card.

Platform	Count	Avg FPS	Avg Scan Duration (s)
Android	175,435	1.00	16.20
iOS	361,924	9.28	20.73

FIGURE 4.4. Failure cases of Boxer on Android and iOS. The number of iOS failures is higher than Android due to the larger representation of iOS devices in the deployment. The average scan duration is the average time for which a user waits before cancelling the scan.

- Boxer is ineffective on Android devices when the frame rate is less than 1 FPS. These devices make up 44.61% of the Android devices in the study and achieve a success rate of 31.87% compared to 49.97% for devices that run at 1-2 FPS and 68.72% for devices that run at 2 FPS or higher.

Figure 4.3 shows aggregate results for iOS and Android. While the success rate for iOS is 88.60%, the success rate for Android is much lower at 46.72%. It is important to note here that the average success rate on iOS is somewhat optimistic since devices like iPhone 6 and below were omitted from the deployment since these devices are unable to run Boxer’s ML models at a frame rate that is practical for a public deployment.

4.2.5.2. *Further analysis of failure cases.* This study also measures how long users are willing to wait before giving up, in case the card scanner is unable to extract the information from the card. Knowing the time that people are willing to wait while scanning informs decisions when designing the system and trading off scan times vs accuracy and fraud signal fidelity.

From the real-world deployment of 3,505,184 scans, there were 537,359 failed attempts where users gave up on trying to scan their card. On average, Android users waited 16.20s and iOS users waited 20.73s to scan their cards before giving up (Figure 4.4).

4.2.6. Context for the results. For the fraud challenge, Boxer uses OCR to extract and verify the card number that the app has on record for any user. Thus, anyone who is unable to scan their card number will be unable to pass the fraud challenge. Additionally, OCR is the first model in the Boxer pipeline and is used to extract data like the first six digits (BIN), which is then correlated with other features like the credit card design to determine fraud. However, if the first model in the pipeline fails to run, the device is implicitly denied the service.

Boxer solves the business problem that it intends to solve, since it runs OCR successfully on 84.7% of the devices overall. However, the success rate on devices that run at a rate of less than 1 FPS is mere 31.87%, and these devices make up 44.61% of the Android devices measured in this study, introducing a potential ethical conundrum by blocking users solely because they have an inexpensive device. In this study however, Boxer was used only for improving the user experience and not for blocking suspicious users. On the devices where it was unable to extract the card number, users had to manually enter their card information to proceed.

Ethical anti-fraud systems for mobile payments

5.1. Motivation

In chapter 3, I presented Boxer, a system for scanning credit cards to prove that the user possesses the genuine physical card. Boxer is an example of a user-facing security challenge, a form of step-up authentication employed by apps to add security. Some other examples of this style of verification are Coinbase’s ID verification where they ask users to scan an ID to prove who they are in the real world [49], and Lime’s Access program that allows people of a low socioeconomic status to scan IDs and utility bills to prove that they qualify for discounted rental fees [59].

These challenges have the potential to skirt the difficult ethical issues that apps face with security decisions in their apps. In a typical app, there will be an algorithm that predicts whether a user or a transaction is suspicious. These algorithms could potentially rely on features that unfairly influence its decision, such as a zip code. To reduce the impact of mistakes by their algorithms, apps can use user-centric security measures in lieu of suspending users or blocking transactions. This technique allows users that the algorithm blocks incorrectly to verify themselves or their payment methods automatically. Thus, even if their algorithm has bias [34], challenges provide an avenue for making sure that everyone can access the app.

Unfortunately, challenges open a new set of ethical conundrums. Apps that want to respect end-user privacy and run their challenges via compute intensive machine learning models on the device will have to cope with the 1-4 orders of magnitude difference in capabilities on the devices that they will see in practice, as was shown in the previous chapter. Apps that opt for predictable machine learning (ML) performance by streaming data to a server and running their ML there will have to deal with a 1000x difference in bandwidth between 3G and 5G networks [127], and the people who use it may have to pay for that bandwidth directly. Security challenges must deal with these subtleties of practical deployments or else they run the risk of blocking users unethically.

The most dangerous aspect of the ethical implications inherent with security challenges is that they solve an app’s business problem but have the potential to still make compromises on the experience of users of a low socioeconomic status. One example of this trade off is with Lime’s Access program [59], which allows users from low-income households to get reduced rates on Lime rentals by proving that they qualify for the program by scanning welfare documents or utility bills. These documents contain personal information that typical Lime users do not have to provide, and Lime does not process these documents themselves, they use a third party for this service [58]. While this program is commendable and Lime should be applauded for implementing it, it would be better if Lime could prove that these documents are genuine without needing to send sensitive information to a third-party server.

Another example of this trade off is Boxer, presented in chapter 3. Based on the data obtained from the measurement study (chapter 4), it fails on 68.13% of Android devices that run its machine learning (ML) algorithms at less than one Frame Per Second (FPS). Slower ML inference corresponds to lower frame rates and thus fewer inputs that the system processes for verification. Like Lime’s Access program, apps that use Boxer solve the business problem – only 4.19% of the total devices identified in the measurement study are Android devices that run Boxer’s ML at less than one FPS. By using Boxer, apps recover most of the people that their security systems flag incorrectly. However, by blocking devices that are unable to run their ML models fast enough, it runs the risk of denying access to potentially at-risk populations simply because they have an old or an inexpensive device.

The inability to run challenges on resource-constrained devices introduces a new bias that the existing formulations of machine learning fairness [7, 15, 34, 71, 75] are ill-equipped to solve. Existing formulations of machine learning fairness modify either the decision engine or the feature set corresponding to an individual of a protected group to ensure that protected attributes (e.g., race) do not affect the outcome. However, being unable to run models on resource-constrained devices robs the decision engine of the inputs it needs to make a decision in the first place. Although the decision engine could randomly pass individuals whose devices are low end or randomly block otherwise good users to provide a notion of fairness, both degrade the performance of the overall

system since they weaken the ability to distinguish between legitimate and fraudulent users. No algorithmic or theoretical notion of fairness can account for this lack of data.

This dissertation’s position is that ethical security challenges should run effectively on resource-constrained devices. In line with the aforementioned objective, this chapter presents Daredevil, a system for running complex client-side ML models for security on the full range of devices one is likely to see in practice today. Daredevil’s design includes decomposing machine learning tasks for redundancy and efficiency, streamlining individual tasks for improved performance, and exploiting task and data parallelism.

5.2. Overview

This chapter introduces Daredevil, a new system that is designed to realize ethical deep learning powered user-centric security challenges, with the goal of providing equal access to all users. Although Daredevil is built to reduce card-not-present credit card fraud, the insights gained from it can also be applied to design other end-user security challenges.

To provide equal access, Daredevil must be fast (even on resource-constrained devices that lack hardware acceleration for machine learning), Daredevil must respect end user privacy, and be accurate to avoid incorrectly flagging otherwise good users as being fraudulent.

5.2.1. Threat model. In the threat model, the goal is to reduce financial fraud while ensuring that all legitimate users can pass the challenge. The focus is on challenges that apps can use to verify that people possess a genuine credit card. It is assumed that the attacker has stolen credit card credentials (e.g., the card number and billing ZIP code), but does *not* possess the real credit card.

Daredevil’s machine learning models run client side, where it processes credit card images on the device before passing a distilled summary of the machine learning output to the server, and the server makes the ultimate decision about if a scan is genuine. As the models run client side, Daredevil is susceptible to attackers who tamper with the app, the video stream, or its machine learning models. Although there are some measures in place to assess the integrity of the client-side software (e.g., DeviceCheck [44] on iOS and SafetyNet [57] on Android), this type of

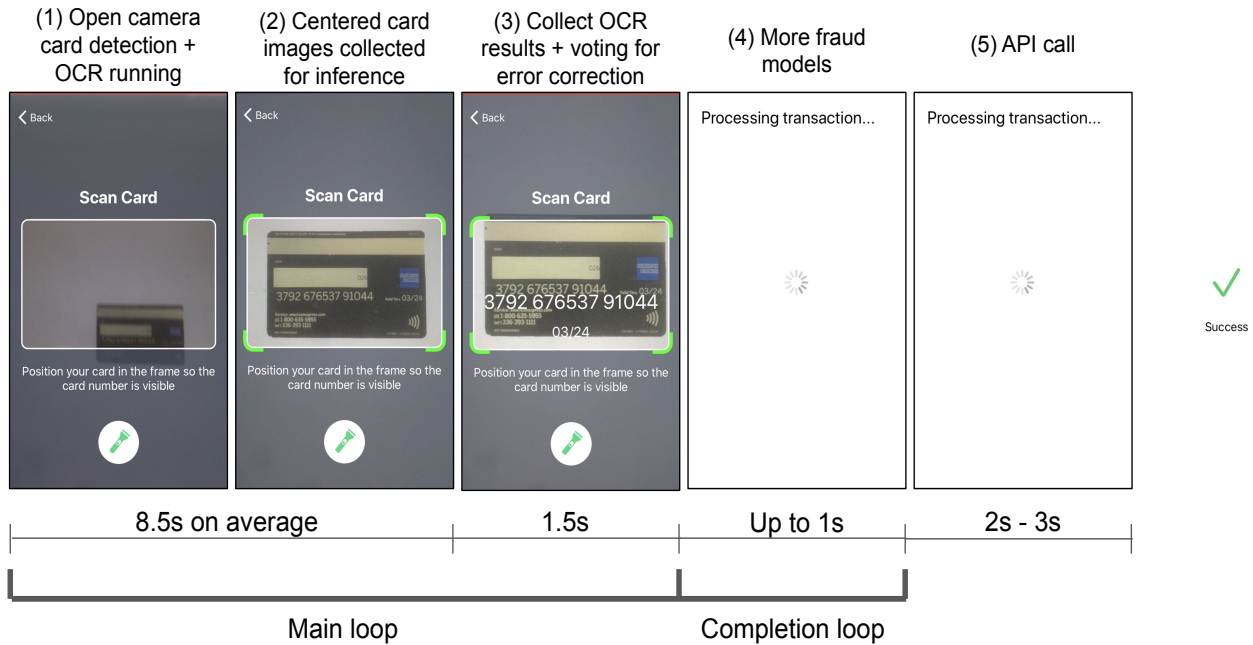


FIGURE 5.1. Daredevil scanning one side of a card from a user’s perspective.

assurance is still an ongoing arms race between app builders, device manufacturers, and attackers. Daredevil’s design favors end-user privacy even though it does open it up to client-side attacks.

5.2.2. Architecture. To scan cards and verify that they are genuine, Daredevil asks users to scan the front and the back of their card. This makes Daredevil flexible enough to verify a wide range of card designs, where meaningful information can be on either side of the card. Scanning both sides also provides more data for it to detect signs of tampering than if it scans only a single side. The checks inspect individual card sides to ensure that they are genuine, as well as combining information from both sides to make sure that it is consistent.

However, scanning both sides of the card complicates the machine learning aspects of verifying a card. First, credit cards are free to print design elements on either side. Second, users are unaware of which side of the card is the front versus the back. Therefore, Daredevil must be flexible enough to pull out the appropriate information to detect fraud dynamically and adapt automatically to scan the appropriate side of the card for each scan. The net result is that to verify cards Daredevil must run more machine learning models than it would if it were just scanning a single side of the card.

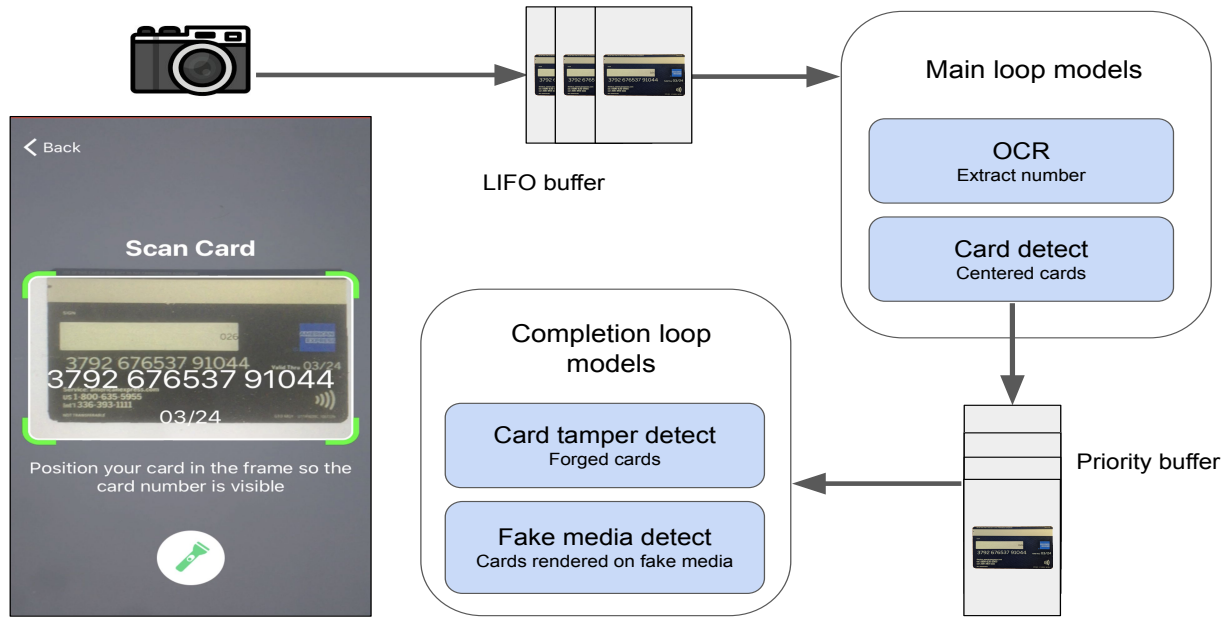


FIGURE 5.2. Machine learning pipeline for client-side models. This pipeline uses two different producer/consumer modules and divides the computation up into a *main loop* and a *completion loop*. The main loop runs on images in real time as the camera extracts images, and the completion loop runs after the main loop finishes but before making the final API call.

Figure 5.1 shows this overall process from a user’s perspective. First, the user (1) opens the flow, which starts the camera. Then (2) when they put the card in the center of the viewport, Daredevil updates the user interface to give them feedback. In parallel, (3) the card detection and the OCR models run and display the details that the OCR extracts from the card. The first successful prediction can take a variable amount of time (8.5s on average in our experiments). After the first successful OCR prediction, Daredevil continues running the card detection and OCR models for another 1.5s. This is done to collect any additional predictions for error correction. During this time, all the additional predictions made by the model are stored and the prediction with the maximum count (votes) is used as the final prediction. After the error correction process completes, (4) Daredevil runs the fake media detection and card tampering detection models on a subset of the images that are processed for up to 1s, before (5) making an API call to the server to

judge if the scan included a genuine physical card. This API call includes the output of the client-side machine learning models and the server-side logic implements rules to make a final overall decision about the validity of a scan.

Figure 5.2 shows the client-side machine learning pipeline for processing images (frames) from the camera. This pipeline uses two different producer/consumer modules and divides the computation up into a *main loop* and a *completion loop*. The main loop runs on images in real time as the camera extracts images, and the completion loop runs after the main loop finishes but before making the final API call.

This flow described above shows the scanning process for a single side of the card, but Daredevil scans both sides of the card using the same basic process before making the final API call. Daredevil introduces a card detection model that detects the side of the card (number side and non-number side), which is used as the basis for the two-side scan. See Section 5.3.4 for more details.

5.3. Design

5.3.1. Challenge: Where to run verification? Card verification can either run on the client or on the server. Server-side verification moves compute intensive machine learning inference away from the edge, which ensures verification can run on all phones regardless of their compute capabilities while also simplifying the role of the client to merely relay data to the server. However, server-side verification puts higher strain on network bandwidth and latency, with the need to transmit frames from the camera to the server, resulting in delays in verification and potentially unacceptable increases in cost for the end user who has to pay for the extra bandwidth.

Server-side verification also disregards end-user privacy. With server-side verification, the app sends sensitive user information, such as card images, to the server, thereby introducing potential avenues for data breach.

Running verification on the mobile client involves running compute intensive machine-learning inference on the client and only sending high-level features to the server. This client-first architecture puts less strain on the network and can process more frames faster by virtue of running closer

to the camera. Importantly, client-side verification is more respectful of end-user privacy since it avoids sending sensitive card images to the server.

5.3.2. Solution: Run verification on the client. The position of this work is that there are more good users than fraudsters and respecting the good user’s privacy should be the foremost concern for anyone attempting to combat fraud. Additionally, one way fraudsters source stolen card information is through data breaches, and this work strives to minimize these avenues. Thus, Daredevil chooses to run its verification on the client. Daredevil’s system design and algorithmic improvements strive to ensure the running of uniform verification on resource-constrained and well-provisioned devices across different platforms.

5.3.3. Challenge: How to ensure high verification accuracy on a mobile phone? The input to Daredevil’s models is an image or a video stream of a user holding a card. Changes in illumination, varying camera quality, orientation of the payment card, wear patterns on the card, and so on add to the stochasticity of the inputs, which makes it difficult to ensure high accuracy. However, since this input is used to verify or block a user, high accuracy is critical to provide uniform verification.

A common solution to ensure high accuracy in machine learning is to increase the model size. However, apps are hesitant to increase the size of their binary [108] since mobile networks can be slow and content distribution networks are expensive (a 5MB machine learning model downloaded 50 million times in a month costs north of \$30k / month), complicating model downloads in the background. All of which puts pressure on client-side machine learning to keep model sizes down while still providing fast and accurate predictions.

5.3.4. Solution: Decompose verification to sub-tasks for improved efficiency and redundancy. Daredevil decomposes card verification into multiple tasks, with each task having its own independent machine learning model. Decomposition of the verification process into sub-tasks keeps each sub-task efficient while also providing redundancy across tasks for improved accuracy. Decomposition also enables the pipeline to iteratively refine models for each individual task until the models reach an acceptable level of accuracy.

Daredevil decomposes verification into four distinct sub-tasks: OCR, card detection, fake media detection, and card tampering detection. OCR scans the number side of the card and extracts the card number, card detection detects frames where the user centers the card in the viewport and detects the side of the card that the user scans (number or non-number side), and fake media detection checks both sides of the card to detect cards scanned off fake media such as device screens, paper, cardboard etc.

Card tampering detection also scans both sides of the card to detect signs of tampering and inconsistencies. Daredevil scans both sides since newer card designs have meaningful information printed on both front and back. For instance, newer Wells Fargo payment cards contain the bank and payment network logos on one side and the card number and expiry on the other side. In this case, if the card tampering detection detects a Wells Fargo card number on one side and detects a conflicting bank logo on the same or opposite side, Daredevil flags the scan as fake.

Decomposition leads to higher accuracy in two ways. First, decomposition makes the overall system more efficient, allocating limited ML resources towards the images that are most likely to generate meaningful signals (Section 5.3.4.1). Second, decomposition provides redundant signals to increase the confidence of the predictions that Daredevil makes (Section 5.3.4.2).

5.3.4.1. *Efficiency with decomposition.* If every frame coming from the camera is passed through all the machine learning models, then computation is wasted. For example, if there is an image without a card in it, then running the fake media detection model or the card tampering detection model on that image is wasteful because there isn't even a card in the image, and it won't provide meaningful results.

Instead, to make the overall ML pipeline more efficient, Daredevil divides computation up into a *main loop* that runs on all frames in real-time, and a *completion loop* that defers running of models and operates on only a subset of the frames that it believes are most likely to have relevant fraud signals. Logic in the main loop dictates which frames it passes on to the completion loop, which in Daredevil are any images that have centered cards in them. Figure 5.2 shows Daredevil's decomposition.

At the heart of the design is the card detector model. The card detector model is a 3-class image classifier that is trained to detect a centered image of the number side or a centered image

of the non-number side of a card. The card detector also has a third class, called the background class, to filter out frames that contain off-center cards or no cards at all.

Daredevil executes the card detector and OCR models in the main loop. The reason that these are run in the main loop is that they both produce user-visible outputs (Figure 5.1). The card detection model highlights the corner of the viewport when it detects a centered card and the OCR model displays the recognized card number and expiration date using an animation as it captures them. Thus, these models must run in the main loop to process frames in real-time and display their results to the user. Daredevil finishes the main loop after the OCR model makes a successful prediction and the card detection model detects the presence of the card in the viewport for an additional 1.5 seconds after the first successful prediction by the OCR model.

Daredevil executes the fake media detection and card tampering detection models on the completion loop. These models only produce a result that the system uses to detect fake cards via an API call, so Daredevil defers execution until after the main loop finishes and only runs them on a subset of frames (up to six in the current system) identified by the card detector model that are likely to produce evidence of fake cards. The decomposition keeps the system efficient by having the completion loop save computation by only processing frames with centered cards.

5.3.4.2. *Redundancy with decomposition.* Redundancy is the most important lesson learned from this implementation. Even if a model achieves an accuracy of 100% on a benchmark validation dataset, it can still fall short in a practical system. Instead, one needs to supplement these predictions with additional data via voting and validation signals.

Daredevil uses different forms of redundancy for each of its models to provide high confidence in the accuracy of its decisions. Some models have a built-in validation signal for redundancy, while others require external validation signals.

More concretely, OCR has redundancy built into its design from the Luhn algorithm [33], which is a checksum used to validate credit card numbers. Thus, OCR predictions are validated by making sure that they satisfy the Luhn checksum.

In contrast, the card tampering detection model detects prominent objects on cards (e.g., the Visa symbol) and the fake media detection model detects cards scanned off fake media, and neither contain a built-in validation signal. Thus, Daredevil uses the predictions of the card detection

Task	Redundancy used	Redundancy provided
Card detection	None	Centered and focused card present
OCR	Luhn + voting	Card number and location
Card tampering	Voting + validation from card detection and OCR	None
Fake media detection	Voting + validation from card detection	None

FIGURE 5.3. Summary of task-level redundancy in Daredevil.

model and OCR to provide redundancy. Correlating predictions between models reinforces their decisions. For example, predictions of seeing a card by the card detection model, and detecting the presence of a Visa symbol by the card tampering detection model reinforce each other, since these predictions provide a higher confidence of a card being present. This may not be the case if a user scans a driver’s license (or a piece of plastic with just the number on it), where the card detection model might predict a card but the card tampering detection model may not detect any visual design elements.

Additionally, OCR, card tampering detection, and fake media detection benefit from voting on predictions across the frames they process for redundancy. For example, if the fake media detection model processes five frames and predicts the presence of a computer screen on three of them, and no screen on the remaining two, its final decision is that a screen is present. Figure 5.3 summarizes the different forms of redundancy used with each model.

5.3.5. Challenge: How to account for resource-constrained mobile phones? Owing to differences in sensor quality and compute capabilities, there is a stark difference in the performance of running image processing machine learning tasks on resource-constrained and well-provisioned phones. At best, the performance differences inconvenience users by making them wait longer to verify their cards; at worst, it prevents them from verifying their payment methods. In either case, fraud systems penalize users attempting to verify their payment method simply for not possessing a well-provisioned phone.

The measurement study in the previous chapter (Section 4.2) shows first-hand the stark differences in running the same machine learning models on well-provisioned and resource-constrained devices in a production setting. Even though machine learning inference is expected to improve with streamlined accelerated hardware support on iOS which will bridge the gap between resource-constrained and well-provisioned iPhones, it continues to be a problem on Android phones due to

inherent hardware heterogeneity. With over 2000 SoCs in distribution, optimizing for each of them is exceedingly difficult. Thus, to have uniform verification on all devices irrespective of hardware capabilities, there is a need for software enhancements for efficient machine learning inference.

5.3.6. Solution: Refine machine learning models and improve system design to provide faster effective frame rates. Daredevil’s solution to account for resource-constrained phones consists of algorithmic machine learning improvements for faster inference times and refined system design for higher utilization of the hardware.

5.3.6.1. *Improvements in machine learning.* The following two key principles inform the machine learning re-design:

(1) Optimize machine learning for resource-constrained phones: Machine learning optimization for resource-constrained phones translates to well-provisioned phones as well, but the reverse is not true since well provisioned phones often employ hardware acceleration optimized for efficient machine learning inference. Having this hardware support means that the capacity of machine learning models can be increased either by adding more parameters or by breaking a problem into sub-problems each executed with a separate machine learning model. This has a sub-linear slow-down in performance, leading to a better speed versus accuracy trade off. However, on resource-constrained phones adding parameters to the model results in at least a linear slow-down in performance.

Daredevil thus creates a unified model for OCR, where the model shares parameters for detection and recognition, thereby reducing the parameters by half. This leads to a quadratic speed up on resource-constrained phones and close to a linear speed up on well-provisioned phones as well. The new model also occupies half the disk and memory space of the original model, an added benefit for memory constrained devices. In addition to the algorithmic improvements, using a single model avoids expensive and complex processing to convert the output of one model into the input of another, leading to a more efficient implementation with less code needed to interpret the results.

(2) Optimize machine learning for the common case: Following the previous design principle of using a single model for OCR implies that the system is operating at half the machine learning capacity as before (due to the same model being used for detection and recognition), leading to an inevitable tradeoff between accuracy and speed. With a unified model for OCR there

is a need to add complex auxiliary layers at multiple stages in the model to scan all payment card designs. However, these auxiliary layers add parameters to the model as well as increase the post processing complexity making them prohibitively slow on resource-constrained devices.

Daredevil thus adds native support in the model for the most common designs and employs system design strategies to account for less common card designs. This ensures that machine learning inference is efficient for the common case, employing gated execution of more complex pipeline for less common cases.

OCR model design: With the above two design principles, a new OCR model is designed and implemented to work in a single pass. The new model draws on ideas from existing work on Faster-RCNN [106], SSD [86] and Yolo [104].

Boxer’s detection and recognition stages, which were implemented using two separate models, are replaced with a single network. The network reasons globally about the entire image resulting in end-to-end training and faster inference. The model is implemented as a fully convolutional MobileNetV2 [110] with auxiliary features for detection and recognition, unifying separate components of the detection and recognition into a single network. These features are appended to the network at different layers to account for multi-sized feature maps, like SSD [86]. This flexibility gives Daredevil the ability to operate on credit cards with varied font sizes.

The OCR model operates on an input image size of 600x375 pixels, which is close to the aspect ratio of a credit card. As with any CNN, the feature map shrinks in size and expands in the depth dimension as the network processes the image. Auxiliary layers are added to the network at two places, one where the feature map size is 38x24, and another where the feature map size is 19x12. Adding multi-layer predictions at these two layers captures the vast majority of credit card fonts. The activations corresponding to a feature map of size 38x24 are useful for small and flat font payment cards, while the activations corresponding to the feature map size 19x12 are used for embossed cards that have bigger fonts.

At the output feature maps, each activation is responsible for detecting a digit. To extract the card number from an image, there is a need to localize and recognize individual digits. Knowing the location and value of each digit in the input image aids in post processing to remove false positives. Accordingly, each activation in the two output feature maps is mapped to a regression layer (for

localization) and a classification layer (for recognition). The regression layer is implemented with anchor boxes like Faster-RCNN [106], where the possible output locations are captured with multi-aspect-ratio bounding boxes. Unlike Faster-RCNN which uses nine anchor boxes per location, this model only uses three, since these are sufficient for OCR.

To each output feature map activation a regression layer is appended that consists of mapping each input activation to 12 output activations, since the model outputs three bounding box proposals each containing four coordinates. Each of these proposals (bounding boxes) can contain a digit that the classification layer detects. The classification layer maps each input activation to 33 output activations, 11 activations (background, 0 to 9) per bounding box. During inference standard post processing techniques like non-max suppression [27] and heuristic based refining are applied that are relevant to different credit card designs.

The OCR model has difficulty in localizing small objects precisely, much like Yolo [104] and SSD [86]. Since each output activation is responsible for detecting a single digit, if the corresponding receptive field of a single activation spans multiple digits, the model will only be able to detect a single digit. One way to fix this corner case is to make the input feature map size bigger or add auxiliary layers earlier in the network where the feature map sizes are bigger. However, this adds more computation to the machine learning inference, effectively decreasing the frame rates on resource-constrained devices.

To successfully perform OCR on payment cards with tiny fonts, the ratio of the size of an individual digit is detected and compared to the size of the input feature map. If it is below the empirically determined threshold, a zoomed in image of the input is passed through the machine learning pipeline, effectively mapping a card with small font to one with a relatively bigger font that the model supports natively. This flow adds latency to the overall inference pipeline; however, Daredevil only needs to trigger it sparingly.

Daredevil uses 1 million real and synthetic card images to train the OCR model. However, based on the internal benchmark datasets, this model is unable to reproduce Boxer OCR's precision and recall, owing to the overall reduced number of parameters. To account for this reduction, an additional 1.5 million synthetic credit card images are generated for training. Ultimately, the OCR

model is trained with 2.5 million real and synthetic card images to match Boxer OCR’s baseline on the benchmark datasets.

5.3.6.2. *Improvements in system design.* To further increase the frame rate, Daredevil’s system design is refined to use a producer/consumer pipeline with a bounded buffer. Multiple frames from the camera are collected and machine learning inference is executed on all of them in parallel. Since running machine learning inference takes time, this design ensures that fetching frames from the camera is not blocked, making the entire system parallel from reading camera frames to completing machine learning inference.

Overall, buffering images and running the same inference in parallel leads to speedups of up to 117% for this workload (see Section 5.4.7 for the results). Processing more frames per second is critical for improving the end-to-end success rate for complex machine learning problems that demand high accuracy as concluded in the measurement study (see Section 4.2 in the previous chapter).

5.4. Evaluation

In the evaluation, the following questions are answered:

- Does Daredevil bridge the gap between low- and high-end devices?
- Does Daredevil prevent fraud in the wild using an ethical approach?
- What is Daredevil’s false positive rate when scanning real cards and running anti-fraud models?
- Does the use of redundancy improve overall accuracy?
- What is the impact of back-end networks and data augmentation on overall success rates?
- How does Daredevil’s OCR compare against other card scanners?
- What is the impact of the producer/consumer design on frame rates?
- Will increasing the frame rate further continue to increase the success rate?

5.4.1. Does Daredevil bridge the gap between low- and high-end devices? This section reports Daredevil’s performance for its most complex and carefully designed machine learning model: OCR. Although OCR is a critical part of the fraud system (see Section 5.4.2 for real-world

Model	Size	# params.	# 2D Conv(s)	# Depth-wise Conv(s)
Daredevil’s OCR	1.65MB	861,242	39	25
Boxer’s OCR	2.94MB	1,528,919	30	18

FIGURE 5.4. Comparison of model parameters and architecture of Daredevil’s OCR (44% fewer parameters) and Boxer’s OCR. Developers using architectures similar to these models for other applications can expect to see similar frame rates as presented in this evaluation.

results of using Daredevil to stop fraud), in this experiment OCR is used to help people add credit and debit cards to an app more effectively by scanning instead of typing in the numbers.

5.4.1.1. *Measurement Platform.* To measure the performance of Daredevil’s OCR, a correlation study is performed by making it available to third-party app developers and measuring the success rate for the users of their live production apps. Here, success rate is defined as the ratio of the number of users where Daredevil’s OCR successfully extracted the card number to the total number of users who used it. For Daredevil’s Android OCR SDK, results from anonymous statistics sent by 70 apps that deploy the library from December 2019 to late November 2020 are presented. For Daredevil’s iOS OCR SDK, results from anonymous statistics sent by 44 apps that deploy the library from late July 2020 to late November 2020 are presented.

5.4.1.2. *Testbed.* Daredevil’s Android OCR SDK ran on a total of 477,594 Android devices spanning a total of 722 Android device types. This included 328,600 Samsung devices spanning 302 Samsung device types, 42,619 Huawei devices spanning 111 Huawei device types, 6,876 Xiaomi devices spanning 38 Xiaomi device types, 22,952 LG devices spanning 78 LG device types, 31,699 Google devices spanning 17 Google device types, 18,407 Motorola devices spanning 58 Motorola device types, 8,751 OnePlus devices spanning 29 OnePlus device types and a tail of 17,690 devices, spanning 89 device types and 28 vendors. Daredevil’s iOS OCR SDK ran on a total of 1,102,666 iOS devices spanning a total of 28 iOS device types.

5.4.1.3. *Task.* Daredevil employs a single-stage OCR where both detection and recognition happen in a single pass. The input image from the camera is processed and sent to the OCR model which outputs a string of digits.

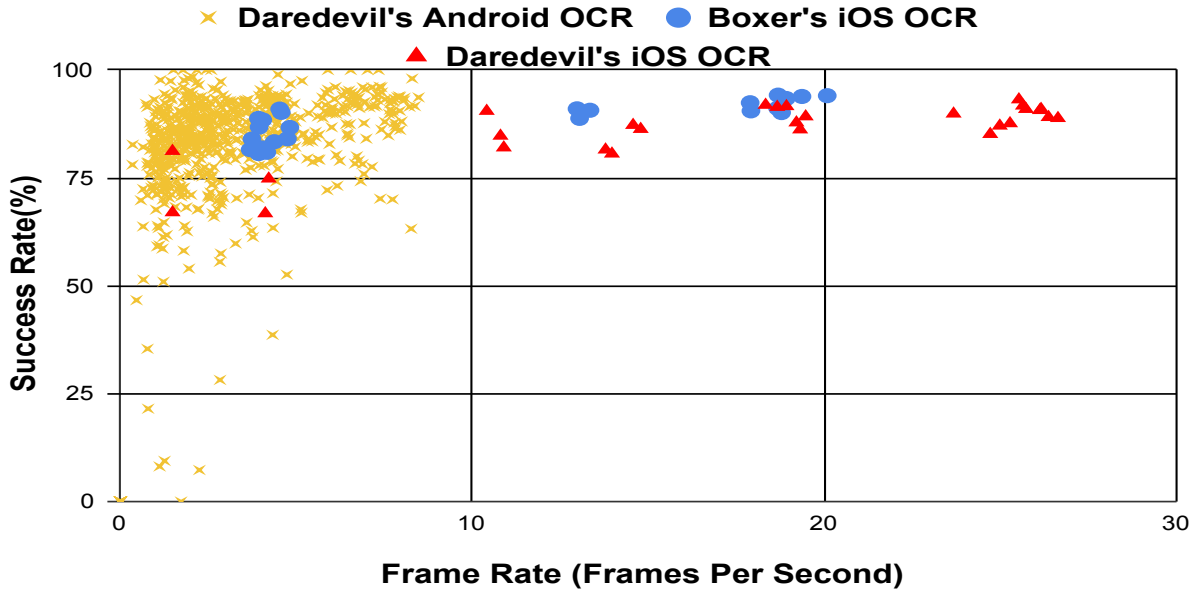


FIGURE 5.5. OCR success rate vs frame rate on Daredevil’s Android OCR, Boxer’s iOS OCR and Daredevil’s iOS OCR. Each point is the average success rate and frame rate for a specific device type. This figure shows that by improving the machine learning models and increasing the frame rate, higher success rates can be achieved.

Daredevil’s OCR uses a fully convolutional MobileNetV2 [110] with auxiliary features for detection and recognition and occupies 1.65MB on disk. The OCR model processes an input image of size 600x375 and generates 51,300 output values which are used to detect and localize the information for extraction. It has a total of 861,242 parameters of which 830,362 are trainable. Daredevil’s OCR uses 44% fewer parameters than Boxer’s OCR. Figure 5.4 shows a comparison of the model parameters between Boxer’s OCR and Daredevil’s OCR.

Daredevil uses the same inference engines (CoreML for iOS and TFLite CPU for Android) as Boxer, detailed in the measurement study Section 4.2. Like Boxer, all the models are quantized using 16-bit floating point weights.

Like Boxer, Daredevil prompts users to scan their credit cards. When invoked, it starts the camera and prompts users to place their card in the center of the viewport. Daredevil’s OCR processes frames obtained from the camera and attempts to extract the card number and expiry from the card. Upon success, it displays the card number and expiry to the user and sends the scan statistics to its server. In case the OCR is unable to extract the number, the flow doesn’t time-out.

Version	Count	Avg Suc Rate	Avg FPS	Avg Dur (s)
Daredevil's iOS OCR	1,102,666	89.13%	20.00	9.37
Boxer's iOS OCR	3,175,912	88.60%	10.00	10.02
Daredevil's Android OCR	477,594	88.46%	4.07	10.55
Boxer's Android OCR	329,272	46.72%	1.30	15.45

FIGURE 5.6. Comparison of Daredevil's OCR and Boxer's OCR. It can be seen that Daredevil's OCR not only provides over 41% improvement in success rates on Android but also improves iOS by close to 1%.

Instead, it lets the user cancel the scan which provides an additional user-level metric that can be used to guide a future iteration.

5.4.1.4. *Results- Key Performance Metrics.* This study uses the following metrics.

Success rate: Success rate is defined as the ratio of the number of users where the OCR successfully extracted the card number to the total number of users using it.

Frame rate: Frame rate is defined as the number of frames from the camera processed by the OCR pipeline per second.

This study shows the impact that the Daredevil's OCR model (Section 5.3.6.1) has on the overall scanning success rate. The informal goal with Daredevil's OCR was to improve the success rates on Android to match Boxer's iOS OCR. Daredevil's OCR uses algorithmic machine learning improvements, empirical accuracy-preserving optimizations, high fidelity synthetic data, and an improved system design to achieve this goal.

Figure 5.5 shows the results of Daredevil's OCR deployed on Android (referred to as Daredevil's Android OCR) and iOS (referred to as Daredevil's iOS OCR) against Boxer's iOS OCR. This figure shows that Daredevil's OCR improvements increase the success rate on Android to closely match the success rates of Boxer's iOS OCR, despite the massive hardware advantages present on iOS. Seeing the success of Daredevil's Android OCR, it was ported to iOS and provided a more than 2x speedup in frame rates and a moderate improvement in the success rates as well (Figure 5.6). The increase in frame rates also led to Daredevil's iOS OCR being able to support iPhone 6 and below, not supported in Boxer's iOS OCR.

Concretely, Figure 5.6 shows the average frame rate improves from 1.30 FPS on devices running Boxer's Android OCR to 4.07 FPS on devices running Daredevil's Android OCR. Daredevil's

	Daredevil's OCR		Boxer's OCR	
Android FPS	Count	Success rate	Count	Success rate
< 1 FPS	23,314 (4.88%)	37.92%	146,890 (44.61%)	31.87%
1–2 FPS	48,271 (10.10%)	84.08%	97,798 (29.70%)	49.97%
>= 2 FPS	406,009 (85.01%)	91.88%	84,584 (25.68%)	68.72%

FIGURE 5.7. Success rates for Android devices running Daredevil's OCR and Boxer's OCR by frame rate. It can be seen that Daredevil's OCR significantly reduces the percentage of devices that operate below 1 FPS.

Android OCR also increases the average success rate from 46.72% to 88.46%. There is also an improvement in success rates on iOS, going from 88.60% on Boxer's iOS OCR to 89.13% on Daredevil's iOS OCR. Additionally, the average scan duration decreases from 15.45s to 10.55s on Android and from 10.02s to 9.37s on iOS. In this system, the scan duration timer is started when the user clicks on the "scan card" button and finished after the scan is completed, which includes accepting camera permissions, pulling their card out of their wallet, scanning the card, and the 1.5s voting phase for error correction in the main loop.

Daredevil's OCR also improves the usability of card scanning with only 4.88% (Figure 5.7) of Android phones being able to process fewer than 1 FPS, compared to 44.61% with Boxer's OCR. Similar to Boxer's OCR, the success rate for Android devices with less than 1 FPS (37.92%) is lower than the average success rate for Android overall (88.46%); however, the overall increase in devices that can run the Daredevil's OCR at 1 FPS or higher leads to a higher overall success rate (Figure 5.6 and Figure 5.7).

Figure 5.7 shows that for both Boxer's OCR and Daredevil's OCR, as the frame rate increases the overall success rate increases as well. Beyond 1 FPS, the success rate for Daredevil's OCR witnesses a precipitous rise compared to Boxer's OCR, which can be attributed to Daredevil's OCR being trained with orders of magnitude more data (Section 5.4.5), the use of an efficient machine learning pipeline (Section 5.3.4), and the marginal improvements seen from the updated back-end network (Section 5.4.5). It is clear that Boxer's OCR can also benefit from these improvements; however, given that 44.61% of the Android devices operate at below 1 FPS for Boxer's OCR (and Daredevil's OCR also struggles with devices that operate at frame rates below 1 FPS), a significant portion of the devices will be excluded from these improvements.

Platform	Count	Avg FPS	Avg Scan Duration (s)
Daredevil's Android OCR	55,093	3.04	22.58
Daredevil's iOS OCR	119,826	18.77	17.38

FIGURE 5.8. Failure cases for Daredevil's OCR on Android and iOS. The number of iOS failures is higher than Android due to the larger representation of iOS devices in the deployment. The average scan duration is the average time for which a user waits before cancelling the scan.

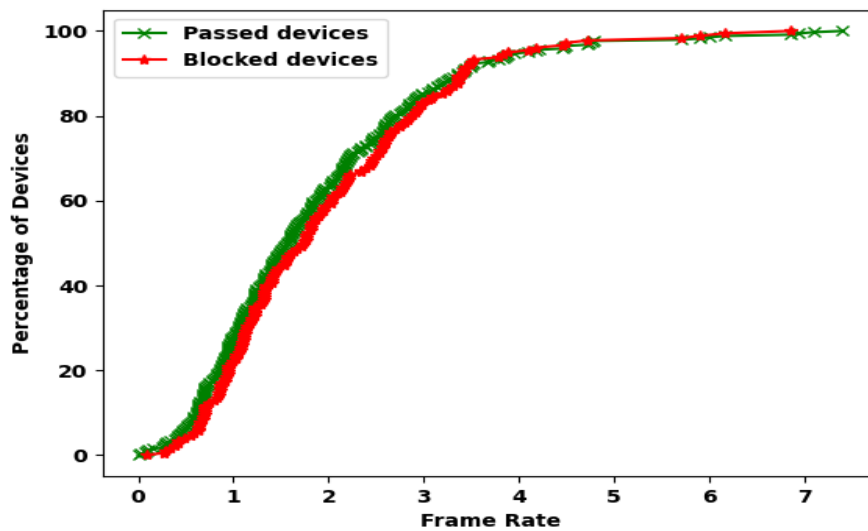


FIGURE 5.9. Cumulative Distribution Function (CDF) of percentage of devices against the frame rates for devices passed and blocked by Daredevil. It can be seen that the two plots look very similar, indicating that Daredevil's fraud decision is largely independent of the frame rate.

5.4.1.5. *Analysis of failure cases of Daredevil's OCR.* From the real-world deployment of Daredevil's OCR there were 174,919 failed attempts where users gave up on trying to scan their card. On average Android users waited 22.58s and iOS users waited 17.38s to scan their cards before giving up. This is shown in Figure 5.8.

5.4.2. Does Daredevil’s prevent fraud in the wild using an ethical approach? To evaluate Daredevil’s ability to stop fraud in real-time, results from a large international app deploying Daredevil are reported. For a test period of 3 months, the app flagged 12,474 transactions as suspicious and challenged them with Daredevil to verify their payment method.

Daredevil passed 7,612 transactions and blocked the remaining 4,862 transactions. Of the 7,612 transactions passed by Daredevil, only 12 resulted in chargebacks, leading to a false negative rate of 0.16%. Here, a false negative is a fraudulent transaction incorrectly labeled as legitimate and a false positive is a legitimate transaction incorrectly labeled as fraudulent. This study is unable to report the false positive rate since the app did not share the false positive data, please see Section 5.4.3 for an evaluation of the Daredevil’s false positive rate. Based on this initial test, the app has decided to continue to deploy Daredevil.

To determine if Daredevil’s fraud decisions are correlated with the device frame rates, the performance characteristics of the passed and blocked devices are further analyzed. The average frame rate of devices that Daredevil passed was 1.84 FPS and the average frame rate of the devices that Daredevil blocked was 1.94 FPS, indicating that the frame rates for the two groups are roughly the same. To visualize these results, the Cumulative Distribution Function (CDF) of the percentage of devices vs frame rate (FPS) for the two groups is plotted in Figure 5.9. It can be seen that the plots look very similar, indicating that frame rate is not a determining factor between the blocked and passed groups.

For companies, chargebacks are the ground truth because they represent exactly what they are liable for financially. However, it is possible that there was fraud that happened but the victim failed to report the fraudulent charge to their issuing bank, thus the actual amount of fraud may be higher than the chargeback count that is reported in this experiment.

5.4.3. What is Daredevil’s false positive rate when scanning real cards and running anti-fraud models? To evaluate Daredevil’s false positive rate, 105 real cards are scanned in a lab setting. In this experiment, the fraud flow invoked and the number of scans that the system incorrectly flags as being fraudulent are recorded. This section complements the real-world evaluation of Daredevil’s fraud systems in 5.4.2 that shows the false negative rate.

	Card tampering detect. # errors	Fake media detect. acc.
No Card Detection	1.94 errors per frame	86.24%
With Card Detection	1.26 errors per frame	95.26%

FIGURE 5.10. Results from the user study indicate fewer errors made by the card tampering detection model and higher accuracy of fake media detection model when aided by card detection. An error for the card tampering detection model is one where it incorrectly detects an object (like VISA logo, etc.) *not* present on the card or fails to detect an object present on the card.

105 different real cards are scanned multiple times on different resource-constrained and well-provisioned Android and iOS devices for a total of 310 scans. The devices used are iPhone SE (1st gen), Google Pixel 2, Nexus 6, iPhone 6s, and iPhone 11. Of these 310 scans, Daredevil incorrectly flags seven scans as fraudulent, giving a false positive rate of 2.2%. The false positives are uniformly spread across all devices, indicating that Daredevil does not unfairly permit well-provisioned or resource-constrained devices, similar to the fraud decisions as discussed in Figure 5.9.

Six out of the seven reported false positives were transient in nature, i.e. further scans of the same card (which are expected from a good user) did not result in false positives. The other card was consistently flagged incorrectly by the fake media detection model on all the devices in the test set.

5.4.4. Does the use of redundancy improve overall accuracy? This sections seeks to evaluate the effectiveness of the redundancy based decomposition strategy (described in Section 5.3.4) in aiding fraud detection. Specifically, this sections evaluates the gains in accuracy on executing the card tampering detection and fake media detection models in the completion loop.

A user study is performed with and without the card detection model in the main loop to see how the card detection model benefits the card tampering detection and fake media detection models running in the completion loop. The user-facing feedback from the card detection model ensures that users center their credit cards so that both models necessarily make their predictions on valid credit card images.

Users participating in the study randomly run one of two versions of the app and scan 30 different predetermined credit card images on a browser that are provide via a link. These scans are then used to evaluate the impact of the feedback from card detection in terms of the number of

Back End	Size	No. of Params.	Recall	Precision	FPS on Pixel 3a
MBv1	1.8MB	869,754	54.06%	100%	7.19
MBv2	1.65MB	861,242	56.25%	100%	7.09

FIGURE 5.11. Comparison of model parameters and accuracy metrics on the benchmark datasets using Daredevil with back-ends MobileNet V1 (MBv1) and MobileNet V2 (MBv2). It can be seen that using MobileNet V1 as back-end leads to less than 1% increase in model parameters with no decrease in precision and marginal decrease in recall. It should be noted that the second model (with back-end MobileNet V2) is currently in production, all the statistics from Daredevil’s evaluation correspond to this model.

mistakes made by card tampering detection (i.e. objects present on the card that the model fails to detect as well as objects not present on the card that the model incorrectly detects) and the accuracy of fake media detection in detecting both the presence and absence of screens.

Figure 5.10 summarizes the results. The design of decomposition centered on the card detection model ensures that high-quality frames are passed to the machine learning models, resulting in fewer errors for the card tampering detection model and decreasing the errors per frame from 1.94 errors per frame down to 1.26 errors per frame. This change also improves accuracy for the fake media detection model, increasing the accuracy from 86.24% to 95.26%. Overall, these improvements lead to more accurate fraud detection.

5.4.5. What is the impact of back-end networks and data augmentation on overall success rates? Boxer’s OCR uses MobileNet V1 as a back-end, while Daredevil’s OCR uses MobileNet V2. To quantify the impact of back-end networks and determine whether the improvements in Daredevil’s OCR are solely due to updating the back-end network, an experiment is performed that validates the models on image frames extracted from videos recorded by users scanning their credit cards. Crucially, this is the same benchmark used to evaluate models that are shipped in production. The test set consists of 640 image frames extracted from 32 videos. Daredevil’s OCR is trained with MobileNet V1 and MobileNet V2 back-ends, and the results are presented in Figure 5.11. A correct prediction is defined as one where the model can correctly extract the card number from the image frame, while an incorrect prediction is one where the model extracts an incorrect card number (valid but incorrect). All frames where the model is able to extract only a partial

No. of images	Recall	Precision
495,134	20%	98.46%
939,165	27.96%	98.89%
1,374,707	42.08%	99.26%
2,006,452	49.06%	100%
2,500,612	56.25%	100%

FIGURE 5.12. Impact of varying the amount of training data on model accuracy. The model consists of Daredevil’s OCR with MobileNet V2 back-end.

number are considered missed predictions. Accordingly, recall is the fraction of the frames where the model produced a correct prediction and precision is the fraction of all the predictions that were correct.

Critically, from Figure 5.11 it can be seen that using MobileNet V2 instead of MobileNet V1 as the back-end network results in less than 1% reduction in the number of parameters, indicating that the reduction in the overall parameters is a direct result of Daredevil’s OCR architecture independent of the back-end network. It can also be seen that Daredevil’s OCR with MobileNet V1 closely matches Daredevil’s OCR with MobileNet V2 in recall and precision (Figure 5.11) further highlighting its back-end agnostic nature.

To quantify the impact of data augmentation on the improvement of overall success rates, Daredevil’s OCR (Mobile Net V2 back-end) is trained by varying the amount of training data. The training data is generated using a custom Generative Adversarial Network (GAN) [29] architecture and standard data augmentation techniques are also used in addition to the GAN. The models are also evaluated using the same benchmark as before and the results are presented in Figure 5.12.

In summary, it can be concluded that Daredevil’s OCR architecture is necessary to achieve the desired frame rate and high-fidelity synthetic data is necessary to achieve the desired accuracy.

5.4.6. How does Daredevil’s OCR compare against other card scanners? Card.io [98] is a popular open-source scanning library commonly used in the industry. An experiment is run to compare Daredevil’s OCR against Card.io to measure its scan success rates on a benchmark test set of 100 credit cards. Daredevil’s OCR is able to extract the correct card number from each card, while Card.io is able to extract the correct card number from only 58 cards. Accordingly, Daredevil OCR’s precision and recall are both at 100%, while Card.io’s precision and recall are 100% and

58% respectively. The lower recall of Card.io is attributed to its inability to scan cards with flat fonts.

5.4.7. What is the impact of the producer/consumer design on frame rates? Daredevil processes frames obtained from a live camera feed. In most cases, the camera runs at a higher frame rate than the machine learning model, meaning that applications will have to drop some number of frames while the user is scanning their card. A natural and common solution to this problem is to block the live feed while the prediction runs, waiting for the machine learning models to finish processing the frame before grabbing the next available frame from the camera. This solution leads to a lower effective frame rate because of the waiting time, but is memory efficient and ensures that the models always have fresh data to process by virtue of using only the latest frame from the camera.

Instead of processing each camera frame serially and blocking the live feed while the model runs, Daredevil uses a producer (the camera)/consumer (machine learning models) architecture with a bounded LIFO buffer to store the most recent frames, and runs multiple predictions in parallel. This architecture comes at the increased cost of memory but enables the machine learning models to execute without any waiting and ensures that the models process frames that are close to what the user sees.

It has been already shown that the producer/consumer design leads to higher frame rates and success rates in production (Section 5.4.1). In this section, a controlled lab experiment is conducted to compare the frame rates between the blocking design and producer/consumer design of running the main loop (the card detection and OCR models) on frames produced from a fixed camera feed. This is followed by a qualitative analysis of why the blocking design is slower on both Android and iOS, despite the considerable differences in how the two platforms execute machine learning inference.

Specifically, three different variations are considered: (1) a blocking style with a single instance of the main loop models driven at the frame rate of the camera, (2) a non-blocking style using a buffer to store the two most recent frames with a single thread running the main loop models, and (3) a non-blocking style using a buffer to store the two most recent frames with two threads on iOS and four threads on Android running the main loop models. This experiment is run by measuring

Device	Blocking	+ Buffer	+ Parallel
iPhone 5s	1.65 fps	1.70 fps	2.95 fps
iPhone SE	7.60 fps	7.90 fps	14.90 fps
iPhone XR	28.45 fps	32.60 fps	32.60 fps
LG K20 Plus	1.03 fps	1.04 fps	1.39 fps
Xiaomi Redmi 7	3.16 fps	3.47 fps	4.89 fps
Pixel 2	3.66 fps	4.35 fps	7.95 fps

FIGURE 5.13. Frames per second (FPS) for 20 second run. This figure shows the performance improvement measured by frames processed by the main loop per second with the baseline of a blocking system, a system that buffers images, and a system that buffers images and runs the ML models in parallel.

the frame rates observed on running the three variations on different iOS and Android devices of varying capabilities for 20 seconds each.

Figure 5.13 summarizes the results from these experiments. From these results, a clear increase in the frame rates can be observed across all phones on both iOS and Android when moving from a blocking system to a system that buffers frames to a system that buffers frames and runs the main loop models in parallel.

Improvements in frame rates due to buffering alone range from 1% to 19%, with faster devices seeing larger gains. The reason that faster devices see larger gains is because the time spent waiting for a camera frame is a larger portion of the overall execution time as the time spent on machine learning predictions goes down.

5.4.8. Will increasing the frame rate further continue to increase the success rate?

This section serves to answer the question of whether increasing the current frame rates would lead to further improvements in success rate without changing the machine learning model. Since card scanning involves sending frames from a live camera feed through a machine learning model, faster frame rates could imply two consecutive frames being practically identical to the eyes of a machine learning model, leading to no gains obtained from a higher frame rate. Alternatively, it could be that there are sufficient differences between two consecutive frames for the machine learning model to produce a different and possibly better prediction, resulting in a shorter scanning duration.

Concretely, consider an example where an OCR model is able to process frames from the user's video feed at a rate of 5 FPS, and the user scans for 10 seconds. This means that OCR runs

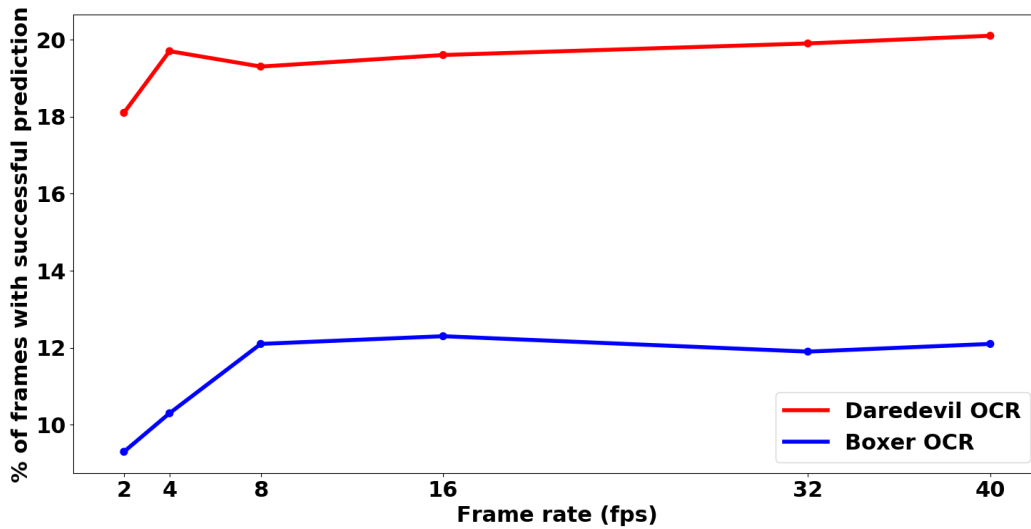


FIGURE 5.14. Plot shows that as the frame rates increase, the fraction of frames with successful predictions roughly remains constant, meaning that the number of frames with successful predictions increases with frame rate. Thus, systems enhancements to increase the frame rate, even with the same machine learning model can lead to faster scanning times.

inference on 50 frames in total. Let the frames on which the model makes correct predictions be referred to as the *useful frames*. If this model makes correct predictions on 10 frames, then there are 10 useful frames from the total set of 50 frames. Now suppose, the same OCR model processes the same 10 second feed at 10 FPS instead of 5, i.e., this model processes a total of 100 frames. If this setting results in more useful frames, then running at a higher frame rate would lead to shorter scanning times on average.

To study this, videos of users scanning cards from the user study described in Section 5.4.4 are analyzed. Different frame rates are *simulated* by extracting frames at differently spaced intervals from the recorded videos. Closer intervals represent faster frame rates and possibly identical frames, and vice versa for wider intervals. These frames are then passed through two different OCR models (Boxer OCR and Daredevil OCR) and for each frame rate, the percentage of useful frames obtained to the total number of frames processed is computed.

Figure 5.14 plots the variation of frame rates to percentage of frames with successful predictions averaged over 27 different scanning videos sampled from the user study. The plots are roughly

constant for both Boxer OCR and Daredevil OCR. This indicates that with increasing frame rates and correspondingly increasing the number of frames processed by the models, the number of useful frames (i.e., the number of frames where OCR succeeds) also increases. These results suggest that even closely spaced frames contain sufficient diversity leading to different, and possibly correct predictions with the same machine learning model. Thus, further systems enhancements that lead to higher frame rates with the same OCR model contribute to faster scan times and better user experience.

5.5. Related work

This work is related to papers in the areas of financial fraud, challenge based authentication, computer vision, machine learning systems and machine learning for mobile.

Recent work has focused on devising challenges that rely on having users interact with their mobile phones to collect signals that are then processed for verification [83,124]. Liu et al. propose CardioCam [83] to verify users based on their cardiac biometrics. Researchers have also devised authentication systems where users are challenged to respond to a Captcha challenge on their mobile phones, while collecting audio and visual data of the response that is transmitted to a secure server for processing [124].

The execution of machine learning models on resource constrained platforms such as mobile phones has seen active research in both algorithmic machine learning improvements [36,129] as well as enhanced system design [32,85,130]. Liu et al. devise a selection framework, AdaDeep, that automatically selects a combination of compression techniques to be applied to a given neural network to balance between performance and availability of resources [85]. Closer to the work presented in this thesis, researchers at Facebook extensively profile the wide diversity in compute capabilities on mobile phones for machine learning [130]. They also identify the benefits of optimizing to run inference on CPUs over GPUs to provide stable execution on Android devices, and Daredevil follows this general plan where Android models executed on the CPU but the hardware acceleration available is used on iOS to speed up the models. Ran et al. [103] create a client-server hybrid framework to provide sufficient compute power for running augmented reality apps. The authors in [92], [132] conduct a measurement study of mobile performance analysis of various deep

learning models and conclude the need for extensive optimization and both on-device and cloud based inference.

Recently there has been work on improving the performance of parallel DNN training [38, 90]. Narayanan et al. [90] cast DNN training as a computational pipeline to efficiently utilize the hardware resources. In contrast, Huang et al. [38], while also using pipelining to train large models, significantly reduce the memory overhead by recreating intermediate values.

Payment card fraud using card skimmers has been studied recently by Scaife et al. [113]. In this work, researchers built a card skimmer detector that can be used at physical payment terminals such as ATMs and gas stations. In another work, Scaife et al. [112] did a survey of gas pump card skimmer detection techniques including Bluetooth skimmer detection on iOS and Android apps, to identify common skimmer detection characteristics.

5.6. Future Work

Both Boxer and Daredevil run machine learning models client-side. Even though this design favors end-user privacy, it opens up many avenues of attack. From the apps perspective, while access to rich sensor data and compute capabilities can facilitate high-utility task-specific challenges, the apps need a way to assert security properties of these challenges as well as protect their intellectual property since the attackers could tamper with the app, the video stream, or the models. From the users perspective, while these challenges reduce friction and improve user-experience (payment checkout), the users need to be able to trust these apps with their highly-discriminative data since there is nothing preventing the apps from making sensitive inferences from the sensor data.

An ideal execution environment for these challenges is one which is a generic, privacy-preserving, task-agnostic, isolated, hardware-backed and programmable environment that can support client-side end-to-end untrusted machine learning applications. The execution environment should be able to securely access the data source to guarantee integrity of the input data, while also protect the secrecy and integrity of the machine learning models and algorithms from the outside world (user or OS). It should also be able to limit, in a task-agnostic way, the amount and nature of the data from the sensors that the machine learning models can process. Existing research has taken a two-pronged approach to solve these issues in isolation, either by leveraging Trusted Execution

Environments (TEE) to provide isolation or by creating application-specific privacy filters to protect user data. However, these two problems are intimately related and require a unifying framework. Such a unifying framework has the potential to unlock many more features without compromising on the security and privacy of the end-user or the application. I hope to see a publication in the future that implements this idea.

CHAPTER 6

Conclusion

In this work, I conceptualized, designed, implemented, and evaluated systems to realize deep learning for security on end-devices. In doing so, I demonstrated that deep learning can facilitate newer frontiers in security and empower data engineers with features that can better discriminate between good and bad actors. At the same time, while these checks are designed to be neutral and an improvement over prior prejudiced policies, these checks have the potential of reinforcing the same inequality. It is thus imperative to extensively evaluate the ethical implications of compute intensive systems when used for security, since a security decision can result in adverse action being taken against an individual or a group.

In Chapter 2, the system Percival demonstrated that it is possible to devise models that block ads while rendering images inside the browser. Percival’s implementation showed a rendering time overhead of 4.55% for Chromium and 19.07% for the Brave browser, demonstrating the feasibility of deploying deep neural networks inside the critical path of the rendering engine of a browser. It was shown that the perceptual ad blocking model can replicate EasyList rules with an accuracy of 96.76%, making Percival a viable and complementary ad blocking layer. Percival also demonstrated off the shelf language-agnostic detection due to the fact that its models do not depend on textual information. Finally, Percival provides a compelling blocking mechanism for first-party Facebook sponsored content, for which traditional filter-based solutions are less effective.

In chapter 3, Boxer, a new system for enabling apps to scan payment cards and determine if they are genuine, was introduced. Boxer combines three image analysis techniques with a novel secure counting abstraction on top of modern security hardware to provide a holistic solution to card-not-present attacks performed at scale. To date, Boxer has already scanned over 100 million cards, detected real attacks, and demonstrated how its design keeps an eye towards the future by anticipating future attacks and building defenses for them.

In chapter 4, it was demonstrated that while deep learning based security checks can limit the prejudices of prior algorithms, deep-learning-based security challenges have the potential of reproducing historical prejudices, improving the security and user experience of one group at the expense of completely blocking the other. The payment verification system Boxer was evaluated with a wide-scale measurement study consisting of 3,505,184 devices that ran in real apps. It was demonstrated that while Boxer can solve the app’s business problem by functioning reliably on high-end phones, it has the potential to disproportionately block users from low socio-economic tiers who rely on lower tier smartphones.

In chapter 5, with the lessons learned from the measurement study presented in chapter 4, a new system called Daredevil was designed, a payment card verification system that used deep learning optimizations and improved system design to build a complex security system that works uniformly on low-end and high-end mobile devices. The chapter presented the results from 1,580,260 devices from Daredevil’s public deployment to demonstrate the equitable nature of the system across all devices.

Bibliography

- [1] C. F. B. ADS., *Improving the Consumer Online Ad Experience*. <https://www.betterads.org/research/>.
- [2] K. AHUJA, K. SIKKA, A. ROY, AND A. DIVAKARAN, *Understanding visual ads by aligning symbols and objects using co-attention*, CoRR, abs/1807.01448 (2018).
- [3] J. AKHILOMEN, *Data mining application for cyber credit-card fraud detection system*, in *Advances in Data Mining. Applications and Theoretical Aspects*, P. Perner, ed., Berlin, Heidelberg, 2013, Springer Berlin Heidelberg, pp. 218–228.
- [4] APPLE, INC., *CoreML*. <https://developer.apple.com/machine-learning/core-ml/>.
- [5] V. A. BALASUBRAMANIYAN, A. POONAWALLA, M. AHAMAD, M. T. HUNTER, AND P. TRAYNOR, *Pindr0p: Using single-ended audio features to determine call provenance*, in *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS '10*, New York, NY, USA, 2010, ACM, pp. 109–120.
- [6] Y. BENGIO, A. COURVILLE, AND P. VINCENT, *Representation learning: A review and new perspectives*, *IEEE transactions on pattern analysis and machine intelligence*, 35 (2013).
- [7] A. BEUTEL, J. CHEN, T. DOSHI, H. QIAN, A. WOODRUFF, C. LUU, P. KREITMANN, J. BISCHOF, AND E. H. CHI, *Putting fairness principles into practice: Challenges, metrics, and improvements*, in *Proceedings of the 2019 AAAI/ACM Conference on AI, Ethics, and Society, AIES '19*, New York, NY, USA, 2019, Association for Computing Machinery, p. 453–459.
- [8] N. BHASKAR, M. BLAND, K. LEVCHENKO, AND A. SCHULMAN, *Please pay inside: Evaluating bluetooth-based detection of gas pump skimmers*, in *28th USENIX Security Symposium (USENIX Security 19)*, Santa Clara, CA, Aug. 2019, USENIX Association, pp. 373–388.
- [9] M. BUSTA, L. NEUMANN, AND J. MATAS, *Deep textspotter: An end-to-end trainable scene text localization and recognition framework*, in *The IEEE International Conference on Computer Vision (ICCV)*, Oct 2017.
- [10] C. CAKEBREAD, *Looking to buy some stolen credit card numbers? just head to facebook*, December 2017. <http://www.businessinsider.com/pages-advertising-stolen-credit-card-numbers-are-all-over-facebook-2017-12>.
- [11] N. CARLINI AND D. WAGNER, *Audio adversarial examples: Targeted attacks on speech-to-text*, *Proceedings - 2018 IEEE Symposium on Security and Privacy Workshops, SPW 2018*, (2018).
- [12] CAYAN, *Preventing card-not-present fraud*. https://cayan.com/Site/Media/Cayan/Insights-Content/preventing-card-not-present-fraud_cayan.pdf.

- [13] S. CHEN, N. CARLINI, AND D. A. WAGNER, *Stateful detection of black-box adversarial attacks*, CoRR, abs/1907.05587 (2019).
- [14] A. CORP., *14 credit card skimmers found in arizona in 2019*. <https://www.abc15.com/news/data/credit-card-skimmers-found-in-arizona-reaches-14-so-far-in-2019>.
- [15] S. DUTTA, D. WEI, H. YUEKSEL, P.-Y. CHEN, S. LIU, AND K. R. VARSHNEY, *An information-theoretic perspective on the relationship between fairness and accuracy*, 2019.
- [16] B. EDELMAN, *False and deceptive display ads at yahoo's right media, 2009*. <http://www.benedelman.org/rightmedia-deception/#reg>.
- [17] S. EGELMAN, February 2019. <https://blog.appcensus.io/2019/02/14/ad-ids-behaving-badly/>.
- [18] S. ENGLEHARDT AND A. NARAYANAN, *Online tracking: A 1-million-site measurement and analysis*, in Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS '16, New York, NY, USA, 2016, ACM.
- [19] T. EVERTS, *The average web page is 3MB*. <https://speedcurve.com/blog/web-performance-page-bloat/>.
- [20] H. FARID, *Digital image ballistics from jpeg quantization*, 2006.
- [21] H. FARID, *Image forgery detection*, IEEE Signal Processing Magazine, 26 (2009), pp. 16–25.
- [22] FEDERAL TRADE COMMISSION, *Advertising and Marketing on the Internet*. <https://www.ftc.gov/tips-advice/business-center/guidance/advertising-marketing-internet-rules-road>.
- [23] D. FREEMAN, S. JAIN, M. DURMUTH, B. BIGGIO, AND G. GIACINTO, *Who are you? A statistical approach to measuring user authenticity*, in NDSS, The Internet Society, 2016.
- [24] J. FRIDRICH, D. SOUKAL, AND J. LUK, *Detection of copy-move forgery in digital images*, 2003.
- [25] D. GARCIA AND R. DE QUEIROZ, *Face-spoofing 2d-detection based on moiré-pattern analysis*, IEEE Transactions on Information Forensics and Security, 10 (2015), pp. 778–786.
- [26] K. GARIMELLA, O. KOSTAKIS, AND M. MATHIOUDAKIS, *Ad-blocking: A Study on Performance, Privacy and Counter-measures*, in Proceedings of WebSci '17, Troy, NY, USA, 2017, WebSci '17.
- [27] R. GIRSHICK, J. DONAHUE, T. DARRELL, AND J. MALIK, *Rich feature hierarchies for accurate object detection and semantic segmentation*, in 2014 IEEE Conference on Computer Vision and Pattern Recognition, 2014, pp. 580–587.
- [28] I. GOODFELLOW, N. PAPERNOT, S. HUANG, Y. DUAN, P. ABBEEL, AND J. CLARK, *Attacking Machine Learning with Adversarial Examples*, OpenAI. <https://blog.openai.com/adversarial-example-research>, (2017).
- [29] I. J. GOODFELLOW, J. POUGET-ABADIE, M. MIRZA, B. XU, D. WARDE-FARLEY, S. OZAIR, A. COURVILLE, AND Y. BENGIO, *Generative adversarial networks*, 2014.
- [30] GOOGLE, *Edge TPU*. <https://cloud.google.com/edge-tpu>.
- [31] K. GROSSE, N. PAPERNOT, P. MANOHARAN, M. BACKES, AND P. MCDANIEL, *Adversarial examples for malware detection*, in European Symposium on Research in Computer Security, Springer, 2017.

- [32] S. HAN, H. SHEN, M. PHILIPSE, S. AGARWAL, A. WOLMAN, AND A. KRISHNAMURTHY, *Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints*, in Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys '16, New York, NY, USA, 2016, Association for Computing Machinery, p. 123–136.
- [33] HANS PETER LUHN, *Computer for verifying numbers*, August 1960. <https://patents.google.com/patent/US2950048>.
- [34] M. HARDT, E. PRICE, AND N. SREBRO, *Equality of opportunity in supervised learning*, in Proceedings of the 30th International Conference on Neural Information Processing Systems, NIPS'16, Red Hook, NY, USA, 2016, Curran Associates Inc., p. 3323–3331.
- [35] A. HOWARD, M. SANDLER, G. CHU, L. CHEN, B. CHEN, M. TAN, W. WANG, Y. ZHU, R. PANG, V. VASUDEVAN, Q. V. LE, AND H. ADAM, *Searching for mobilenetv3*, CoRR, abs/1905.02244 (2019).
- [36] A. G. HOWARD, M. ZHU, B. CHEN, D. KALENICHENKO, W. WANG, T. WEYAND, M. ANDREETTO, AND H. ADAM, *Mobilenets: Efficient convolutional neural networks for mobile vision applications*, CoRR, abs/1704.04861 (2017).
- [37] S. HUANG, N. PAPERNOT, I. GOODFELLOW, Y. DUAN, AND P. ABBEEL, *Adversarial attacks on neural network policies*, arXiv preprint arXiv:1702.02284, (2017).
- [38] Y. HUANG, Y. CHENG, A. BAPNA, O. FIRAT, D. CHEN, M. CHEN, H. LEE, J. NGIAM, Q. V. LE, Y. WU, AND Z. CHEN, *Gpipe: Efficient training of giant neural networks using pipeline parallelism*, in Advances in Neural Information Processing Systems 32, Curran Associates, Inc., 2019, pp. 103–112.
- [39] M. HUH, A. LIU, A. OWENS, AND A. A. EFROS, *Fighting fake news: Image splice detection via learned self-consistency*, arXiv preprint arXiv:1805.04096, (2018).
- [40] Z. HUSSAIN, C. THOMAS, M. ZHANG, Z. AGHA, X. ZHANG, N. ONG, K. YE, AND A. KOVASHKA, *Automatic understanding of image and video advertisements*, Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, 2017-Janua (2017).
- [41] F. N. IANDOLA, M. W. MOSKEWICZ, K. ASHRAF, S. HAN, W. J. DALLY, AND K. KEUTZER, *Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size*, CoRR, abs/1602.07360 (2016).
- [42] —, *Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size*, CoRR, abs/1602.07360 (2016).
- [43] A. IGNATOV, R. TIMOFTE, W. CHOU, K. WANG, M. WU, T. HARTLEY, AND L. V. GOOL, *AI benchmark: Running deep neural networks on android smartphones*, CoRR, abs/1810.01109 (2018).
- [44] A. INC., *Devicecheck documentation*, 2019. <https://developer.apple.com/documentation/devicecheck>.
- [45] —, *Vendorid documentation*, 2019. <https://developer.apple.com/documentation/uikit/uidevice/1620059-identifierforvendor>.
- [46] A. P. INC., *Adblock Plus for Chrome support*. <https://adblockplus.org/>.

- [47] B. C. INC., *Combo ip/bin checker*. <https://www.bincodes.com/ip-bin-checker/>.
- [48] B. S. INC., *Brave - Secure, Fast & Private Web Browser with Adblocker*. <https://brave.com/>.
- [49] C. INC., *Id document verification*. <https://help.coinbase.com/en/coinbase/getting-started/authentication-and-verification/identity-verification.html>.
- [50] E. INC., *ExpressVPN*. <https://www.expressvpn.com/>.
- [51] E. L. INC., *EasyList*. <https://easylist.to>.
- [52] F. INC., *Facebook Ad placements*. <https://www.facebook.com/business/help/407108559393196>.
- [53] G. INC., *Chromium Graphics*. <https://www.chromium.org/developers/design-documents/chromium-graphics>.
- [54] G. INC., *Ghostery - Privacy Ad Blocker*. <https://www.ghostery.com/>.
- [55] G. INC., *Puppeteer: Headless Chrome Node API*. <https://github.com/GoogleChrome/puppeteer>.
- [56] ———, *Key and id attestation*, 2019. <https://source.android.com/security/keystore/attestation>.
- [57] ———, *Safetynet attestation api*, 2021. <https://developer.android.com/training/safetynet/attestation>.
- [58] L. INC., *Apply for lime access*. <https://www.fountain.com/limebike/apply/united-states-limeaccess>.
- [59] ———, *Lime access: Mobility for all*. <https://www.li.me/community-impact>.
- [60] M. INC., *Firefox's Enhanced Protection*. <https://blog.mozilla.org/blog/2019/09/03/todays-firefox-blocks-third-party-tracking-cookies-and-cryptomining-by-default/>.
- [61] N. N. G. INC., *Annoying online ads do cost business*. <https://www.mngroup.com/articles/annoying-ads-cost-business/>.
- [62] S. INC., *Selenium: Web Browser Automation*. <https://www.seleniumhq.org>.
- [63] S. INC., *Similar Web*. <https://www.similarweb.com/>.
- [64] S. INC., *Skia Graphics Library*. <https://skia.org/>.
- [65] U. INC., *uBlock Origin*. <https://www.ublock.org/>.
- [66] V. INC., *Why people hate ads?* <https://www.vieodesign.com/blog/new-data-why-people-hate-ads/>.
- [67] U. IQBAL, P. SNYDER, S. ZHU, B. LIVSHITS, Z. QIAN, AND Z. SHAFIQ, *Adgraph: A graph-based approach to ad and tracker blocking*, in 2020 IEEE Symposium on Security and Privacy (SP), 2020, pp. 763–776.
- [68] M. ISAAC, *Uber's c.e.o. plays with fire*, April 2017. <https://www.nytimes.com/2017/04/23/technology/travis-kalanick-pushes-uber-and-himself-to-the-precipice.html>.
- [69] P. ISOLA, J.-Y. ZHU, T. ZHOU, AND A. A. EFROS, *Image-to-image translation with conditional adversarial networks*, CVPR, (2017).
- [70] M. JETHANI, *Implement hide-if-contains-snippet*. <https://issues.adblockplus.org/ticket/7088/>.
- [71] H. JIANG AND O. NACHUM, *Identifying and correcting label bias in machine learning*, CoRR, abs/1901.04966 (2019).

- [72] JUNIPER RESEARCH, *Online Payment Fraud Whitepaper*. <http://www.experian.com/assets/decision-analytics/white-papers/juniper-research-online-payment-fraud-wp-2016.pdf>.
- [73] H. KANNAN, A. KURAKIN, AND I. GOODFELLOW, *Adversarial Logit Pairing*, arXiv preprint arXiv:1803.06373, (2018).
- [74] N. KARAPANOS, C. MARFORIO, C. SORIENTE, AND S. CAPKUN, *Sound-proof: Usable two-factor authentication based on ambient sound*, in 24th USENIX Security Symposium (USENIX Security 15), Washington, D.C., 2015, USENIX Association, pp. 483–498.
- [75] N. KILBERTUS, M. ROJAS-CARULLA, G. PARASCANDOLO, M. HARDT, D. JANZING, AND B. SCHÖLKOPF, *Avoiding discrimination through causal reasoning*, in Proceedings of the 31st International Conference on Neural Information Processing Systems, NIPS’17, Red Hook, NY, USA, 2017, Curran Associates Inc., p. 656–666.
- [76] J. Z. KOLTER AND E. WONG, *Provable defenses against adversarial examples via the convex outer adversarial polytope*, arXiv preprint arXiv:1711.00851, 1 (2017).
- [77] A. KRIZHEVSKY, I. SUTSKEVER, AND G. 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, USA, 2012, Curran Associates Inc.
- [78] A. KURAKIN, I. GOODFELLOW, AND S. BENGIO, *Adversarial machine learning at scale*, arXiv preprint arXiv:1611.01236, (2016).
- [79] N. LAB, *More than half of local independent news sites are selling sponsored content*. <https://www.niemanlab.org/2016/06/more-than-half-of-local-independent-online-news-sites-are-now-selling-sponsored-content-survey/>.
- [80] W.-H. LEE, X. LIU, Y. SHEN, H. JIN, AND R. B. LEE, *Secure pick up: Implicit authentication when you start using the smartphone*, in Proceedings of the 22Nd ACM on Symposium on Access Control Models and Technologies, SACMAT ’17 Abstracts, New York, NY, USA, 2017, ACM, pp. 67–78.
- [81] Z. LI, K. ZHANG, Y. XIE, F. YU, AND X. WANG, *Knowing your enemy: Understanding and detecting malicious web advertising*, in Proceedings of the 2012 ACM Conference on Computer and Communications Security, CCS ’12, New York, NY, USA, 2012, ACM, pp. 674–686.
- [82] T. LIBERT, *Exposing the hidden web: An analysis of third-party HTTP requests on 1 million websites*, CoRR, abs/1511.00619 (2015).
- [83] J. LIU, C. SHI, Y. CHEN, H. LIU, AND M. GRUTESER, *Cardiocam: Leveraging camera on mobile devices to verify users while their heart is pumping*, in Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys ’19, New York, NY, USA, 2019, Association for Computing Machinery, p. 249–261.
- [84] M.-Y. LIU, T. BREUEL, AND J. KAUTZ, *Unsupervised image-to-image translation networks*, in Advances in neural information processing systems, 2017, pp. 700–708.

- [85] S. LIU, Y. LIN, Z. ZHOU, K. NAN, H. LIU, AND J. DU, *On-demand deep model compression for mobile devices: A usage-driven model selection framework*, in Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys '18, New York, NY, USA, 2018, Association for Computing Machinery, p. 389–400.
- [86] W. LIU, D. ANGUELOV, D. ERHAN, C. SZEGEDY, S. REED, C.-Y. FU, AND A. C. BERG, *Ssd: Single shot multibox detector*, in Computer Vision – ECCV 2016, B. Leibe, J. Matas, N. Sebe, and M. Welling, eds., Cham, 2016, Springer International Publishing, pp. 21–37.
- [87] A. MADRY, A. MAKELOV, L. SCHMIDT, D. TSIPRAS, AND A. VLADU, *Towards deep learning models resistant to adversarial attacks*, in International Conference on Learning Representations, 2018.
- [88] S. MARE, A. MOLINA-MARKHAM, C. CORNELIUS, R. A. PETERSON, AND D. KOTZ, *ZEBRA: zero-effort bilateral recurring authentication*, in 2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18–21, 2014, 2014, pp. 705–720.
- [89] M. MERENDA, C. PORCARO, AND D. IERO, *Edge machine learning for ai-enabled iot devices: A review*, Sensors, 20 (2020), p. 2533.
- [90] D. NARAYANAN, A. HARLAP, A. PHANISHAYEE, V. SESHADRI, N. R. DEVANUR, G. R. GANGER, P. B. GIBBONS, AND M. ZAHARIA, *Pipedream: Generalized pipeline parallelism for dnn training*, in Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP '19, New York, NY, USA, 2019, Association for Computing Machinery, p. 1–15.
- [91] I. NUTTALL, *Binlist: An open-source list of bank bin/iin numbers*. <https://github.com/iannuttall/binlist-data>.
- [92] S. S. OGDEN AND T. GUO, *Characterizing the deep neural networks inference performance of mobile applications*, 2019.
- [93] N. PAPERNOT, P. D. MCDANIEL, AND I. J. GOODFELLOW, *Transferability in machine learning: from phenomena to black-box attacks using adversarial samples*, CoRR, abs/1605.07277 (2016).
- [94] N. PAPERNOT, P. D. MCDANIEL, I. J. GOODFELLOW, S. JHA, Z. B. CELIK, AND A. SWAMI, *Practical black-box attacks against deep learning systems using adversarial examples*, CoRR, abs/1602.02697 (2016).
- [95] N. PAPERNOT, P. D. MCDANIEL, S. JHA, M. FREDRIKSON, Z. B. CELIK, AND A. SWAMI, *The limitations of deep learning in adversarial settings*, CoRR, abs/1511.07528 (2015).
- [96] K. PATEL, H. HAN, A. K. JAIN, AND G. OTT, *Live face video vs. spoof face video: Use of moiré patterns to detect replay video attacks*, in 2015 International Conference on Biometrics (ICB), May 2015, pp. 98–105.
- [97] D. PATHAK, P. KRAHENBUHL, J. DONAHUE, T. DARRELL, AND A. A. EFROS, *Context encoders: Feature learning by inpainting*, in Proceedings of the IEEE conference on computer vision and pattern recognition, 2016, pp. 2536–2544.
- [98] PAYPAL, *Card.io: Scan credit cards in your mobile app*. <https://card.io>.

- [99] R. B. PIPES, *Moiré analysis of the interlaminar shear edge effect in laminated composites*, Journal of Composite Materials, (1971).
- [100] A. PLUS, *Advantage ABP successfully blocking ads on Facebook*. <https://adblockplus.org/blog/advantage-abp-successfully-blocking-ads-on-facebook-for-1-year-1-month-and-1-day>.
- [101] ———, *Ping pong with facebook*. <https://adblockplus.org/blog/ping-pong-with-facebook>.
- [102] E. PUJOL, T. BERLIN, O. HOHLFELD, A. FELDMANN, AND T. BERLIN, *Annoyed users: Ads and ad-block usage in the wild*.
- [103] X. RAN, H. CHEN, X. ZHU, Z. LIU, AND J. CHEN, *Deepdecision: A mobile deep learning framework for edge video analytics*, in IEEE INFOCOM 2018 - IEEE Conference on Computer Communications, 2018, pp. 1421–1429.
- [104] J. REDMON, S. K. DIVVALA, R. B. GIRSHICK, AND A. FARHADI, *You only look once: Unified, real-time object detection*, 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), (2016), pp. 779–788.
- [105] S. REN, K. HE, R. GIRSHICK, AND J. SUN, *Faster r-cnn: Towards real-time object detection with region proposal networks*, in Advances in Neural Information Processing Systems 28, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, eds., Curran Associates, Inc., 2015, pp. 91–99.
- [106] ———, *Faster r-cnn: Towards real-time object detection with region proposal networks*, in Advances in Neural Information Processing Systems 28, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, eds., Curran Associates, Inc., 2015, pp. 91–99.
- [107] K. ROGERS, *Facebook's Arms Race with Adblockers Continues to Escalate*. https://motherboard.vice.com/en_us/article/7xydvx/facebooks-arms-race-with-adblockers-continues-to-escalate.
- [108] SAM TOLOMEI, *Shrinking APKs, growing installs*. <https://medium.com/googleplaydev/shrinking-apks-growing-installs-5d3fcba23ce2>.
- [109] SAMSUNG INC., *What is FPS?* <https://www.samsung.com/global/galaxy/what-is/frames-per-second/>.
- [110] M. SANDLER, A. G. HOWARD, M. ZHU, A. ZHMOGINOV, AND L. CHEN, *Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation*, CoRR, abs/1801.04381 (2018).
- [111] M. SATYANARAYANAN, *The emergence of edge computing*, Computer, 50 (2017), pp. 30–39.
- [112] N. SCAIFE, J. BOWERS, C. PEETERS, G. HERNANDEZ, I. N. SHERMAN, P. TRAYNOR, AND L. ANTHONY, *Kiss from a rogue: Evaluating detectability of pay-at-the-pump card skimmers*, in 2019 IEEE Symposium on Security and Privacy (SP), 2019, pp. 1000–1014.
- [113] N. SCAIFE, C. PEETERS, AND P. TRAYNOR, *Fear the reaper: Characterization and fast detection of card skimmers*, in 27th USENIX Security Symposium (USENIX Security 18), Baltimore, MD, 2018, USENIX Association, pp. 1–14.

- [114] N. SCAIFE, C. PEETERS, C. VELEZ, H. ZHAO, P. TRAYNOR, AND D. ARNOLD, *The cards aren't alright: Detecting counterfeit gift cards using encoding jitter*, in 2018 IEEE Symposium on Security and Privacy (SP), 2018, pp. 1063–1076.
- [115] R. R. SELVARAJU, A. DAS, R. VEDANTAM, M. COGSWELL, D. PARIKH, AND D. BATRA, *Grad-cam: Why did you say that? visual explanations from deep networks via gradient-based localization*, CoRR, abs/1610.02391 (2016).
- [116] A. SENTINEL, *Adblock Plus, Sentinel*, 2018.
- [117] J. STAPLETON AND R. S. POORE, *Tokenization and other methods of security for cardholder data*, Information Security Journal: A Global Perspective, 20 (2011), pp. 91–99.
- [118] G. STOREY, D. REISMAN, J. R. MAYER, AND A. NARAYANAN, *The future of ad blocking: An analytical framework and new techniques*, CoRR, abs/1705.08568 (2017).
- [119] STRIPE, *Disputes and fraud*. <https://stripe.com/docs/disputes>.
- [120] A. SUPPORT, *I'm seeing "sponsored" posts on Facebook again*. <https://help.getadblock.com/support/solutions/articles/6000143593-i-m-seeing-sponsored-posts-on-facebook-again>.
- [121] S. SUWAJANAKORN, S. M. SEITZ, AND I. KEMELMACHER-SHLIZERMAN, *Synthesizing obama: Learning lip sync from audio*, ACM Trans. Graph., 36 (2017), pp. 95:1–95:13.
- [122] F. TRAMÈR, P. DUPRÉ, G. RUSAK, G. PELLEGRINO, AND D. BONEH, *Ad-versarial: Defeating perceptual ad-blocking*, CoRR, abs/1811.03194 (2018).
- [123] F. TRAMÈR, A. KURAKIN, N. PAPERNOT, I. GOODFELLOW, D. BONEH, AND P. MCDANIEL, *Ensemble adversarial training: Attacks and defenses*, 2020.
- [124] E. UZUN, S. CHUNG, I. ESSA, AND W. LEE, *rtcaptcha: A real-time captcha based liveness detection system*, in NDSS, 02 2018.
- [125] E. UZUN, S. P. H. CHUNG, I. ESSA, AND W. LEE, *rtcaptcha: A real-time CAPTCHA based liveness detection system*, in NDSS, The Internet Society, 2018.
- [126] A. VASTEL, P. SNYDER, AND B. LIVSHITS, *Who Filters the Filters: Understanding the Growth, Usefulness and Efficiency of Crowdsourced Ad Blocking*, arXiv preprint arXiv:1810.09160, (2018).
- [127] VERIZON, *What is the difference between 3G, 4G and 5G?* <https://www.verizon.com/about/our-company/5g/difference-between-3g-4g-5g>.
- [128] VISA, *Counterfeit fraud at U.S. chip-enabled merchants down 70%*. <https://usa.visa.com/visa-everywhere/security/visa-chip-card-stats.html>.
- [129] R. J. WANG, X. LI, AND C. X. LING, *Pelee: A real-time object detection system on mobile devices*, in Proceedings of the 32nd International Conference on Neural Information Processing Systems, NIPS'18, Red Hook, NY, USA, 2018, Curran Associates Inc., p. 1967–1976.

- [130] C. WU, D. BROOKS, K. CHEN, D. CHEN, S. CHOUDHURY, M. DUKHAN, K. HAZELWOOD, E. ISAAC, Y. JIA, B. JIA, T. LEYVAND, H. LU, Y. LU, L. QIAO, B. REAGEN, J. SPISAK, F. SUN, A. TULLOCH, P. VAJDA, X. WANG, Y. WANG, B. WASTI, Y. WU, R. XIAN, S. YOO, AND P. ZHANG, *Machine learning at facebook: Understanding inference at the edge*, in 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA), 2019, pp. 331–344.
- [131] X. XING, W. MENG, B. LEE, U. WEINSBERG, A. SHETH, R. PERDISCI, AND W. LEE, *Understanding malvertising through ad-injecting browser extensions*, in Proceedings of the 24th international conference on world wide web, International World Wide Web Conferences Steering Committee, 2015.
- [132] M. XU, J. LIU, Y. LIU, F. X. LIN, Y. LIU, AND X. LIU, *A first look at deep learning apps on smartphones*, in The World Wide Web Conference, WWW '19, New York, NY, USA, 2019, Association for Computing Machinery, p. 2125–2136.
- [133] S. K. YARLAGADDA, D. GÜERA, P. BESTAGINI, F. M. ZHU, S. TUBARO, AND E. J. DELP, *Satellite image forgery detection and localization using GAN and one-class classifier*, CoRR, abs/1802.04881 (2018).
- [134] K. YE AND A. KOVASHKA, *ADVISE: Symbolism and external knowledge for decoding advertisements*, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 11219 LNCS (2018).
- [135] M. ZANIN, M. ROMANCE, S. MORAL, AND R. CRIADO, *Credit card fraud detection through parenclitic network analysis*, CoRR, abs/1706.01953 (2017).
- [136] P. ZHOU, X. HAN, V. I. MORARIU, AND L. S. DAVIS, *Learning rich features for image manipulation detection*, in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2018, pp. 1053–1061.