Boxer: Preventing fraud by scanning credit cards

Zainul Abi Din and Hari Venugopalan, *UC Davis;* Jaime Park, *Bouncer Technologies;* Andy Li, *Segment;* Weisu Yin, *UC Davis;* Haohui Mai, *Hengmuxing Technologies;* Yong Jae Lee, *UC Davis;* Steven Liu, *Bouncer Technologies;* Samuel T. King, *UC Davis and Bouncer Technologies*

https://www.usenix.org/conference/usenixsecurity20/presentation/din

This paper is included in the Proceedings of the 29th USENIX Security Symposium.

August 12–14, 2020

978-1-939133-17-5

Open access to the Proceedings of the 29th USENIX Security Symposium is sponsored by USENIX.

Boxer: Preventing fraud by scanning credit cards

Zainul Abi Din
UC Davis

Hari Venugopalan
UC Davis

Jaime Park
Bouncer Technologies

Andy Li
Segment

Weisu Yin
UC Davis

Haohui Mai
Hengmuxing Technologies

Yong Jae Lee
UC Davis

Steven Liu
Bouncer Technologies

Samuel T. King
UC Davis and Bouncer Technologies

Abstract

Card-not-present credit card fraud costs businesses billions of dollars a year. In this paper, we present Boxer, a mobile SDK and server that enables apps to combat card-not-present fraud by scanning cards and verifying that they are genuine. Boxer analyzes the images from these scans, looking for telltale signs of attacks, and introduces a novel abstraction on top of modern security hardware for complementary protection.

Currently, 323 apps have integrated Boxer, and tens of them have deployed it to production, including some large, popular, and international apps, resulting in Boxer scanning over 10 million real cards already. Our evaluation of Boxer from one of these deployments shows ten cases of real attacks that our novel hardware-based abstraction detects. Additionally, from the same deployment, without letting in any fraud, Boxer's card scanning recovers *89%* of the good users whom the app would have blocked. In another evaluation of Boxer, we run our image analysis models against images from real users and show an accuracy of 96% and 100% on the two models that we use.

1 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 [13, 28], 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 [51]. 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 [47]). Thus, credit card companies incentivize app builders financially 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 [12]. Plus, 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, also has the potential to easily verify legitimate users.

Unfortunately, card scanners designed for adding credit cards to an account are not designed for security. In our evaluation, we run a myriad of tests against commercially-deployed card scanners and find that none of them can stop a text editor with the credit card number written on it scanned off a computer screen – the least sophisticated attack we evaluate. In our experience with Boxer in production environments, we have seen photoshopped cards, cards scanned off of computer and phone screens, and a credit-card-specific version of credential stuffing where attackers entered hundreds of credit card numbers on the same device to detect which ones were valid. Card scanners designed for adding credit cards to an account are woefully ill equipped to deal with any of these

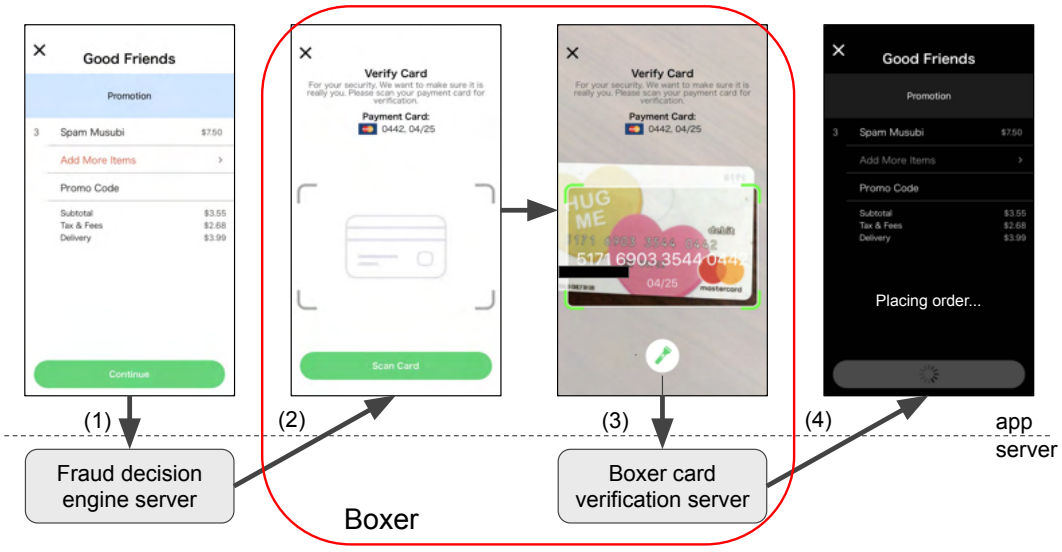

Figure 1: This figure shows how a food delivery app can use Boxer to verify a credit card for a suspicious transaction. In this example, the food delivery app (1) detects a suspicious transaction. Rather than blocking it, (2) they forward the user to Boxer's card scanner. Boxer's card scanner scans the 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.

types of attacks.

In this paper, we present Boxer, a new system for deterring card-not-present fraud. The first part of Boxer is a card scanner that we designed from the ground up for security. The wide deployment of card scanning suggests that it already provides a good user experience, thus our focus is on the techniques to verify that a card scanned by a user is in fact a genuine physical credit card. To the best of our knowledge, we are the first to show how to verify cards from a scan.

The second part of Boxer is a secure counter that is based on security hardware found on modern smartphones. Our 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 our secure counters maintain end-user privacy.

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

Our work has already started to have an impact in practice with major apps for e-commerce, bike rentals, airlines, deliveries, and payments integrating Boxer into their apps. **Our basic card scanner has already scanned more than 10 million cards in production systems running within large international apps**. In addition, our secure counting abstraction and advanced card scanner are running in several large apps and successfully detecting fraud.

2 Motivating example

This section walks through an autobiographical motivating example of card-not-present fraud and how Boxer can help defend against it.

Mallory is a fraudster. She buys stolen credit card numbers from other attackers, which they send to her in a text file [8, 12, 21, 46]. At first, she uses these stolen card numbers to buy food from a food delivery app, called *Foodie*, for herself and her friends. Then, she sees an opportunity to go into business monetizing her stolen credit card numbers.

To monetize stolen credit cards, Mallory acts as an agent service selling food delivery at a heavy discount. In this scheme she collects money from the person who wants food and "pays" Foodie using stolen credit cards, leaving Foodie stuck with the bill [11]. Given the profits from this attack, Mallory recruits a team to help and as they scale their enterprise, Foodie is now losing nearly 5% of their revenue to card-not-present credit card fraud.

Foodie first becomes aware of their fraud problem when Visa reaches out to Foodie due to their chargeback ratio going above 1%. At a chargeback ratio above 1%, Foodie is at risk of having Visa remove them from the payment network [52], effectively killing Foodie's growing business.

Foodie acts by hiring a data scientist, Ari, to help detect fraud. Ari crafts some business rules [49] to identify the most egregious transactions, and then trains a machine learning model to generalize to other transactions [45]. As Ari's model hits production, fraud plummets and order is restored at Foodie.

However, this calm is short lived as Ari only measures his model's impact on chargebacks and *not* on the users that his model flags incorrectly [10]. It is not until Ari's model disables one of Foodie's investors that Foodie starts to look at the impact of incorrect model decisions. Upon further investigation, they realize that they are losing more money due to lost business from blocking legitimate transactions that Ari's model flags than they would have lost from chargebacks.

To help with their false positive problem, Foodie hires Brie, who had been working on stopping fake accounts at a large social network. Brie knows that by providing users with a way to verify themselves automatically she can recover almost all the false positives while still preventing most of the fraud [32]. Brie uses the Boxer "scan your card" challenge that asks suspicious users to scan their credit card on their phone to proceed (Figure 1). She knows that most legitimate users have their card in their wallet, whereas attackers like Mallory just have a text file with card numbers, making it easy for good users to pass but hard for attackers. After Brie launched this challenge, Foodie recovers over 80% of their false positives, while keeping general fraud rates low.

Although "scan your card" deters many attackers, Mallory evolves her attacks to evade or deceive this challenge. However, Boxer's holistic approach comprising of advanced scanning and secure counting limits these attacks, making it difficult for Mallory to commit fraud, while continuing to be easy for good users. We defer the discussion of possible attacks and Boxer's countermeasures against them to the remainder of the paper.

3 Threat model, assumptions, and goal

In our 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 with the stolen information.

We consider attacks that vary across a broad range of attack sophistication where the key differences lie in the technical sophistication, physical, and monetary resources available, as well as knowledge of the banking system. We consider 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*. They can carry out large scale automatic attacks. Other attackers use humans and real devices to carry out credit card fraud, relying on human scale to attempt fraudulent transactions one at a time, who we consider to be *unsophisticated attackers*.

Our goal is to stop attacks from both sophisticated as well as unsophisticated attackers. However, our goal is *not* to stop *all* fraudulent transactions, but rather to make stolen credit card attacks economically infeasible across this broad spectrum of attacker sophistication.

4 Boxer design principles and overview

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

Our first general defensive philosophy is to compose complementary defenses. Financial fraud is diverse, ranging from groups of humans carrying out attacks manually using real iPhones to full-blown automation, bots, and machine learning. Rather than try to devise a single defense to stop them all, we compose several complementary pieces to make an overall defensive system. We strive to have one component cover the weaknesses or blind spots of another.

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

4.1 Boxer design principles

In this section we describe our general design for scanning credit cards to verify that they are genuine. Although our focus is on scanning credit cards, we expect these general principles to apply to similar problems, such as scanning IDs, selfie checks, or verifying utility bills. Our design has five general principles that guide our implementation.

Principle 1: Scan the card to extract relevant details and check them against what the app has on record. In Boxer, we scan the credit card number using optical character recognition (OCR, Section 6.1) and check that 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 6.2). 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 empower attackers to tamper 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 6.3). By detecting a screen, we can prevent one simple avenue for producing and scanning fake card images.

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

Figure 2: Comparing defensive techniques. In this table, we compare 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.

Principle 4: Associate attacker activities with items that are expensive. In Boxer, we track activities and increment a secure counter when they occur on the same device (Section 7). 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. In Boxer, we put a premium on end-user privacy by only using device identifiers that users can reset (Section 7) and by running our machine learning models on the client (Section 8).

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 fraud (Figure 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, we use both card scanning and secure counting together to help make up for the shortcomings of the other.

5 Image analysis motivation

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 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 [31, 38, 48], to the best of our knowledge, the problem of creating fake credit and debit cards has not been studied. To answer if creating realistic fake card images is possible and whether existing methods can detect them, we design and implement Fugazi, a new, automatic system for creating realistic fake card images.

The inability of current state-of-the-art image tampering detection techniques to detect Fugazi influences the eventual design of our image analysis pipeline.

5.1 Fugazi

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

Overall, we have three goals with Fugazi. First, we want to create a dataset under a controlled setting where we can filter out specific artifacts from the camera and other telltale signs of automation that the models might learn as a shortcut for learning the overall task. Second, we want to push the boundaries of creating fake images in the above controlled setting with the goal to create imperceptible fakes for humans and machines alike. Third, we want our methods to scale, since the existing image manipulation datasets [35, 53] contain only hundreds of images not enough to train deeper models.

Figure 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 a modified generative adversarial network (GAN) system pix2pix [27] 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 4). Finally (4) Fugazi uses Poisson blending and Lanczos resampling to minimize artifacts that indicate digital tampering.

This version of Fugazi represents our fourth iteration on its design where our informal goal was to keep working on it until the authors of this paper could not distinguish between fake and real cards. Our first iteration used traditional computer vision algorithms and there were always clear artifacts. Our 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

Figure 3: Fugazi's basic process for creating fake card images. The process uses four steps and combines traditional image manipulation techniques with deep learning.

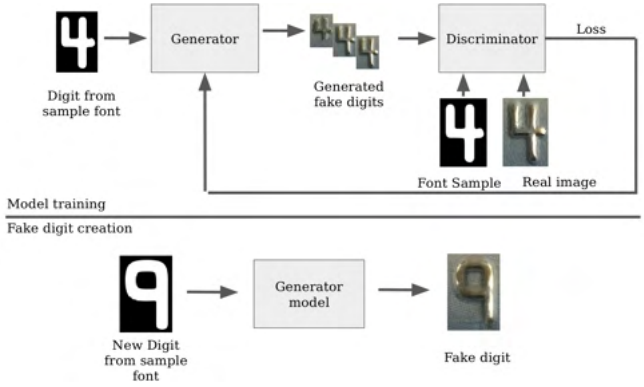

Figure 4: Fugazi digit generation process. In the above example we want to inject digit *9* in place of digit *4* in the original card. We find the digit *4* in the sample font. We then train the model to reproduce the original digit from the sample font digit. Afterwards, we use the trained model to create the textured version of digit *9* from the corresponding font digit. The model reproduces lighting, shade, pose close to the original digit.

visual artifacts for more complicated textures. Our third iteration also used image-to-image translation, but only for the region of the card that contained the number. At first this technique produced great looking numbers but had a clear bounding box around the number. To remove the bounding box, we used yet another image-to-image translation step specifically to smooth out the bounding box. This third iteration was the first to produce card images that we were unable to distinguish between fake and real, but it was too complex and took too long to create new fakes, which motivated our ultimate use of traditional vision algorithms combined with small and well defined image-to-image translation tasks.

Figure 5 shows two examples of fake cards generated by Fugazi and Figure 16 in the Appendix shows more examples.

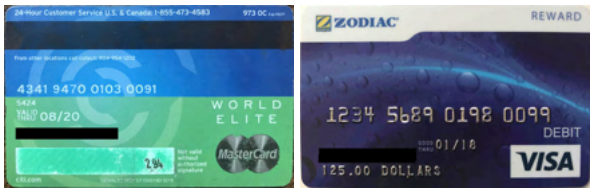

Figure 5: Examples of credit card images generated by Fugazi

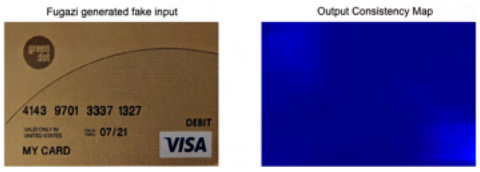

Figure 6: Fake sample and corresponding consistency map generated by the self-consistency based deep learning model [22]. Since the image is fake, and the map is uniform, we can see that Fugazi is able to overcome the proposed model.

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 [56]. To answer the question of whether or not machine learning is sufficient to detect tampered images, in this section we evaluate fakes generated by Fugazi, which we consider as a proxy for high quality fakes, against general image tampering detection models that achieve state-of-the-art performance on benchmark image manipulation datasets [22], [56]. We describe more experiments attempting to detect Fugazi in Appendix B.

5.2.1 Evaluating Fugazi with state-of-the-art methods

We employed some of the existing state-of-the-art deep learning and traditional image forensics algorithms to detect Fugazi generated samples.

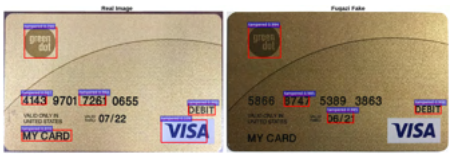

Figure 7: Tampering regions determined by the Faster R-CNN based model [56] on a real image (left), and Fugazi fake (right). Not only does the model fail to detect the tampered regions in the Fugazi fake, it also mistakenly detects untampered regions on both images.

We first tested the model proposed on self-consistency for detecting fake images [22]. This model produces a consistency map that indicates regions within the card that attackers have tampered. Figure 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.

We also evaluated Fugazi against a modified Faster R-CNN model [40] proposed in recent work by Zhou, et al. [56]. Figure 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, we tested 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. We describe the techniques and results in Appendix B.1 and Appendix B.2.

5.2.2 Further difficulties with practical deployments

All the fake image detection techniques we test 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.

5.3 Where do we go from here?

We have shown that 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. We augment machine learning 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 our knowledge of the design of credit cards (Section 6.2). Since we design the rules 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 stops attacks before they cause any damage. For more advanced attackers we focus on other aspects of the overall attack.

Our secure counting abstraction (Section 7) 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 our detection system can use to identify bad actors. Our screen detection model (Section 6.3) detects card images that attackers scan off screens, a common technique employed by attackers who use real phones and the real app to carry out fraud.

6 Image analysis

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 7) 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 we outline in Section 4.1. Section 8 discusses our implementation.

6.1 Optical character recognition

OCR is how we extract a card number out of a video stream when a user scans their card. In Boxer, OCR serves as the baseline of our defense where we use this scanned card number to match against the card number that the app has on record. Although, unsophisticated fake cards can bypass OCR by itself (Section 9.7), it will deter some attackers and acts as a first line of defense, feeding the card's BIN into our more advanced image analysis stages.

Perhaps ironically, we use Fugazi fakes (Section 5.1) to train Boxer's OCR system. Our design of Fugazi makes generating synthetic labelled data for training trivial.

We cast OCR as a special case of object detection, where we train a smaller more constrained version of a traditional object detector tailored specifically for credit and debit cards.

Figure 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 our data of the card BIN to identify any inconsistencies.

6.2 BIN consistency and expectation check

Our 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). Our goal is to train a model that we can use to verify that a card "looks" like a card from that BIN and 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.

Our first iteration of BIN consistency was a BIN/Texture check where the model identifies issuing banks from the card image 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 we would need to get new samples often. Given our 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 we use contains 348,925 unique BINs worldwide [2], 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, we cast BIN consistency check 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.), type of card (debit or credit), are finite and persistent regardless of the background texture used to print the card. This ensures we can uniquely identify a BIN from a combination of these objects independent of the background texture.

Our current 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 our knowledge of card

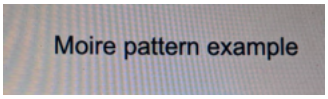

Figure 9: Moiré patterns observed on capturing a laptop screen on a mobile phone. These patterns are an inherent aliasing effect that arise from differences in spatial frequency of the laptop screen and the mobile camera.

BINs to identify fraud. Figure 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 sends 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. Note: We do *not* send any part of the input image 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 our analysis on higher level and common features, we can train an effective object detection model using less data. Also, we can use our server to collect BIN and object data mappings to serve as the ground truth for the mapping between a BIN and the objects and locations that they tend to have.

6.3 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 we focus on screens since we have observed attackers attempt to do so in the wild (Section 9.4).

We observe that there are telltale signs of images scanned off screens and seek to use them. These signs include screen edges or reflections, that attackers can carefully avoid, and more intrinsic signs such as Moiré patterns [39] which are much harder to avoid.

Moiré patterns, as shown in Figure 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 [37].

We detect these signs by training a binary image classifier.

7 Secure counting abstraction

Boxer enables app builders to count events that it associates with hardware devices. This section describes our 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.

We recognize that we are using Apple's hardware mechanism in a way that they did not design for, but we find that it is close to what we would want. The Appendix discusses our experience deploying the counting abstraction, limitations, and suggestions for how Apple and Google could better support our counting abstraction.

7.1 Why counting?

Before we describe how we count, we 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 [24] and Google [25] provide app builders with solid mechanisms for ensuring that a request comes from a legitimate iOS or Android device. Some attackers even carry out this hardware-based technique at scale [34] due to these limitations on their attacks.

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 accounts using the same device. If app builders can count these events on a per-device basis, they can detect the attacks, as we show in Section 9.3.

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 [15, 26]. Existing industry solutions to counting that we have first-hand experience with suffer from these problems. Our secure counter is novel because it respects end-user privacy while still empowering apps to maintain counts even across resets.

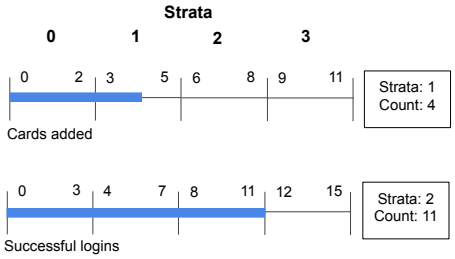

Figure 10: Counts and their associated strata. This figure shows counts for cards added and successful logins and their corresponding strata.

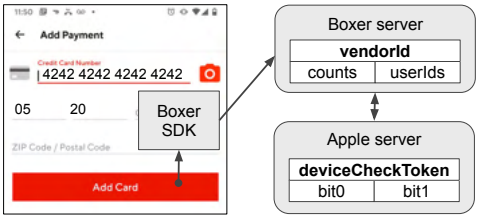

Figure 11: The architecture for the secure counting abstraction. This figure shows how Boxer updates counts after an app adds a card. The app calls into the Boxer SDK, which calls the Boxer server, where Boxer maintains a database of counts. The Boxer server manages the DeviceCheck bits by accessing Apple's servers on behalf of the app.

7.2 Secure counting basics

At the heart of our secure counting abstraction is Apple's DeviceCheck abstraction [24]. DeviceCheck uses hardware-backed tokens stored on a device, which our server uses to query two bits per device from Apple's servers. DeviceCheck is supported 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, we use them to define a range of possible counts. Figure 10 shows a device where we are tracking cards added and successful logins. For this app, the app builder expects a maximum of 11 cards added and 15 successful logins, which Boxer divides into four sections, or strata. We divide the counts by four so that we can represent each of the four strata using Apple's two hardware bits. In our example, this device has a count of four cards added and eleven successful logins, which map to strata one and two respectively.

The software counts and hardware strata complement each other where we use software counts in the common case where the user maintains the same *vendorId* (Apple's privacy-

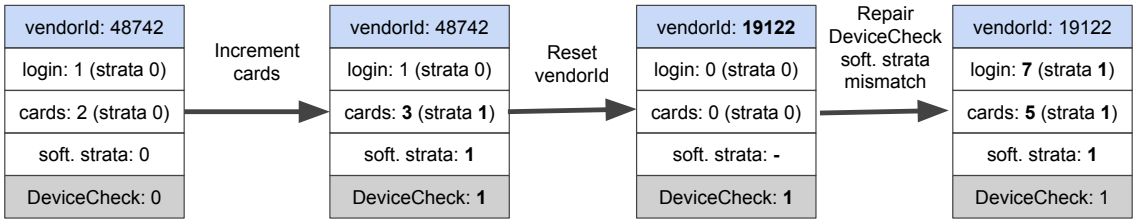

Figure 12: Example of Boxer's secure counting system. In this figure, we show the counting system in four different states with three transitions between them. In this example, Boxer is counting cards added and logins, and tracking these on a per vendorId basis.

friendly deviceId abstraction [5]), but hardware strata to recover lost count values when we see a device reset. For attackers that reset their device, our counting abstraction provides monotonically increasing count values, but for legitimate users who reset their device, by dividing the counts up into four strata we limit the amount that our counts will increase on a device reset to avoid falsely flagging good users.

Figure 11 shows our 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 our 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. Our 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 us to access Apple's servers on their behalf. Subsequently, the app can query counts from Boxer's server.

7.3 Counting and inconsistencies

Figure 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. We define 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, we guarantee 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.*

Our rules for counting are:

- The strata for a count = floor(count * 4 / max_count).
- The software strata = max(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.

Our system also handles counts where the maximum count is less than four and we handle the case where the attacker moves vendorIds between attacker-controlled devices, but we omit the details from this paper.

Initialization edge case We handle 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 we see such a configuration, we push 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.

8 Implementation

This section discusses our overall implementation of Boxer. We discuss our overall implementation and our ML based image analysis pipeline.

In general, our system includes libraries that run on Android and iOS that app builders can put into their apps. For Android we use the standard jCenter repository to deploy our library, and for iOS we use 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.

Our system also includes a server portion that consumes the output of our 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.

Our goal for our 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 we expect for card from that BIN, and detect any cards scanned off computer screens. In our current implementation, we use 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.

We run models client-side because it provides stronger privacy by virtue of *not* sending images of cards to our 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 we sent images to a server.

9 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. We evaluate Boxer on its performance against real attacks against these deployments (Sections 9.2, 9.3, 9.4, 9.6) and follow up with a more rigorous and controlled in-house evaluation against anticipated attacks (Sections 9.4, 9.5).

9.1 Handling production data

We use real data from production systems to train our defensive models and we report results based on real people using the apps that use our system. As such, for any data we use we employ access control, store it in an encrypted loopback device, and only use end-to-end encrypted file systems when we do open the encrypted loopback device.

9.2 Does Boxer recover false positives in a real deployment?

To evaluate Boxer's ability to recover false positives, we report on results from an app that shipped our SDK. In this deployment the app allowed users flagged by their fraud systems to verify their cards using Boxer instead of blocking them, which is what they did before using Boxer.

From January 22nd, 2020 to February 5th, 2020 we sample 45 users whom the app's systems flagged as fraudulent. 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 do 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 our 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.

9.3 Can Boxer's secure counting catch real attacks?

To evaluate secure counting, we report on data from an app that shipped our SDK in their production system. 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 although our count is passive, they do actively block transactions from suspicious users. Having a passive count is advantageous because we can 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. We also record all the unique userIds that we see for a device but record that in a database and do *not* use secure counting to track that yet. We took a random sample of ten users who hit this maximum count and report the results.

The first question we wanted to answer was whether attackers reset their device. We track device resets by observing an inconsistency between the software count and hardware stratum and record a timestamp for when the reset happens. In our 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 we wanted to answer was whether counting cards added to a device would be useful for stopping fraud. To answer this question, we pulled the userIds from our database for all users who added a card to one of the devices that hit the six-card limit and inspected all their transactions 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 we inspected manually, we had no false positives – fraudsters used all the devices we inspected for attacks. The attacks fell into three different categories, but they were all attacks.

Although we do not know the recall of the Boxer card added count, which would be a measure of how much of the total fraud problem does this signal catch, we can confirm 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 we inspected, one device had three unique users, and another had six unique users all who added cards on the same device, suggesting that tracking unique logins per device could be another useful signal.

Accuracy	Precision	Recall
96.25%	98.25%	94.25%

Figure 13: Screen detection results on a dataset of 800 images having an even split of samples containing and not containing screens.

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

From a production dataset, we randomly select 63 images where attackers scanned cards rendered on screens as our validation set. Boxer's screen detection model caught all 63 attacks. All 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, we perform a more extensive internal evaluation. We manually collected 400 images of different credit cards captured across multiple screens. These 400 samples had credit cards displayed on multiple screens, and we captured them using multiple devices, showing the screen edge in some cases, and not showing in others. We combined this 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 13 summarizes these results.

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

9.5 How viable is the BIN consistency and expectation check?

We design the BIN consistency check to catch attackers who create card images that look like cards, perhaps using a stock card image, but do not match what we expect for a card from that BIN. Since we have not seen this style of attack in the wild, we consider this defense to be a proactive defense that anticipates future attacks. Thus, for evaluation, we test on valid cards to check for false positives and see if it can detect a purposely crafted BIN inconsistent Fugazi fake.

We use a validation dataset containing 2000 legitimate production credit card images. 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.

We created a BIN inconsistent fake card using Fugazi. This card has the BIN of a GreenDot card, but we render it in the

Figure 14: A BIN inconsistent fake card image caught by our 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. Our check detects both inconsistencies, showing that it would flag such a card as fake.

form of a Chase card. While existing apps were unable to detect this fake card (Section 9.7), the BIN check correctly classifies this as a BIN inconsistent image by detecting the presence of a Chase logo as shown in Figure 14.

9.6 What types of attacks are currently being employed by fraudsters, and how does Boxer stop them?

We report the types of attacks from a random sample of attacks observed in the wild. These attacks include:

- 23% of users who did not produce a picture of a card in their scan.
- 74% of users whose scanned cards did not match the card that the app had on record for these users.
- 3% of 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 then a card and users who scanned a card that mismatched what the app had on record for the user. Our 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 our screen detector detects. We did find a single example of a user using a photoshopped card. While Boxer was unable to detect this card, we observe that it had an incorrect font and we expect future systems to detect this style of attack.

Additionally, we describe the attacks stopped by our secure counter in Section 9.3

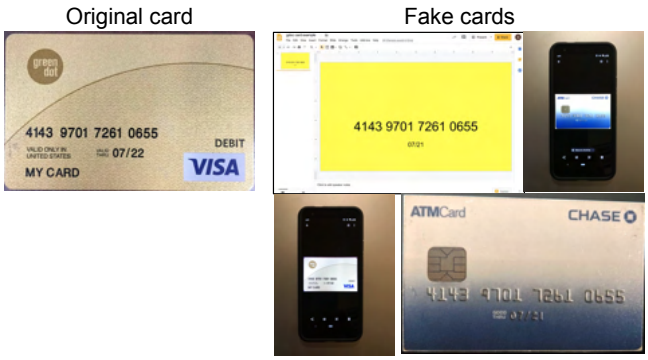

Figure 15: Samples for our case study. This figure shows the original card and four different fake versions of the real card. The fake cards include a Google doc with the number in the middle scanned off a computer screen, a Fugazi fake where the BIN mismatches and scanned off a phone, the real card scanned off a phone, and a Fugazi fake that we printed out using a high quality printer and plastic. We add these fake cards successfully to all the apps that we tested using their card scanner.

9.7 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, we generate several fake cards and add them to a ride sharing app, a food deliver app, an e-commerce app, and a security SDK using their card scanning features. We discuss the ethical considerations of these attacks in Section 9.7.1.

Figure 15 shows both the real card and the fake cards that we use for this experiment. Our 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. We printed the physical card at a local print shop, and it cost $35 per card.

We added the fake cards to all the apps successfully, suggesting that these apps are not looking at the card for signs of tampering. Fraud systems are complicated, and we add cards 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 add the cards.

However, with the food delivery app we created a new account and the cards we added were fake, a classic pattern for financial fraud. After adding the card, we also made a

purchase, showing that this card bypassed all fraud checks. The security SDK that we 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 is false in our experiment.

9.7.1 Ethical considerations

In our experiments, we add fake cards to accounts on real apps. In one experiment, we made a purchase using the fake card. However, the credit card number that we used in this experiment was one from a pre-paid debit card that we had purchased. Thus, we paid the merchant for the food that we received. Also, we consulted with our lawyers and they confirmed that what we did was legal, and we believe that it is ethical.

10 Related work

Payment cards are vulnerable to skimming attacks where data is stolen and sold online [1]. Researchers did a study of card skimmer technology and used it to develop a card skimmer detector [43]. This technology exploits the physical constraints required for a card skimmer to work properly. Scaife et al. [41] 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. [9] evaluate the effectiveness of using Bluetooth scans to detect card 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 [42].

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 [44]. Researchers have shown how BIN can be used in conjunction with the IP address for a BIN/IP check [3] 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, good purchased, shipping address to train a neural network based fraud detection anomaly system [6]. More recently, Zanin proposed a combination of data mining and parenclitic network analysis to ascertain the validity of credit card transactions [55].

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 [16]. Techniques, such as cloning [19] and JPEG quantization [17], use the fact that the underlying statistics of any digitally forged altercation would not match that of a real image, although they look indistinguishable to a human being. Such techniques have also been incorporated into the deep learning era, to train a model to learn the distribution of either real images, and identify fakes through anomaly detection techniques [54], or learn distributions of real and fake images, and accordingly classify an image at test time.

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 [37]. More recently, Gracia and Queiroz also use Moiré pattern analysis to detect replay attacks [20].

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 [7, 50]. Researchers have also proposed a number of systems to enable login systems to verify additional factors implicitly [29, 30, 33]. Finally, researchers have shown how to be smart about when to even ask for additional factors via statistical methods [18].

11 Conclusion

Many apps use scanning to make it easy for users to add payment cards to their apps. Although the current generation of scanners are good at performing OCR, they are *not* ready to stop attacks.

This paper introduced Boxer, a new system for enabling apps to scan payment cards and determine if they are genuine. 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.

Boxer is already beginning to have an impact, with our SDK actively taking production traffic from large and international apps. To date, we have already scanned over 10 million cards, detected real attacks, and shown how our design keeps an eye towards the future by anticipating future attacks and building defenses for them.

Acknowledgments

We would like to thank Pete Chen, David Wagner, Nolen Scaife, Hao Chen, Joy Geng, and Jason Lowe-Power for providing feedback on drafts of our paper. We would also like to thank our shepherd, Patrick Traynor, and the anonymous reviewers who provided valuable feedback on this work. This research was funded in part by a grant from Bouncer Technologies and NSF grant IIS-1748387.

Appendix

A Improving hardware for rate limiting

Based on our experience using Boxer's counting abstraction in a production environment, we describe some of the pragmatic and important lessons learned from our experiences and we suggest modest modifications to the hardware available on iOS and Android devices to better support device-based rate limiting in general and our counting abstraction in particular.

A.1 Impact on legitimate users

Section 7.3 describes how Boxer maintains monotonically increasing counts even as attackers reset their vendorId or move a valid vendorId from one device to another. This section focuses on our legitimate users and how our abstractions strive to handle less common, but still possible, cases well. In particular, the two cases we discuss are (1) users buying a used device and (2) users uninstalling and reinstalling the app.

When a user buys a used device, they inherit the DeviceCheck stratum from the previous owner as DeviceCheck stratum are bound to devices. In the most extreme case, attackers who sell devices with DeviceCheck stratum set to 3 would result in counts already being set to their maximum value. To mitigate this potential risk, we set our reset period (when we reset counts back to zero) to one month, the shortest time period available when using DeviceCheck. DeviceCheck provides timestamps on a month-level of granularity, so each time the timestamp in DeviceCheck mismatches the current month, Boxer resets all counts.

When a user uninstalls and reinstalls the app, or otherwise resets their vendorId, Boxer increases their counts due to software and DeviceCheck strata inconsistencies. However, they will increase their strata only if they increment a count. For example, if a user is at stratum 1 and they uninstall and reinstall the app but never add a card, then their stratum will remain at 1. App uninstall and reinstall cycles will reduce the number of counts available to users, but by dividing our counts into strata we still leave some room for counting.

A.2 Limitations

Based on our experience of running Boxer in a production environment, we discovered two main limitations of our approach.

First, whenever Boxer needs to set counts due to software and DeviceCheck strata inconsistencies, counts from one event may be set even if the user hasn't performed the action associated with the event. Second, we found the DeviceCheck API to be difficult to work with because we need to maintain consistent state across our own counting data and Apple's DeviceCheck state. This classic distributed systems problem is especially difficult for Boxer because we use a read-modify-write pattern to update DeviceCheck bits and we must be able to withstand an attacker who sends a massive number of requests for the same device in parallel, but Apple provides no mechanisms for us to do this consistently. Appendix A.3 discusses how we handle this limitation.

A.3 DeviceCheck for distributed systems

The fundamental problem with DeviceCheck, from a distributed systems perspective, is that app builders need to keep their software strata and Apple's DeviceCheck state consistent. Apple exposes a simple get / set interface, which we assume is atomic and sequentially consistent, but because there are only two bits and no fine-grained timestamps there isn't much we can build on top of it.

In Boxer we serialize all read-modify-write updates to Apple's servers while still allowing read-only requests to access the bits concurrently. To serialize, we use a distributed locking scheme built on top of Google's Cloud Datastore using transactions and a simple lock model. Even with this synchronization scheme and blunt serialization policy, Boxer can handle millions of active devices per month. See Appendix A.5 for more details.

One option for dealing with race conditions is to ignore them since these are used for rate limiting. Allowing 14 failed login attempts instead of 12 is still effective rate limiting.

Based on our experience, we'd like to see a simple extension to the DeviceCheck API to facilitate efficient race-free counting. We propose a short-lived (e.g., 60 seconds) deviceId in addition to the two bits that DeviceCheck gives us that we can use within our own synchronization scheme. Short-lived deviceIds should be simple for Apple to implement on top of any reasonable storage system and has a minimal impact on end-user privacy as legitimate users will always have the same vendorId for concurrent requests on the same device. Only attackers will have different vendorIds for concurrent requests from the same device. By exposing a time-based deviceId, app builders can synchronize access to the DeviceCheck bits for each device, enabling Boxer to handle any practical scale.

A.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 [25] provides a hardware-backed uniqueId that cycles every 30 days. Boxer could use this uniqueId directly in place of Apple's vendorId, which would enable Boxer to count events without needing to synchronize with external servers, like we do when we use DeviceCheck.

However, one limitation of Google's design, from Boxer's perspective, is that end users are unable to reset their uniqueId

before that 30-day period expires. And, this limitation is fundamental as the ability to reset the uniqueId is akin to resetting the hardware.

A second limitation of Google's design is that they hard-code the 30-day cycle parameter, making it impossible for app builders to extend their rate limit period. In contrast, Apple's DeviceCheck design provides a timestamp and leaves it up to the app builder as to when they want to cycle their state, providing more flexibility for app builders to customize their use.

Currently, Google restricts the use of the uniqueId only to system level processes, but as an alternative to DeviceCheck the uniqueId provides some compelling improvements in terms of counting system implementation simplicity.

A.5 Is serializing access to Apple's DeviceCheck servers practical?

To make sure that Boxer's software strata and DeviceCheck state remain consistent, we serialize all read-modify-write updates to DeviceCheck state. To measure the impact of this policy, we measure the latency of read-modify-write updates and the latency of our distributed locks, all of which run in our Goolge App Engine server system. We measured ten updates and report the average across all ten trials.

The total time for handling locking and read-modify-write updates to Apple's DeviceCheck is 916ms. At this latency, Boxer can handle 2.8M updates every 30 days, and assuming an average of 1.5 updates per device per month means that Boxer can handle 1.9M active devices per month. Of the 916ms, 297ms are from our distributed locking scheme, leaving the remaining 619ms for DeviceCheck API calls. Instead of using distributed locks, we could have routed all read-modify-write updates to the same server and used local locks for synchronization, effectively eliminating the 297ms spent on distributed locks. In this case, Boxer can handle 4.2M updates spread across 2.8M active devices over a 30 day period.

These results show that even for a naive implementation and blunt serialization policy, Boxer can handle millions of active devices per month.

B More experiments with Fugazi

To understand if there are any fundamental differences between real samples and those created by Fugazi, instead of creating a Fugazi fake with a new card number, we create a Fugazi version of an original card image (i.e., a fake where the injected digits are the same as that of the original). We then compute the pixel-wise difference image between them. We ensure that the Fugazi fake has the same texture as the original and we align them perfectly. This alignment guarantees that only Fugazi introduces differences between the two. Figure 18 shows the difference image and we make the following observations. First, most parts of the Fugazi generated image are identical to the original, indicating a potential difficulty for machine learning models attempting to detect discriminating features. Second, the difference image, however, highlights exactly those places where Fugazi had to do the most work (Sec 5.1), showing imperfections in Fugazi fakes.

Owing to the observed differences between real cards and their corresponding Fugazi fake versions, we attempted a number of defenses to reliably detect Fugazi.

B.1 Evaluating Fugazi with traditional image forensics techniques

The non-deep learning forensics techniques we attempted, to detect Fugazi rely on differences in the frequency of noise present in natural and tampered images [4], CFA artifacts that are generated by a demosaicing algorithm run on modern digital cameras to reconstruct color images, and thus, not be present in generated images [36] and possible discontinuities in JPEG compression artifacts arising in digitally generated images [23]. However, as Figure 17 shows, none of these techniques detect any discernible differences between real and Fugazi generated samples.

B.2 Internal evaluation of Fugazi with image classification and anomaly detection

We first trained and evaluated a binary classifier to distinguish between real cards and Fugazi fakes. While the trained classifier generalized to detect other fakes that a particuar version of Fugazi generated, it was consistently fooled by those generated by a slightly altered implementation of Fugazi.

Although a binary classifier might work as a signature detection technique where it can identify a specific version of Fugazi, it is unable to detect general fake cards. This limitation is significant in our setting, where fraudsters refine their techniques to adapt with growing defenses.

Next, we cast Fugazi card detection as an anomaly detection problem. We modeled an autoencoder as a deviation-based anomaly detector [14] which we trained to reconstruct only real samples. Our hypothesis was that training only on real samples leads to sub-optimal reconstruction of fakes. Once quantified, we use the reconstruction error to distinguish between real and fake images, since fakes would have a higher reconstruction error.

With our observation that there are imperfections in Fugazi fake cards, the autoencoder based anomaly detection can potentially catch them without being influenced by a particular type of fake. This observation stems from the fact that we only train our autoencoder on real data.

However, when we experimented with several different Fugazi fakes, the reconstruction loss values across training epochs was spread out, with the loss value being higher for

Figure 16: Fake credit cards that we generated using Fugazi.

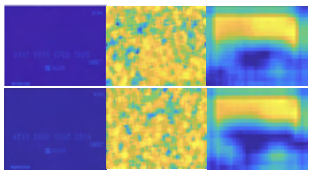

Figure 17: Outputs generated on running noise analysis [4], CFA detection [36] and JPEG inconsistency detection [23], respectively on real and fake images. The top row shows the results on fake images, and the bottom row for real images. The images on the left show no difference in the frequency of noise between real and fake images. The images in the middle show no specific regions that do not contain CFA artifacts (regions shown in blue), and the images on the right show JPEG compressions localizing to the same untampered region (regions shown in yellow) in real and fake images.

Figure 18: Difference image between a real image and a Fugazi generated image with the same digits and same texture, that we positioned to align with each other. The black regions in the differnce image show identical portions between the two images, while the white regions highlight differences.

real cards in some cases, and higher for fakes in others. This data suggests that autoencoder was unable to detect fake cards.

References

[1] 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.

[2] Binlist: An open-source list of bank bin/iin numbers. https://github.com/iannuttall/binlist-data.

[3] Combo ip/bin checker. https://www.bincodes.com/ip-bin-checker/.

[4] Noise analysis for image forensics. https://29a.ch/2015/08/21/noise-analysis-for-image-forensics.

[5] Vendorid documentation, 2019. https://developer.apple.com/documentation/uikit/uidevice/1620059-identifierforvendor.

[6] John Akhilomen. Data mining application for cyber credit-card fraud detection system. In Petra Perner, editor, *Advances in Data Mining. Applications and Theoretical Aspects*, pages 218–228, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[7] Vijay A. Balasubramaniyan, Aamir Poonawalla, Mustaque Ahamad, Michael T. Hunter, and Patrick 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, pages 109–120, New York, NY, USA, 2010. ACM.

[8] V. Benjamin, W. Li, T. Holt, and H. Chen. Exploring threats and vulnerabilities in hacker web: Forums, irc and carding shops. In *2015 IEEE International Conference on Intelligence and Security Informatics (ISI)*, pages 85–90, 2015.

[9] Nishant Bhaskar, Maxwell Bland, Kirill Levchenko, and Aaron Schulman. Please pay inside: Evaluating bluetooth-based detection of gas pump skimmers. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 373–388, Santa Clara, CA, August 2019. USENIX Association.

[10] Airbnb Data Science Blog. Fighting financial fraud with targeted friction, February 2018. https://medium.

com/airbnb-engineering/fighting-financial-fraud-with-targeted-friction-82d950d8900e.

[11] Uber Engineering Blog. Advanced technologies for detecting and preventing fraud at uber, June 2018. https://eng.uber.com/advanced-technologies-detecting-preventing-fraud-uber/.

[12] Caroline 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.

[13] Cayan. Preventing card-not-present fraud. https://cayan.com/Site/Media/Cayan/Insights-Content/preventing-card-not-present-fraud_cayan.pdf.

[14] Carl Doersch. Tutorial on variational autoencoders, July 2016. https://arxiv.org/abs/1606.05908.

[15] Serge Egelman, Feburary 2019. https://blog.appcensus.io/2019/02/14/ad-ids-behaving-badly/.

[16] H. Farid. Image forgery detection. *IEEE Signal Processing Magazine*, 26(2):16–25, March 2009.

[17] Hany Farid. Digital image ballistics from jpeg quantization, 2006.

[18] David Freeman, Sakshi Jain, Markus Durmuth, Battista Biggio, and Giorgio Giacinto. Who are you? A statistical approach to measuring user authenticity. In *NDSS*. The Internet Society, 2016.

[19] Jessica Fridrich, David Soukal, and Jan Luk. Detection of copy-move forgery in digital images, 2003.

[20] Diogo Garcia and Ricardo De Queiroz. Face-spoofing 2d-detection based on moiré-pattern analysis. *IEEE Transactions on Information Forensics and Security*, 10:778–786, 04 2015.

[21] A. Haslebacher, J. Onaolapo, and G. Stringhini. All your cards are belong to us: Understanding online carding forums. In *2017 APWG Symposium on Electronic Crime Research (eCrime)*, pages 41–51, 2017.

[22] Minyoung Huh, Andrew Liu, Andrew Owens, and Alexei A. Efros. Fighting fake news: Image splice detection via learned self-consistency. *arXiv preprint arXiv:1805.04096*, 2018.

[23] Chryssanthi Iakovidou, Markos Zampoglou, Symeon Papadopoulos, and Yiannis Kompatsiaris. Content-aware detection of jpeg grid inconsistencies for intuitive image forensics. *Journal of Visual Communication and Image Representation*, 54:155–17, 2018.

[24] Apple Inc. Devicecheck documentation, 2019. https://developer.apple.com/documentation/devicecheck.

[25] Google Inc. Key and id attestation, 2019. https://source.android.com/security/keystore/attestation.

[26] Mike 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.

[27] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A Efros. Image-to-image translation with conditional adversarial networks. *CVPR*, 2017.

[28] Juniper Research. Online Payment Fraud Whitepaper. http://www.experian.com/assets/decision-analytics/white-papers/juniper-research-online-payment-fraud-wp-2016.pdf.

[29] Nikolaos Karapanos, Claudio Marforio, Claudio Soriente, and Srdjan Capkun. Sound-proof: Usable two-factor authentication based on ambient sound. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 483–498, Washington, D.C., 2015. USENIX Association.

[30] Wei-Han Lee, Xiaochen Liu, Yilin Shen, Hongxia Jin, and Ruby 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, pages 67–78, New York, NY, USA, 2017. ACM.

[31] Ming-Yu Liu, Thomas Breuel, and Jan Kautz. Unsupervised image-to-image translation networks. In *Advances in neural information processing systems*, pages 700–708, 2017.

[32] Lyft Engineering Blog. Stopping fraudsters by changing products, December 2017. https://eng.lyft.com/stopping-fraudsters-by-changing-products-452240f2d2cc.

[33] Shrirang Mare, Andres Molina-Markham, Cory Cornelius, Ronald A. Peterson, and David Kotz. ZEBRA: zero-effort bilateral recurring authentication. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 705–720, 2014.

[34] Mashable. Say goodbye to those fake likes: Huge click farm discovered in thailand, 2017. https://mashable.com/2017/06/13/thailand-click-farm-caught/#ZGoNx7UDDOqj.

[35] NIST. Nist nimble challenge 2018. https://www.nist.gov/itl/iad/mig/nimble-challenge-evaluation-2018.

[36] A. De Rosa P. Ferrara, T. Bianchi and A. Piva. Image forgery localization via fine-grained analysis of cfa artifacts. *IEEE Transactions on Information Forensics and Security*, (5), oct 2012.

[37] 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)*, pages 98–105, May 2015.

[38] Deepak Pathak, Philipp Krahenbuhl, Jeff Donahue, Trevor Darrell, and Alexei A Efros. Context encoders: Feature learning by inpainting. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2536–2544, 2016.

[39] R. Byron Pipes. Moiré analysis of the interlaminar shear edge effect in laminated composites. *Journal of Composite Materials*, 1971.

[40] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 91–99. Curran Associates, Inc., 2015.

[41] 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)*, pages 1000–1014, 2019.

[42] 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)*, pages 1063–1076, 2018.

[43] Nolen Scaife, Christian Peeters, and Patrick Traynor. Fear the reaper: Characterization and fast detection of card skimmers. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1–14, Baltimore, MD, 2018. USENIX Association.

[44] Jeff Stapleton and Ralph Spencer Poore. Tokenization and other methods of security for cardholder data. *Information Security Journal: A Global Perspective*, 20(2):91–99, 2011.

[45] Tao Stein, Erdong Chen, and Karan Mangla. Facebook immune system. In *Proceedings of the 4th Workshop on Social Network Systems*, SNS '11, pages 8:1–8:8, New York, NY, USA, 2011. ACM.

[46] Brett Stone-Gross, Marco Cova, Lorenzo Cavallaro, Bob Gilbert, Martin Szydlowski, Richard Kemmerer, Christopher Kruegel, and Giovanni Vigna. Your botnet is my botnet: Analysis of a botnet takeover. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, page 635–647, New York, NY, USA, 2009. Association for Computing Machinery.

[47] Stripe. Disputes and fraud. https://stripe.com/docs/disputes.

[48] Supasorn Suwajanakorn, Steven M. Seitz, and Ira Kemelmacher-Shlizerman. Synthesizing obama: Learning lip sync from audio. *ACM Trans. Graph.*, 36(4):95:1–95:13, July 2017.

[49] Twitter Engineering Blog. Fighting spam with botmaker, August 2014. https://blog.twitter.com/engineering/en_us/a/2014/fighting-spam-with-botmaker.html.

[50] Erkam Uzun, Simon Pak Ho Chung, Irfan Essa, and Wenke Lee. rtcaptcha: A real-time CAPTCHA based liveness detection system. In *NDSS*. The Internet Society, 2018.

[51] Visa. Counterfeit fraud at U.S. chip-enabled merchants down 70%. https://usa.visa.com/visa-everywhere/security/visa-chip-card-stats.html.

[52] Visa. Visa core rules. https://usa.visa.com/dam/VCOM/download/about-visa/15-April-2015-Visa-Rules-Public.pdf.

[53] Bihan Wen, Ye Zhu, Ramanathan Subramanian, Tian-Tsong Ng, Xuanjing Shen, and Stefan Winkler. Coverage—a novel database for copy-move forgery detection. In *2016 IEEE International Conference on Image Processing (ICIP)*, pages 161–165. IEEE, 2016.

[54] Sri Kalyan Yarlagadda, David Güera, Paolo Bestagini, Fengqing Maggie Zhu, Stefano Tubaro, and Edward J. Delp. Satellite image forgery detection and localization using GAN and one-class classifier. *CoRR*, abs/1802.04881, 2018.

[55] Massimiliano Zanin, Miguel Romance, Santiago Moral, and Regino Criado. Credit card fraud detection through parenclitic network analysis. *CoRR*, abs/1706.01953, 2017.

[56] Peng Zhou, Xintong Han, Vlad I Morariu, and Larry S Davis. Learning rich features for image manipulation detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1053–1061, 2018.