CamForensics: Understanding Visual Privacy Leaks in the Wild

Animesh Srivastava
Duke University
animeshs@cs.duke.edu

Puneet Jain
Hewlett-Packard Labs
puneet.jain@hpe.com

Soteris Demetriou
UIUC
sdemetr2@illinois.edu

Landon P. Cox
Duke University
lpcox@cs.duke.edu

Kyu-Han Kim
Hewlett-Packard Labs
kyu-han.kim@hpe.com

KEYWORDS

Visual Privacy, Android, Camera.

ABSTRACT

Many mobile apps, including augmented-reality games, bar-code readers, and document scanners, digitize information from the physical world by applying computer-vision algorithms to live camera data. However, because camera permissions for existing mobile operating systems are coarse (i.e., an app may access a camera's entire view or none of it), users are vulnerable to visual privacy leaks. An app violates visual privacy if it extracts information from camera data in unexpected ways. For example, a user might be surprised to find that an augmented-reality makeup app extracts text from the camera's view in addition to detecting faces. This paper presents results from the first large-scale study of visual privacy leaks in the wild. We build CamForensics to identify the kind of information that apps extract from camera data. Our extensive user surveys determine what kind of information users expected an app to extract. Finally, our results show that camera apps frequently defy users' expectations based on their descriptions.

1 INTRODUCTION

Cameras are as essential to modern mobile devices as touchscreens and wireless networking. A device's camera allows users to capture and share important moments, and programmatic camera access provides apps with a rich interface for digitizing information about the physical world.

At the same time, cameras create new privacy challenges for mobile operating systems. Apps can often access both essential (e.g., a QR code) and inessential (e.g., text) data within the same camera view. This co-mingling of essential and inessential data could lead to unexpected app behavior from the perspective of a user as she was not aware of it because of the lack of app's disclosure. For example, a benign app whose ostensible purpose is scanning QR codes (expected) could also extract text from the camera feed using optical-character recognition (OCR) (unexpected) for app's developers to understand the app's usage. Similarly, a more curious camera app could perform a face recognition (unexpected) whenever the user takes a selfie (expected). Such unexpected behavior could turn into "leaks" when the app's intent is malicious. Existing mobile platforms provide only coarse-grained access controls for the camera (i.e., an app can access all of a camera's view or none of it), though numerous recent proposals have attempted to protect visual privacy through finer-grained control [1, 8, 13, 14, 24]. And yet despite this large body of work, we are unaware of any large-scale empirical studies that characterize visual privacy in today's mobile apps. Without such a study, critical questions, such as what information mobile apps extract from camera data, and users' privacy expectations of camera apps, will remain unanswered.

Figure 1: An augmented-reality app with access to essential and inessential data in the same camera view.

To shed light on these and related questions, we collect over 230K apps with access to camera data from the Google PlayStore and develop an app analysis tool called *CamForensics*. CamForensics is a custom Android environment and a suite of test inputs that feeds an app simulated camera data containing specific types of information. CamForensics monitors an app's execution as it processes simulated inputs and searches for evidence that the app performs certain image analysis on the camera feed, such as faces and text. From the original corpus of 230K apps, we use CamForensics to study over 600 of the most popular apps that use well-known libraries for augmented reality (AR), bar-code reading, face detection, and OCR. The purpose of this tool is to collect evidence and inform the users of any visual inference drawn from the camera feed, and users can

SenSys '17, Delft, Netherlands

DOI: 10.1145/3131672.3131683

deem it as expected or unexpected. Some of these unexpected image analysis could lead to users breach of visual privacy based on their perspective of the app. Thus, in addition to using CamForensics to study apps' behavior, we distribute a large survey to characterize users' expectations. Our survey presents each participant with information about a subset of 325 apps that CamForensics identified as performing bar-code reading, face detection, OCR and AR, and asks what classes of information the participant believes these apps gather from the camera.

Through CamForensics and our survey, this paper is the first to provide answers to the following questions: (1) how prevalent is computer vision among smartphone apps with camera access, (2) what information do apps extract from camera data, and (3) is the information apps extract consistent with users' expectations. The primary results of our study are as follows:

- **Most apps with camera-access use third-party libraries for image processing.** Computer vision and other image-processing algorithms are difficult to implement, and the vast majority of camera apps use well-known third-party libraries rather than implementing their own. This was critical for CamForensics, because we could model interactions with third-party libraries offline to better infer an app's intentions under testing.
- **Apps routinely defy users' expectations.** In our surveys on selected apps, for 61% of them, users are unable to identify the type of image processing given the app description. Moreover, 19% of apps extract information from camera data that users do not expect. We also find an app whose behavior is suspicious. The majority of our surveys indicate that app developers do a poor job of signaling to users exactly how an app will use camera data.
- **Offloading image processing is not uncommon and presents a gray area for visual privacy.** 10 Augmented Reality apps in our study send camera data over the network for getting the matching results from a remote server. While we see the data sent in the captured network packets, we are unable to confirm the exact nature of content – an app may send original images or extract features without informing the user about it.

2 DATASET

Our work aims to understand how mobile apps use visual information. In particular, we are interested in detecting events when an app's execution does not conform to a user's expectations [6, 23]. Some of these events can be visual privacy violations. To study risks and implications of visual privacy violations in the wild, we download and analyze 230K camera-based Android apps. Our data collection focuses on Android apps [26], however, it is generalizable to other platforms.

2.1 Data Collection Methodology

Android apps can be collected by crawling Google PlayStore using a web crawler. However, it is a challenging task because PlayStore thwarts such efforts. For example, it blocks the IPs connecting for polling their service aggressively. The other option is to download apps by visiting their homepage, possibly in an automated manner

App description details	
Name	Name of the app
AppId	Package name
Category	PlayStore category
IsFree	If the app is free
Reviewers	Total number of reviews
ScoreTotal	Total number of ratings
ScoreCount	Average rating of the app
Installations	Number of downloads
Permissions	Permissions required by the app
Description	Description of the app provided by app developer

Table 1: The relevant fields present in the app database.

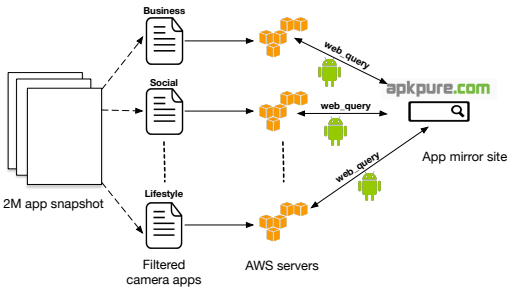

Figure 2: Data collection methodology.

by reverse engineering user interface. Unfortunately, this process is tediously slow and would take months before a reasonable size dataset is gathered. Furthermore, frequent changes in App store APIs and homepage rendering make our attempts futile.

Previous work [29], and individual aficionados [12] have built a PlayStore crawler. The latter periodically snapshots metadata of the PlayStore. Table 1 lists some of the important fields present in a snapshot. We leverage snapshot dated July 18, 2016 in this study. It is important to note that snapshot database *contains only* metadata and *does not contain* application binaries (apk files).

The snapshot contains metadata of 2.07M apps in PlayStore – which roughly equates to the total number of apps at that time. However, for an indepth study, metadata alone is not sufficient – a copy of apps' executables (apks) is required. Since PlayStore does not allow aggressive downloading, we poll a PlayStore mirror site (`apkpure.com`) to achieve our goal.

Figure 2 illustrates our data collection methodology. We filter the snapshot of 2.07M apps and select apps that need the camera permission (`android.permission.CAMERA`). These apps are identified by the permission description "`TAKE PICTURES AND VIDEOS`". Once identified, the apps are referred by package names (e.g., WhatsApp by `com.whatsapp`). For each package, we construct a url in the following form: `https://apkpure.com/whatsapp/<<package_name>>/download?from=details`. The url is posted on `apkpure.com` and the response is a webpage that contains a direct download link. To expedite the entire process, we parallelize over 16 AWS (Amazon Web Services) instances. We spent 10 days in downloading 230K camera-based apps between July 19 – 29, 2016.

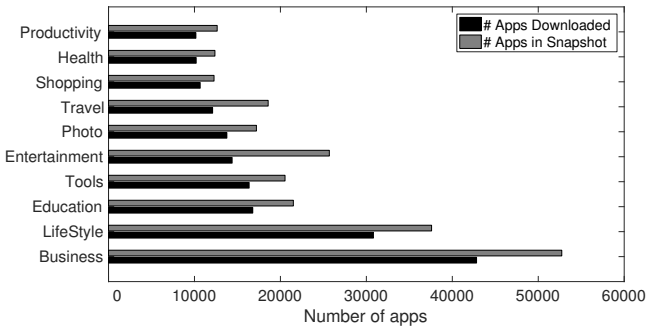

Figure 3: Distribution of apps downloaded in the top-10 popular camera-based categories.

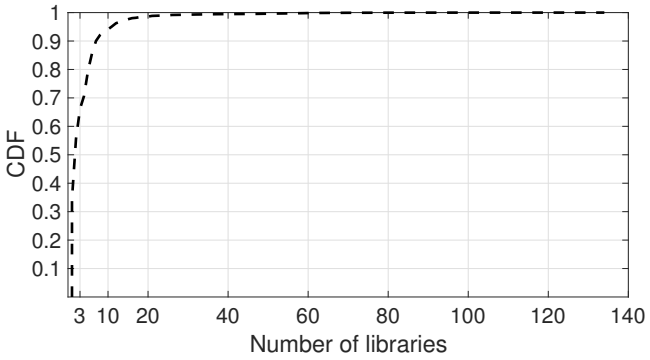

Figure 4: CDF of the number of third-party libraries per app.

We were able to download 230K of 327K apps that had camera permission in the snapshot. The downloaded apps belong to 20 different categories. Figure 3 depicts the count of camera-based apps in the 10 most popular categories (the ones with most number of apps). By nature, the snapshot database has non-uniform number of apps per category. Moreover, at download time, several apps were not available at the mirror site which led to non-uniform number of apps downloaded per category. It is to be noted that our study *does not include paid apps* since the mirror site only hosts free apps.

2.2 Use of Third-Party Libraries

We observe from the dataset that the camera apps which perform some kind of image processing rely heavily on native libraries. While some of these native libraries are private, others are freely available for reuse. These libraries alleviate developers from the burden of writing complex algorithms and save valuable development time. Since libraries are written to be used in a black-box manner, app developers typically include them on the basis of utility, without comprehensive understanding. Therefore, a library mishandling visual information can quickly affect a large population (all apps using the library and all users that install those apps).

In studying violations of user expectations in visual information handling, one could study the practices of each camera-based app individually. This would be a cumbersome and tedious process. Based on the observation of the broad use of third-party libraries, we subselect the ones that help in performing image processing. We then use them as common execution environments across a number of apps – drawing insights regarding visual privacy violations.

We inspect all apps in the dataset to understand the general use of third-party libraries. We first extract libraries from the apk file using the `apktool` [28]. Every app that uses a third-party native library contains a special designated folder (e.g., `<app_package>/libs/armeabi`). For each app, we inspect this folder and prepare a list of the libraries found. The average number of native libraries per app in our dataset is 3.8. Figure 4 shows an empirical CDF of the number of native libraries per app. As evident from the plot that distribution is long-tailed, implying that only a handful of apps contain a very large number of third-party libraries. The maximum number of 134 libraries is found in the `PicsArt Photo Studio` app. `PicsArt` is a picture editor app and provides several image manipulation features. Overall, in our dataset, a significant number of apps, 33.3% (76,598/230,000), have at least one third-party library. This verifies that the use of third-party libraries is indeed widespread.

Library labels and their description	
Label	Description
text	character encoding
barcode	QR or barcode reader
pdf	pdf rendering
game	gaming engines
sdk	suport for app development
vision	computer vision support
credit	credit card reader
audio	audio encoding/decoding
database	database support
geolocation	location services
image	image processing support
ocr	text recognition

Table 2: Labels assigned to the different libraries.

We find a total of 15267 unique libraries in the dataset. However, the majority of these libraries are not used for image processing. Therefore, we manually examine and label image processing libraries. Since the number of unique libraries is large, we focus on the top-100 frequently used. Our labeling criteria are based on the following sources: (1) the websites that maintain these libraries such as Github; (2) online discussion forums such as Stack Overflow; and (3) technical experts at our institution. Table 2 shows the set of labels assigned to the libraries.

The process of labelling helps us in identifying the libraries that have access to camera data (online or offline). Essentially, libraries that are assigned labels `barcode`, `vision`, `credit`, `image`, and `ocr` perform some kind of image processing.

Table 3 shows the top-10 frequently used libraries among the camera-based apps. Note that libraries capable to handle visual information such as `libzbarjni.so` for barcode scanning, `libopencv_core.so` for computer vision, and `libcardioDecider.so` for credit card recognizers are among the

Library	Description
libiconv.so	Text encoder/decoder
libzbarjni.so	Barcode detector
libvudroid.so	PDF renderer
libstlport_shared.so	C++ standard library
libunity.so	Gaming engine
libmono.so	.NET app support
libopencv_core.so	Computer vision
libopencv_imgproc.so	Computer vision
libcardioDecider.so	Credit card recognizer
libcardioRecognizer.so	Credit card recognizer

Table 3: Most frequently used libraries.

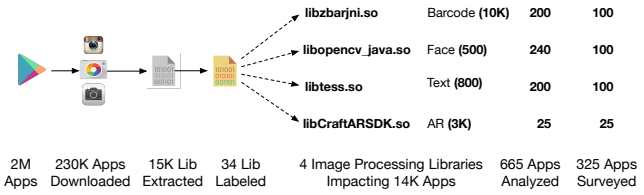

Figure 5: Scope of measurement study.

top. This shows that camera-based apps do turn to third party libraries for processing of visual data.

Given that the use of third-party libraries is common, it is natural to ask what fraction of them are open-source vs closed. During library labeling, we gathered information about the availability of source code of third-party libraries. Among top-100 frequently used libraries, ≈ 55% are open sourced. However, among libraries that perform image processing, the number of open-sourced is much higher, 75%. While code instrumentation of open-source libraries is trivial, analysis of apps which use closed source libraries is challenging. Later sections discuss this issue in detail.

2.3 Scope of Measurement Study

Our study starts with a metadata snapshot of PlayStore. The data contains information of ≈ 2M apps. We identify 327K apps that use the camera and download 230K of them. For the downloaded apps, we extract their native libraries (total of 15K) and identify 34 most popular image processing libraries. For better understanding, we instrument four of them. These four libraries impact 10K barcode scanning, 800 text recognition, 500 face detection, and 25 augmented reality apps. However, we analyze only the most popular apps in these categories (665). Further, we perform surveys on a handful of the analyzed apps to understand users' expectation. Figure 5 shows the global picture of the scope of this study, in terms of the number of apps and third-party library studied respectively.

2.4 Systems Aspect of Image Processing

The use of a vision-based third-party library merely indicates that app might perform image processing. It neither confirms its processing location (local or cloud) nor processing type (face detect, text, or none). We answer these questions below:

What fraction of apps perform image processing on device?

The fraction of apps performing image processing locally can be derived by identifying libraries which allow that. Therefore, we looked at the libraries which were labeled as image processing library (e.g., `barcode`) in the previous section. Among 100 most frequently used libraries, we found that 34 libraries allow image manipulation on the device. Further, we found that these 34 libraries are present in 12.43% of camera-based apps. While alarming, this is our conservative estimate since other apps using not-so-popular image processing libraries are excluded.

What kind of image processing is performed?

To understand the kind of image processing performed, we study 34 libraries and the apps containing them. We found that these libraries are most commonly used for the following tasks: (1) *barcode reading*, (2) *computer vision* operations, (3) *credit card detection and recognition*, (4) *augmented reality*, and (5) *text recognition*. Then, we compute the number of apps performing such type of processing.

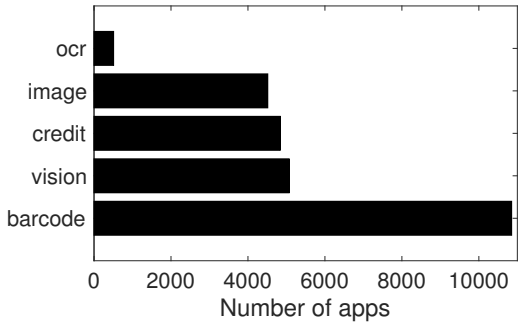

Figure 6: Number of apps using popular image processing libraries.

Figure 6 shows that the number of apps using these libraries are in the order of thousands. Specifically, we found that over 10,000 apps use a barcode reader library. These numbers are a testimony that *on-device image processing is fairly common*. Figure 6 also shows that app developers typically do not implement image processing algorithms, instead rely on readily available libraries. While this practice encourages code reuse, it is worrisome in the sense that a visual privacy leak in one could affect thousands of apps and potentially millions of users.

3 USER STUDY

While it is evident that a visual privacy leak could pose serious privacy risks, no consensus exists on its definition or solution. Visual privacy is of different importance to different people – widely varying based on the utility of app in hand. Therefore, building a solution which is agreeable to the majority is a hard problem. As a first step, we conduct a user study to infer what would the majority agree as visual privacy breach and use it to develop a more generalized definition of visual privacy. Our goal is to quantify *awareness* and *perception* of general population towards visual privacy and subsequently use findings as the basis of system design. We begin with some simple surveys specifically designed to measure user awareness on real-world camera apps. Next, we conduct a study

to determine user's perception of the app under certain scenarios. Specifically, we sought answers to the following questions in our user study:

1) Given real-world app descriptions, how informative are they in making users *aware* of the kind of image processing happening inside the app (See Section 3.1)?
2) Given real-world scenarios and apps performing certain image processing task, how do users *perceive* the situation (See Section 3.2)?

Study setup: We chose Amazon Mechanical Turk (AMT) for conducting user studies[1]. AMT provides an easy-to-use platform to support large scale user studies. The volunteers on AMT are referred to as workers and questions are presented as Human Intelligence Tasks (HITS). Each HIT typically has an associated monetary reward and upon completion, the reward is automatically deposited to a worker's account. A HIT requester (i.e., the survey creator) may select workers based on their qualifications, such as the worker's location or past review scores. Using the AMT platform we were able to distribute surveys which consist in total 325 HITS.

Recruiting participants: We restricted our user study to English speakers, and all questions and related apps' descriptions were in English. We further limited our HITs to workers from English-speaking countries (e.g., the US, Canada, and the UK). We configured the AMT surveys with our HIT worker's eligibility requirements and AMT recruited them. The reward associated with each HIT varied between 1 and 10 cents based on its complexity.

Collecting responses: We put up the user study on AMT and set its expiration period to be 10 days. However, in all the surveys that we conducted, we were able to collect the responses within 7 days. For each question, we collected 16 responses and we set the number of votes to be at least 10 in order to make a consensus. Once the HIT workers have responded, AMT provides a feature for the HIT requester to review the responses and approve them. For stricter vetting on HIT workers, we relied on AMT's rating system where workers were selected with higher ratings.

3.1 User Awareness of Image Processing

The only way for users to guess how an app might use camera data are from the app's description and any runtime cues the app might provide. But even these hints do not provide any guarantees, and an app that ostensibly looks for faces could also extract text. This presents users with a dilemma: trust an app to handle camera data appropriately by giving it full access, or deny camera access and render the app unusable. In this section, we seek to understand how well app descriptions inform users about what information an app extracts from the camera.

We selected a set of 100 apps comprised of 25 each from the following four categories: face detection, OCR, barcode scanning, and apps that perform no image processing. The apps were selected only after confirming logs that they indeed performed such tasks. These apps were selected in the order of decreasing popularity on PlayStore and they contain only English descriptions. A participating worker was presented with a HIT consisting of the app name, the description, and a multiple-choice, multiple-select question. Based on the provided information, we asked whether the worker could infer what kind of image processing the apps perform. AMT workers indicated their response by selecting one or more of the following choices: (1) face detection, (2) text extraction, (3) barcode scanning, and (4) none. For each HIT, we collected 16 responses.

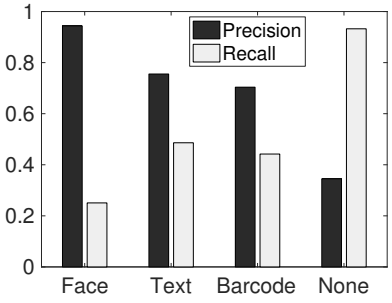

Figure 7: Precision and recall for app categories indicated by AMT workers based on the provided description.

We measured how effectively AMT workers could infer apps' type using precision and recall, as shown in Figure 7. We define precision as *among all apps that a volunteer believed performed a certain type of image processing, what fraction actually did.* The recall is defined as *among all apps that truly performed a certain type of image processing, what fraction did the volunteer identify.* Across all app categories, workers achieved relatively high precision (> 70%), meaning that when a worker believed that an app performed a specific form of image processing it often did.

However, workers also achieved a relatively low recall (< 49%), meaning that workers often failed to identify the kind of image processing that apps performed. OCR was the easiest category to recognize, achieving 49% recall. That is, 49% of the time, workers correctly identified apps that extract text from camera data. This is likely because developers of these apps often use "OCR" in their descriptions. Workers had much more trouble identifying apps in other categories. These results indicate that users often have difficulty identifying what information an app will extract from camera data using app descriptions alone. However, the OCR results offer hope that explicit declarations can help users in correctly understand an app's behavior.

Figure 8 plots the confusion matrix of AMT workers' responses. A high value along the diagonal indicates that the AMT workers correctly inferred how each app processes images. This provides some insight into the sources of workers' confusion. For face detection and barcode scanning, most descriptions do not indicate image processing, and as result users frequently categorized them as 'None.' The results also show that workers often misclassified OCR and barcode apps as each other.

After our initial survey, we studied whether users could identify apps more effectively when given stronger hints about the apps. We provided them with the correct app category, and asked how strongly they believed that the app belonged in that category.

For this study, we prepared a survey with 100 apps containing an OCR library. Each AMT worker was presented with a HIT consisting of the app name, description, question, and five rating options.

[1]The procedures for the studies were vetted and approved in advance by our institution's ethics and legal review board.

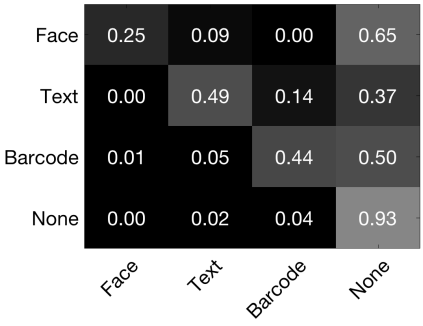

Figure 8: Confusion matrix representation of AMT workers response. The numbers inside each cell indicate the fraction of votes for the corresponding category of image processing.

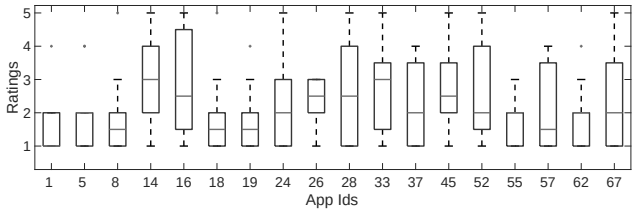

Figure 9: Apps for which the usual consensus is that they are not performing any text detection or recognition. The x-axis represents the app ids and the y-axis represents the ratings. Ratings are on a scale of 1 to 5, where 1 represents strong disagreement and 5 strong agreement. Note that all apps had been observed performing *text detection.*

In each HIT, we ask how strongly the workers felt that the app performs OCR on a Likert scale of 1 to 5 (strongly disagree to strongly agree). For each HIT, we collected 16 responses. Figure 9 shows that for 18 of the 100 apps, AMT workers frequently disagreed that the app performed text processing (10 or more votes with ratings ≤ 2). For example, `ColorSnap Visualizer` is one such app which performs text recognition and all the 16 votes indicated that the HIT workers did not consider it to perform any such activity. `ColorSnap Visualizer` is an app which lets a user pick a color from a picture and then finds the closest color manufactured by *Sherwin-Williams Paints*. The app description has not mentioned anything (as of today) related to the text recognition, however, upon installation we find that it provides a feature to read a color number from the camera feed.

We repeated this survey for 100 apps performing face detection. The apps were selected after logging their use of Android's native OpenCV face detector. As with the OCR apps, based on the app name and description, we asked workers if they thought that the app performed face detection. Figure 10 shows the results. This time, AMT workers mistakenly believed that 26 out of 100 apps were not performing face detection.

Based on all of our surveys, it is clear that *app descriptions rarely allow users to understand what information the apps extract from camera data*. Although we did not observe any malicious behavior, an unscrupulous developer could easily prey on users' confusion without detection.

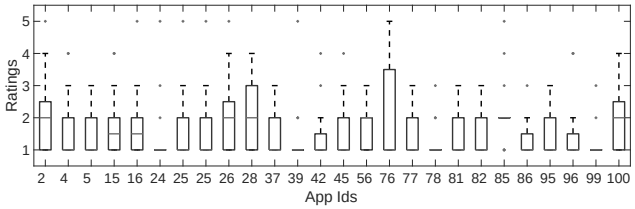

Figure 10: Apps that users mistakenly believed did not perform face detection. The x-axis represents app ids and the y-axis represents the ratings. Ratings are on a scale of 1 to 5, where 1 represents strong disagreement and 5 strong agreement. Note that all these apps perform *face detection.*

Method#	Description
M1	No image analysis
M2	Local image analysis
M3	Local image analysis and results shared with app developers for improvements
M4	Image sent to cloud for analysis and results shared with app developers for improvements

Table 4: Camera data handling methods introduced to the workers in the questionnaire

3.2 User Perception of Visual Privacy

The previous section demonstrates that there is discrepancy between the app descriptions and the apps' handling of camera data. This discrepancy makes users less aware of potential visual privacy risks. An app, after performing an image analysis, can either keep the analysis on the device or it can send it over the network. Once the analysis leaves the device, the user has no control over it. We designed some real-world scenarios which involved apps performing image analysis and sharing the results. We wanted to capture user's perception when they are more aware of the app's handling of image analysis. In each scenario, we introduce study participants to methods in which an app might handle camera data and ask to what extent they are acceptable to them. Finally, we provide participants with several solutions in which apps could make them more aware and ask how strongly they find solutions valuable.

Our user study is located at `https://goo.gl/Y97OFI`). We envision two different cases: (1) Alice is interacting with a camera app to take a picture of a receipt, and (2) Alice is taking a picture with her friends at a bar. Both cases present scenarios where the camera app perform image analysis (text recognition in case (1) or face detection in case (2)) with some or no reliance on the remote server. In each scenarios, the app provides some utility to Alice.

We asked 50 AMT workers to rate their concern on a Likert scale of 1 to 5 (no concern to serious concern) based on the camera data handling method used by the app. Methods are listed in Table 4.

Figure 11 shows that users are increasingly concerned when app tends to move data out of the device. In case of M2 – local image

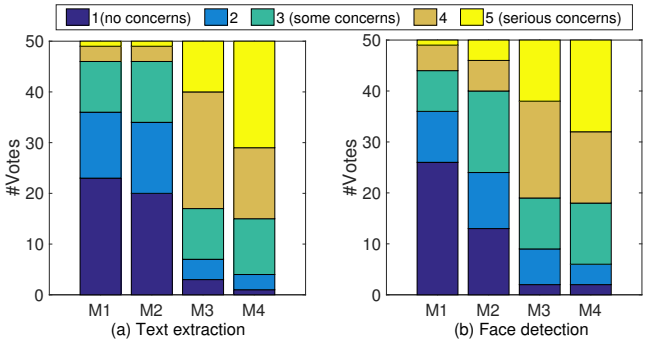

Figure 11: On a scale from 1 to 5, participants' level of concern from different camera data handling methods.

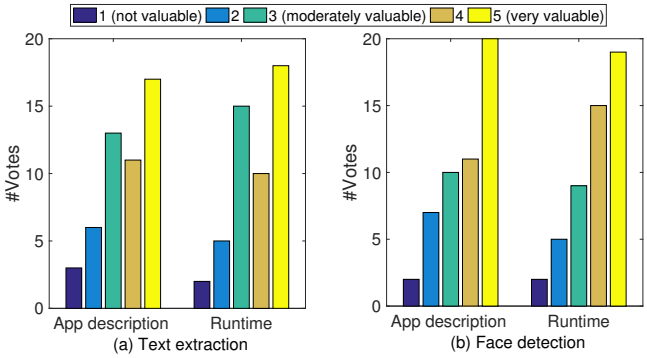

Figure 12: On a scale from 1 to 5, participants' ratings on the value of the disclosure of camera data handling.

analysis, we noticed different levels of concern for text extraction and face detection. With local text extraction majority had little to no issues, however with local face detection most raised some form of concern (25% more). In both the cases, users showed high level of concern when analysis result was shared with the app developer. In case of M4, where the image was sent to a remote server, a vast majority of 44% participants showed "serious concerns".

We further asked the participants whether they find it valuable to have apps disclose their camera data handling. Specifically, we asked them to rate the utility of the following on a Likert scale 1 to 5 (not valuable to very valuable): (a) Apps' app store description of how images are processed and where are they stored, and (b) Run-time app notification describing how images are processed and where they are stored. In both cases, the majority of participants (> 82%) voted that it will be helpful (ratings ≥ 3) if apps disclose the way they handle the camera data. Figure 12 also shows that participants prefer run-time notifications over app store description.

In summary, our AMT based user study provides two valuable takeaways. First, given current app store descriptions of the apps, it is hard for users to judge the kind of information app extracts from camera data. And second, given a choice, users would like to know what kind of information apps extract from camera data, including the location of image processing.

4 THREAT MODEL

We say that a mobile app violates a user's visual privacy when the app extracts more information from a device's camera data than a user expects without being upfront about it. Thus, to detect apps that violate visual privacy, we need to both identify what information an app extracts from camera data and understand what behavior users expect of the app.

To identify what information apps extract from camera data, we use CamForensics. CamForensics is integrated with Android, and we trust the entire Android platform, including the operating system kernel with all support libraries. We assume that an untrusted app can only access camera data through platform-provided APIs, and we assume that a studied app does not collude with other apps installed on the device.

As we described in Section 2, many camera apps rely on third-party libraries for computer vision. In order for CamForensics to understand what information an app extracts from camera data, we must understand how it uses these libraries. Thus, CamForensics only draws conclusions about an app's behavior if the app uses a third-party library whose functionality is already known. CamForensics uses the knowledge to interpret what runtime invocations of the method calls to the third-party library tell us about the app's intentions. An app that is determined to avoid analysis could change library method names to prevent CamForensics from understanding its behavior. However, for the purposes of our study, we assume that apps do not obfuscate their behavior in this way.

Some apps use remote processing to extract information from camera data rather than invoking libraries on the device. Because processing occurs off device, CamForensics cannot draw any conclusions about these apps' behavior. However, remote processing represents a gray area for visual privacy. Even if a user expects an app to extract specific kinds of information from camera data, they may not expect or be comfortable with camera data leaving their device. We will return to the privacy implications of remote processing in Section 7.3.

Since an app indicates its intentions at install time through its description and at runtime through on-screen cues, users' expectations may change between the time that they install an app and the time that they use it. If an app extracts information in a way that is not indicated in either its description or at runtime, then it clearly violates visual privacy. If an app clearly indicates through its description what information it intends to extract, then it does not violate visual privacy. However, as with remote processing, an app that fails to clearly describe its behavior in its description but provides runtime cues represents a visual-privacy gray area. We will return to this issue in Section 7.3.

5 SYSTEM OVERVIEW

Before we delve into the design of CamForensics, we first provide a high-level overview of the principles behind the design of CamForensics and then we provide implementation details.

5.1 Design Principles

CamForensics must be able to collect evidences of the image analysis performed on the captured image without modifying the app and

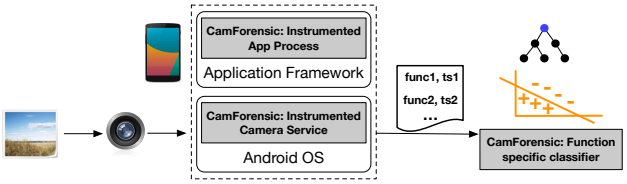

Figure 13: CamForensics has three major components: (1) Dynamic binary instrumentation to capture function logs during the run-time, (2) Instrumented camera service to trigger image analysis, and (3) A classifier to detect image analysis performed by an app.

with least amount of intervention. The following design principles guided our work.

Logs as an evidence of image analysis
The presence of an image processing library inside an app neither confirms nor denies its malicious nature. For example, OpenCV provides several image processing features such as color conversion (e.g., RGB to gray) and face detection. An app using OpenCV library for simple transformations may not pose a threat to user's privacy but the same performing face detection may.

CamForensics uses dynamic binary instrumentation technique to detect function calls in real-time and logs them along with the timestamps. Use of libraries such as OpenCV which provides one API for face detection, can easily be confirmed by the existence of those APIs in the logs. For other functionalities such as credit card reader, there are several functions involved such as edge detection and text extraction. In such cases, although, there are several different functions invoked, the sequence in which they are invoked remains fixed. Therefore, presence of a sequence of a function invocation becomes the evidence of a specific image analysis.

The order of function calls can become complex if the app uses multiple threads accessing the same third-party library. To avoid such a complication, CamForensics also records the thread id which made those function calls which in turn simplifies the ordering of the function calls.

Testing apps without pre-processing
One of the goals of CamForensics is to make the process of testing an app easy with least amount of human intervention. We want to avoid any pre-processing of the app package which involves peeking inside the app package before we begin to test the app.

The dynamic instrumentation technique employed by CamForensics helps us to run apps without pre-processing. Apps are run directly on the device and CamForensics takes control during the run-time as and when a third-party library functions are invoked.

Trigger image analysis
Often, for performance reasons, apps are designed to trigger image analysis if the image meets certain criteria, such as text extraction is triggered when the contrast is above a certain threshold and eye detection is triggered if the image has faces present. The app developer can create and embed such triggers in the apps. However, without access to the source code, it is difficult to know what triggers a specific type of processing on the image.

CamForensics tackles this problem by instrumenting how the images are delivered to the app from the camera sensor. In Android, `camera service` is the central component responsible for delivering images from camera sensor to the requesting app. CamForensics modifies the `camera service` to replay a pre-recorded video stored on the device indefinitely. These videos are recorded in a manner that every frame contains the object of interest such as faces or text. This makes our testing less cumbersome and more consistent. Using this technique an app to be tested can be invoked without worrying about where the device camera is pointing. When running, CamForensics collects function traces in the background.

Automate detection of image analysis
Inferring the kind of image processing from manual analysis of traces is not a trivial task. This is especially true in the case of third-party libraries whose source code is not available. CamForensics uses a machine learning based technique to skim through arbitrarily long function logs collected during an app's run. Based on the classifier output, the kind of image processing is labeled.

To train our machine learning technique, we first identify a few trusted apps which perform certain types of image analysis (e.g., face detect) using a given image processing library (e.g., opencv). We run these trusted apps and use the generated logs for training. For each app under the investigation, we collect run-time logs and apply the trained classifier. CamForensics uses several of such pre-trained classifiers to test the existence of different types of image processing in the wild.

6 IMPLEMENTATION

CamForensics is built in three parts: (1) Module to instrument the app process, (2) Module to instrument `camera service`, and (3) Module to detect the image analysis at app's run-time. We implemented the first two modules by modifying Android Open Source Project (AOSP) version of Android 6.0.1. The third module runs on a server where traces of function calls from the device are uploaded in offline manner. Subsequent sections describe the implementation of these three modules.

6.1 Instrumenting the app process

Intel's Pin [19] is one of the most powerful, robust, and efficient platform for run-time process instrumentation. It provides APIs that are easy-to-use and portable across multiple CPU architectures. A running instance of the app is a process, which when loaded in memory is has four segments: stack, heap, data, and code. Pin can be used to instrument arbitrary memory location of the code segment of the process. Pin instruments app in such a way that it is transparent to the process at run-time. That is to the running instance of a Pin instrumented app, all memory locations and register values appear the same as they would without the instrumentation. This guarantees consistent run-time behavior of the app before and after the instrumentation. To improve the performance of instrumented process and amortize its overhead, Pin uses a code cache to store previously instrumented copies of the app. We used Pin framework for it's simplicity and performance reasons. Further, we developed our own *Pintool* to log function calls to the native libraries for all android apps.

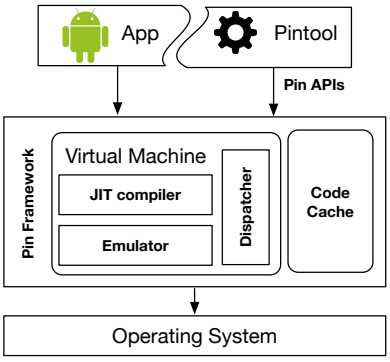

Figure 14: CamForensics's software architecture for instrumenting the app process.

Figure 14 shows overall software architecture for instrumenting app's process. *Pintool* is implemented in the form of a script written in C/C++. It contains the instrumentation code that we intend to insert in the process at run-time. When an app is invoked, the operating system makes a calls to Android run-time (ART) to start a new process. ART performs all housekeeping associated to starting a new process and returns assigned process-id to the operating system. We changed Android source code to make a callback to *Pintool* whenever the operating system finishes loading a process. At this point, *Pintool* automatically attaches itself to the process id supplied by the callback.

Next, *Pintool* attempts to identify the third-party native libraries used by the process. To accomplish that, it pauses the running process using the Unix Ptrace API and reads the symbol table using the API `PIN_Init()`. The read symbols identify third-party libraries used by the process. Based on the identified library names, *Pintool* decides whether a recognizable image processing library is present in the app. If present, *Pintool* continues further analysis of the app.

For each recognizable image processing library present in the app, two cases exist: (a) library is open-sourced (e.g, opencv) in which case *Pintool* monitors calls to a specific set of functions (e.g, `detectMultiScale`), and (b) the library is closed-sourced, hence all function calls are monitored. For functions we intend to monitor, their addresses are registered with *Pintool* using the API `RTN_AddInstrumentFunction()`.

For each registered function, Pin framework acts as a just-in-time compiler. It adds the instrumentation code only when those functions are invoked. Pin framework uses the code cache to store the instrumented code which is executed every time the function is executed. Finally, the *Pintool* uses the API `PIN_StartProgram()` to start the paused app process.

Pintool collects the following information every time a function from a third-party native library is invoked: (1) thread id, (2) function name, (3) name of the library containing the function, and (4) timestamp at which the function was invoked. *Pintool* is designed to spew the recorded events periodically on the device. When the app process is killed *Pintool* detaches itself. We would like to note that hijacking another process using a *Pintool* is not a security flaw. In order to get such an access, the device has to be rooted and the default selinux policy enforcements have to be disabled.

6.2 Camera Service - Record and Replay

Testing camera based apps is a tedious process. This is because of manual effort involved in running the camera and pointing it at a meaningful subject such as face or text. Several apps require running camera for few hours to detect transient malicious behavior (invocation of vulnerable functions). Moreover, an ideal experiment should be repeatable to confirm the vulnerability. In such a case, it is implausible to hold the camera and keep pointing to a particular subject. While one may use a camera mount, we take an alternate approach.

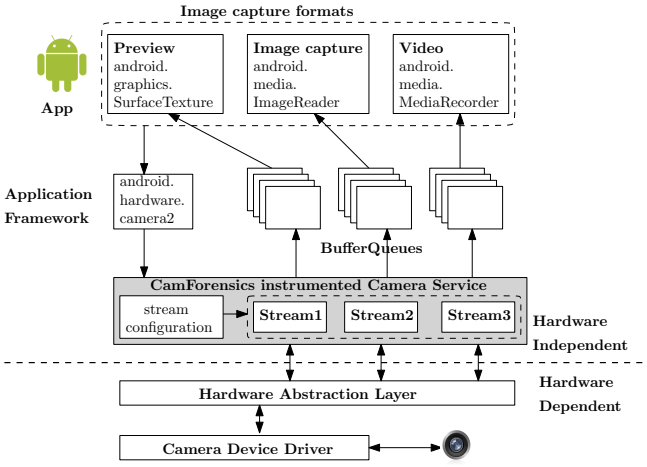

Figure 15: Android's camera subsystem. The colored module represents that CamForensics intercepts the image data coming from the camera sensor and replaces it with it's own data before delivering it to the app.

In Android, all the image based transactions between the camera sensor and any app are mediated by a specialized system service, `camera service` (Figure 15). `Camera service` is a trusted service which reads data from the sensor and supplies it to the layers above. This provides us an opportunity to populate any app with specific images or scenes without modifying its code. At a high-level, we modify `camera service` in a way that it starts reading from a video file instead of the camera sensor. `Camera service` then supplies fabricated frame buffer to the requesting app. We feed data of our choice to make app trigger specific operations – considered sensitive in nature. Moreover, this method also provides us consistency across app testing. We create a configuration file accessible to system service. The configuration file has two information: `mode` and `filename`. We modify the `camera service` to read the configuration file when an app requests camera access. Based on the `mode` in the configuration file, `camera service` does either of the following: (1) When `mode` is 0 (record): deliver the camera frames to the app, record the frames, and store them in a file pointed by `filename`, and (2) When `mode` is 1 (replay): read frames from the file pointed by `filename` and deliver them to the app (instead of frames from the sensor).

In this paper, we focus on detecting visual leaks containing text or human faces. Therefore, the frames that we used to replay for testing contained either text or human faces only.

6.3 Classifying the logs

Traces generated by CamForensics are neither easy nor feasible for a human to analyze. We want to have a system that can look at a log and determine the category of image analysis it belongs to. For this purpose, we use a very simple machine learning technique and implement it using the open-source framework, TensorFlow.

We look at the problem of mapping a trace to a functionality (such as text extraction) as the problem of determing a topic for a given sentence or a document. First, we convert the function trace from a series of string input to a set of learning vectors using *word2vec*, an unsupervised model by Mikolov et al [21]. Such representations of words in a vector space help learning algorithms to achieve better performance in classifying similar words. Note, here a word represents a function call to a native library. Next, we use these word vectors and train a shallow convolutional neural network (CNN) for classification [15].

The combination of *word2vec* and CNN forms an easy to setup classification unit which can be run on simple desktop to perform classification in the order of seconds.

7 EVALUATION

To evaluate CamForensics, we first measure its accuracy of the prediction in classifying an image processing task. Next, we measure the overhead introduced by CamForensics. Finally, we analyze the function and `logcat` trace for apps performing image analysis and present our findings. We specifically look for the apps which use third-party native libraries for: (a) barcode reader, (b) text extraction, (c) face detection, and (d) augmented reality.

7.1 Accuracy of classification

Developers may design their apps in such a way that can lead to a different sequence of functions to perform the same operation. For example, some apps choose to improve the contrast of the image and threshold the image before they perform text extraction, while others tend to detect paragraph before they perform text extraction. This heterogeneity of function call sequences makes the problem non-trivial. Therefore, to automatically detect the presence of a known image analysis we employ topic modeling technique. The aim is to provide a document containing function names, thread ids and timestamps, and determine the image analysis (topic of the document) performed on the camera data.

We set up our machine learning based classifier on a laptop with a 2.6 GHz Intel Core i5 processor with 8 GB RAM. We run an app performing text extraction (apps that use `libtess.so`) and collect function call traces. We use these traces to perform the training. Next, we run other apps which use the same library for text extraction and collect traces for testing. In total, testing logs contain 1000 instances of camera data on which text extraction is performed. We also provide 1000 frames in which no text extraction was performed.

We randomly split the dataset into 80-20 partition for training and evaluation corresponding to 1600 function logs for training and 400 logs for evaluation. We repeat the training and evaluation of the classifier across 10 different splits each of which are selected randomly. Our choice of the CNN training parameters leads to a very quick training period ($\approx$ 20 minutes). Our measurement shows that the median precision and recall for the classification is 0.96 and 0.966 respectively.

7.2 Overhead of using Pin

CamForensics does put additional performance overhead on the app. The performance drop comes from two components: (1) Pin Framework that resides between the app process and the OS, and (2) *Pintool* which injects instrumentation code and traces function calls. To measure the drop in performance, we perform matrix multiplication of size 200×200 on the device for 50 times. We measured total time taken in performing this task with Pin framework under the following scenarios: (1) without any instrumentation, (2) with instrumented function calls to one library, and (3) with instrumented all function calls to all libraries (including function calls to system libraries such as linker). We compare the time taken for the aforementioned scenarios with respect to the baseline where the task was run without the Pin framework.

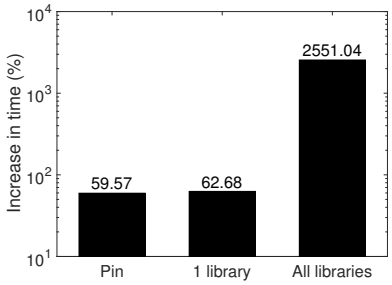

Figure 16: Increase in time-taken to perform 50 matrix multiplications of size 200×200. The y-axis is in the log scale and the value on top of the bar represents increased time in percentage.

Figure 16 shows that running image multiplication on Pin framework without instrumenting any functions increases total task completion time by 59.57%. With more libraries instrumented, the overhead depends upon the number of functions instrumented. When only one library is instrumented, the increased overhead is 62.68% and when all, it is 2551%.

7.3 Categorized Findings

Barcode reader: We analyzed top-200 apps out of 10K apps which contain Zbar (`libzbarjni.so`). Interestingly, on manual inspection, we find that 45 of them do not provide any camera-based functionality. Similarly, another 15 apps do not use barcode scanning and CamForensics correctly identified them. Needless to say that all 60 of these apps request camera permission in their Manifest file and include Zbar in the package binary. We also find that 9 apps bundle multiple barcode libraries, namely, Zbar, Zxing and Google native barcode scanner. This might be due to a common malpractice of app developers to start a new app development by copying from an older repository or let unused libraries reside despite discontinuing or enhancing a feature.

Since most barcode scanning apps are benign, an adversary can build on that trust to get side channel access of camera data. Later,

it may process data locally or transmit without informing the user which is considered as breach of visual privacy as per our definition in user study (Section 3.2). We observe similar behavior in Alive OneScan Pro app. This app during scanning a QR code recognizes that the user is wearing brown colored boots and therefore shows advertisements of similar products. We are surprised by such level of recognition. Our deeper inspection reveals that the app uses a cloud-based recognition service CraftAR to achieve that. In our case, users intention of scanning the barcode is misused to send colored images to the cloud. Further, we check the app's description where it mentions AR as an additional feature. However, it does not mention about images being transmitted to the cloud.

Another very popular app (number of installations 1 – 5M) called Tesco Lotus uses Qualcomm developed AR SDK, Vuforia. However, it does not mention AR in the app store description. At the start of app's activity, it sends device's identity to a server `https://t.appsflyer.com/` with identifiers in plain text. It also sends tracking information periodically to the same server. From the trace logs, we also find that a file on remote AWS server is updated based on the user's activity(`http://tesco2015.s3.amazonaws.com/game/57ff5cea1b571.zip`).

Similarly, popular brand MANGO app uses Vuforia Augmented Reality SDK for interactive shopping experiment on the website. MANGO allows its users to scan fashion model pictures in the web catalog and purchase product worn by them. MANGO periodically accesses `https://www.mango.com/` to show results based on the content of the viewfinder. Again, the app description of MANGO does not mention of AR toolkit or transmitting data out of the device. In total, we find a total of 87 such instances of co-existences of a barcode reader and Vuforia SDK in our dataset.

OCR: We instrumented over 200 apps which use tesseract library (`libtess.so`) for OCR. A majority of these apps use tesseract for document scanning, language translation, and dictionary. Tesseract library is the most popular library for converting one language words to another via camera. Tesseract is also used in assistive apps, where documents or objects follow a fixed template. For example, Power Meter app uses it to automatically extract usage reading, passport scanner for flight reservation, or driver license scanner for car rental. Lastly, several apps perform text recognition on live camera preview, such as to translate navigation signs from a foreign language.

As we inspect these apps, we find that dictionary-based apps state the use of OCR library clearly in their description. Unfortunately, this is not the case with others as they do not mention text recognition in their description, instead highlight other features. However, during usage, they provide some indication of ongoing text processing.

An interesting finding of our study is how OCR library is used in practice. While some attempt to directly extract the underlying words, some try to detect paragraphs before that. Moreover, from the system log it appears that OCR library uses best effort approach. Which means it recurses on the image and tries to optimize various thresholds, scales, quality, etc. until a result is found (or the timer runs out). For example, in the case when enough lines of text are not detected, it tries to increase the contrast of the image. Such approach results in a different series of function calls for different images. This behavior is unlike others, for example in the case of barcode scanning every image follows the same code path.

OpenCV and Face Detection: We instrument 240 apps using OpenCV library (`libopencv_java.so`) and analyze their traces. We find 10 them to use face detection functionality (function call instances of `detectMultiScale`). The face detection functionality is used in two categories of apps: face-based screen unlocking and fashion makeup.

The screen unlocking apps train a classifier from multiple photographs of users face. Later, they use this classifier to recognize the device owner, hence enabling the unlocking functionality. Since these apps explain the entire process up front, they run face detection only after users approval. Considering these apps seek explicit approval, they do not violate users visual privacy. On contrary, the fashion app (`com.modiface.lakme.makeuppro`) runs the face detection right after launching, therefore violates users visual privacy.

OpenCV is mostly used for functionalities other than face detection. The color conversions (`cvtColor`), image I/O (`imwrite`), and edge detection (`canny`) are amongst most frequently used functions. On several instances, we find that apps use other third-party libraries such as a closed source library `libprocessing.so` to perform face detection despite loading OpenCV.

Moreover, a large number of apps access face data directly through the camera drivers. Since face detection is a popular feature needed by a large number of apps, many devices build it in the hardware. Android also provides native APIs for detecting faces in software. In our analysis, 120 apps use native camera driver for active face detection.

Augmented Reality: In our dataset, we find two popular AR SDKs, namely Vuforia by Qualcomm and CraftAR. Although, Vuforia (3092 apps) is more popular than CraftAR (25 apps), we find CraftAR (`libCraftARSDK.so`) more interesting and relevant because it emphasizes object recognition both ways: on-device and in the cloud.

Apart from running apps through CamForensics, we also capture network packets to detect events where a large chunk of data is sent (an indication of possible image transmission). We find that a majority of the apps are marker-based, where the apps first download a database and later match visual features locally, on the device. This is expected because image transmission requires a high upload bandwidth which may not be ubiquitously available.

However, we find 10 apps that send data over the network. In all these apps, the packet send timestamps are tightly correlated to camera trigger event. While we can not comment with full certainty – it is a strong indication that an information related to the camera may be transmitted. Also, we see several log messages such as `CraftARSDK (3.0): Searching image in the server`. Among these 10 apps, only two 'Alive One' and 'Alive One Pro' explicitly display that they send images over the network (on the camera preview). For all such apps, we see a continuous stream of 590 bytes packets. However, combined packet size of transmission does not match the image size. This could be because compressed image size differs significantly from the raw image based on the algorithm used. Moreover, compressed image size also varies with the content in the

scene, hence unpredictable. While this analysis is inconclusive in saying whether the apps send images, features, or something else; it does present a gray area. Such widespread information hiding does raise privacy concerns important for end users.

8 FUTURE WORK

Though CamForensics has proved to be a useful tool for studying visual privacy, its limitations point to several directions for future work. In this section, we highlight the limitations and discuss possible future directions.

The user study described in Section 3.1 was very simple which used only the app's name and its description. In reality app stores, usually, contain several screenshots for an app. Such visual depictions may help the user to understand the kind of image processing involved. Also, some apps provide runtime cues such as point the camera to your face to unlock the device. A user study based on users' experience of using the apps can provide a better estimate of the discrepancy between user's awareness and apps' descriptions.

CamForensics requires some manual effort to gather and analyze app logs. In particular, a researcher must manually explore an app's UI to trigger camera access. Second, while a simple classifier can categorize the kind of image processing is performed, manual effort is needed to infer details such as where the image data or the analysis is being sent. These limitations are reasonable for a study such as ours, but developing a visual-privacy service will require taking humans completely out of the loop. Thankfully, UI exploration could be automated using platform accessibility APIs, and advanced machine learning techniques may prove useful for inferring behavior from log data.

In addition, due to the way the tool relies on the sequence of native method calls, CamForensics can detect access to a version of the native library on which it was trained. In a different version of the same native library, its developer may change the name of the functions, add or remove certain functions. This will change the function call signature to perform the same image processing task. However, this issue can be tackled by providing more and more training samples gathered from the apps which use a different version of the same native library.

CamForensics incurs performance overhead due to the use of dynamic binary instrumentation platform. Dynamic monitoring for visual privacy is a very appealing alternative because it has the potential to recognize violations at the moment they occur. Improving CamForensics to reduce the overhead and integrating runtime visual-privacy checks into mobile platforms is an exciting direction for future work.

9 RELATED WORK

Third-party native library studies: To the best of our knowledge, this paper presents the first study which investigates the third-party native library usage by camera-based apps. There are numerous studies which estimate the trend and growth of native libraries in Android for other purposes [2, 27, 34]. Sun et al. [27] in 2014 investigate top apps from Google PlayStore and find that an average app contains 4 native libraries. Zhou et al. [34] analyze 204K apps from various online stores and report that only 4.52% of the apps contain native libraries, a significantly smaller number.

Violating user expectation: Several apps request more permissions than required by the underlying feature against user's expectations. Studies suggest that the users aware of the nuances of the app do not tend to allow the resource access as often [9, 32]. Researchers have developed Whyper [23] – a tool to estimate the excess permissions requested by an app based on its description.

Untrusted code isolation: Klinkoff et al. [16] focus on native code isolation in *.Net* application framework [30]. Similarly, Siefers et al. [25] present Robusta, a system for native code isolation of Java applications. However, Robusta requires non-trivial modifications to Java Virtual Machine (JVM). NativeGuard [27] is an orthogonal system which uses a similar mechanism for native code isolation but in Android OS. Comparatively, CamForensics is a much simpler approach which does not require complex modification to the Android OS.

Application behavior analysis: CamForensics can be envisioned as a tool to analyze app behavior. Recent work on analyzing app behavior can be categorized in two categories: (a) static, and (b) dynamic. ScanDroid [10] and FlowDroid [3] perform static analysis and may not be sufficient when an app uses third-party libraries. In such cases, dynamic analysis is required. TaintDroid [7] and DroidBox [17] are two popular tools which support taint analysis of Dalvik instructions across API calls to catch privacy leaks. Both of these systems rely on extensive instrumentation of operating system, penalizing the run-time performance.

Using machine learning: In recent times, machine learning (ML) based approaches have shown to outperform traditional whitelisting approaches such as Centroid [4] and AdRisk [11] in detecting advertising libraries. Narayanan et al. develop AdDetect [22] for automatic semantic detection of in-app ad libraries. AdDetect applies hierarchical app package clustering, an ML technique, to detect advertising libraries. Similarly, PEDAL [18] detects advertising libraries by training a classifier based on package relationship and testing them against the code-features from library SDKs. DroidSec [33] is a recent system that uses deep learning model for malware detection – achieving 96% accuracy on the real-world Android apps. AnDarwin [5], WuKong [31] and LibRadar [20] detect app clones by filtering library code using clustering techniques. The underlying assumption is that the library source code does not change and, hence, can be bypassed easily.

10 CONCLUSION

This paper presents the results of a large-scale study of visual privacy in the wild. An app violates visual privacy if it extracts information from camera data in ways that a user does not expect. To study visual privacy leaks, we develop CamForensics that monitors how camera data is handled by third-party libraries at runtime. Using CamForensics, we characterize how over 600 Android apps extract information such as text, faces, and QR codes from devices' camera. In addition, we perform several surveys to characterize what information users expected these apps to extract based on their app store descriptions and to gauge their attitudes toward visual privacy. Our results show that apps frequently defy users' expectations, based on their descriptions, and that users care about how apps process their camera data.

REFERENCES

[1] Paarijaat Aditya, Rijurekha Sen, Seong Joon Oh, Rodrigo Benenson, Bobby Bhattacharjee, Peter Druschel, Tong Tong Wu, Mario Fritz, and Bernt Schiele. 2016. I-Pic: A Platform for Privacy-Compliant Image Capture *(MobiSys)*.

[2] Vitor Afonso, Antonio Bianchi, Yanick Fratantonio, Adam Doupé, Mario Polino, Paulo de Geus, Christopher Kruegel, and Giovanni Vigna. 2016. Going Native: Using a Large-Scale Analysis of Android Apps to Create a Practical Native-Code Sandboxing Policy. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*.

[3] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *ACM SIGPLAN Notices* 49, 6 (2014), 259–269.

[4] Kai Chen, Peng Liu, and Yingjun Zhang. 2014. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 175–186.

[5] Jonathan Crussell, Clint Gibler, and Hao Chen. 2013. Andarwin: Scalable detection of semantically similar android applications. In *European Symposium on Research in Computer Security*. Springer, 182–199.

[6] Soteris Demetriou, Whitney Merrill, Wei Yang, Aston Zhang, and Carl A Gunter. 2016. Free for all! assessing user data exposure to advertising libraries on android. In *NDSS*.

[7] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. 2014. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)* 32, 2 (2014), 5.

[8] Miro Enev, Jaeyeon Jung, Liefeng Bo, Xiaofeng Ren, and Tadayoshi Kohno. 2012. SensorSift: Balancing Sensor Data Privacy and Utility in Automated Face Understanding *(ACSAC)*.

[9] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. 2011. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 627–638.

[10] Adam P Fuchs, Avik Chaudhuri, and Jeffrey S Foster. 2009. Scandroid: Automated security certification of android. (2009).

[11] Michael C Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. 2012. Unsafe exposure analysis of mobile in-app advertisements. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*. ACM, 101–112.

[12] Google Play Store in Numbers. 2016. `https://techflow.me/2014/03/05/my-project-google-play-store-in-numbers/`. (2016). [Online; accessed 3-Dec-2016].

[13] Suman Jana, David Molnar, Alexander Moshchuk, Alan Dunn, Benjamin Livshits, Helen J. Wang, and Eyal Ofek. 2013. Enabling Fine-Grained Permissions for Augmented Reality Applications With Recognizers. In *USENIX Security*.

[14] Suman Jana, Arvind Narayanan, and Vitaly Shmatikov. 2013. A Scanner Darkly: Protecting User Privacy from Perceptual Applications. In *S & P*.

[15] Yoon Kim. 2014. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882* (2014).

[16] Patrick Klinkoff, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2007. Extending. NET security to unmanaged code. *International Journal of Information Security* 6, 6 (2007), 417–428.

[17] P Lantz, A Desnos, and K Yang. 2016. DroidBox: Android application sandbox. `https://github.com/pjlantz/droidbox`. (2016). [Online; accessed 4-Dec-2016].

[18] Bin Liu, Bin Liu, Hongxia Jin, and Ramesh Govindan. 2015. Efficient privilege de-escalation for ad libraries in mobile apps. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 89–103.

[19] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, Vol. 40. ACM, 190–200.

[20] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. 2016. LibRadar: fast and accurate detection of third-party libraries in Android apps. In *Proceedings of the 38th International Conference on Software Engineering Companion*. ACM, 653–656.

[21] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*. 3111–3119.

[22] Annamalai Narayanan, Lihui Chen, and Chee Keong Chan. 2014. AdDetect: Automated detection of Android ad libraries using semantic analysis. In *Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), 2014 IEEE Ninth International Conference on*. IEEE, 1–6.

[23] Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. 2013. Whyper: Towards automating risk assessment of mobile applications. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*. 527–542.

[24] Nisarg Raval, Animesh Srivastava, Ali Razeen, Kiron Lebeck, Ashwin Machanavajjhala, and Lanodn P. Cox. 2016. What You Mark is What Apps See *(MobiSys)*.

[25] Joseph Siefers, Gang Tan, and Greg Morrisett. 2010. Robusta: Taming the native beast of the JVM. In *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 201–211.

[26] 2016 Q2 Smartphone OS Market Share. 2016. `http://www.idc.com/prodserv/smartphone-os-market-share.jsp`. (2016). [Online; accessed 3-Dec-2016].

[27] Mengtao Sun and Gang Tan. 2014. NativeGuard: Protecting android applications from third-party native libraries. In *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*. ACM, 165–176.

[28] A tool for reverse engineering Android apk files. 2016. `https://ibotpeaches.github.io/Apktool/`. (2016). [Online; accessed 3-Dec-2016].

[29] Nicolas Viennot, Edward Garcia, and Jason Nieh. 2014. A measurement study of google play. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 42. ACM, 221–233.

[30] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. 1994. Efficient software-based fault isolation. In *ACM SIGOPS Operating Systems Review*, Vol. 27. ACM, 203–216.

[31] Haoyu Wang, Yao Guo, Ziang Ma, and Xiangqun Chen. 2015. Wukong: A scalable and accurate two-phase approach to android app clone detection. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM, 71–82.

[32] Primal Wijesekera, Arjun Baokar, Ashkan Hosseini, Serge Egelman, David Wagner, and Konstantin Beznosov. 2015. Android permissions remystified: A field study on contextual integrity. In *24th USENIX Security Symposium (USENIX Security 15)*. 499–514.

[33] Zhenlong Yuan, Yongqiang Lu, Zhaoguo Wang, and Yibo Xue. 2014. Droid-sec: Deep learning in android malware detection. In *ACM SIGCOMM Computer Communication Review*, Vol. 44. ACM, 371–372.

[34] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. 2012. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets.. In *NDSS*, Vol. 25. 50–52.