What You Mark is What Apps See

Nisarg Raval, Animesh Srivastava, Ali Razeen, Kiron Lebeck†,
Ashwin Machanavajjhala, Landon P. Cox
Duke University
† University of Washington

ABSTRACT

Users are increasingly vulnerable to inadvertently leaking sensitive information through cameras. In this paper, we investigate an approach to mitigating the risk of such inadvertent leaks called *privacy markers*. Privacy markers give users fine-grained control of what visual information an app can access through a device's camera. We present two examples of this approach: *PrivateEye*, which allows a user to mark regions of a two-dimensional surface as safe to release to an app, and *WaveOff*, which does the same for three-dimensional objects. We have integrated both systems with Android's camera subsystem. Experiments with our prototype show that a Nexus 5 smartphone can deliver near real-time frame rates while protecting secret information, and a 26-person user study elicited positive feedback on our prototype's speed and ease-of-use.

1. INTRODUCTION

Cameras are pervasive and multiplying. All modern PCs and smartphones have cameras, as do most gaming consoles and televisions. Furthermore, emerging wearable computers, household robots, and Internet-of-things devices provide a glimpse of the not-too-distant future: continuous recording by crowds of nearby devices. This proliferation of cameras has created a growing sense that the most confidential details of a person's life are perpetually at risk of leaking.

Our goal is to develop tools that give users fine-grained control over the information that apps can access through a camera. Preventing all leaks, particularly those by determined attackers, is likely impossible, but preventing inadvertent leaks by trusted cameras is much more feasible. Such inadvertent leaks will be problematic for all users, but enterprises are particularly vulnerable since workers are surrounded by secrets and opportunities to accidently leak information abound. For example, video chatting with an external collaborator could accidentally reveal sensitive information drawn on a background whiteboard, and a photo of a receipt on a desk could also capture portions of other nearby documents.

Prior work on protecting visual secrets has relied on computer vision to identify classes of objects within images. Vision algorithms can be used to transform images so that (1) they contain only objects that an app is allowed to view [9, 10, 20] (e.g., providing a gesture-recognition application with only an outline of a user's hands), or (2) they do not contain any sensitive objects [4, 6, 16] (e.g., blurring background faces).

Unfortunately, both approaches suffer from the same limitations. First, system designers are unlikely to anticipate all objects that users will want to hide or reveal. Absent classifiers will either lead to more leaks or hinder legitimate apps' functionality. However, even for object classes that a designer correctly anticipates, developing recognition algorithms that are accurate, precise, and efficient enough to handle realtime analysis is challenging. For example, it is unclear how one would develop practical recognizers for items as nuanced as sales figures, meeting notes, and receipts.

Due to these limitations, we have been investigating an alternative approach based on *privacy markers* [14]. Compared to prior work, privacy markers provide a natural way for users to express fine-grained access-control policies over visual information while also simplifying the system software responsible for enforcing restrictions. In this paper, we describe the design and implementation of two privacy-marker systems: *PrivateEye* and *WaveOff*.

PrivateEye allows users to mark a two-dimensional region by enclosing it with a special shape. The shape can be drawn by hand (e.g., on a whiteboard with standard markers) or embedded in a digital document (e.g., on a slide using presentation software). A PrivateEye-enabled device recognizes the shape in its camera stream at runtime and marks those portions of the data. WaveOff allows users to mark three-dimensional objects through a special user interface (UI). The WaveOff UI shows a live camera feed, on top of which users can specify a bounded region containing the object they wish to mark. WaveOff extracts visual features for the object from the region and stores them in a database so that objects can be identified in future images.

MobiSys'16, June 25-30, 2016, Singapore, Singapore

DOI: http://dx.doi.org/10.1145/2906388.2906405

Alice is attending a client presentation in Shanghai. Since some of the text is in Mandarin, Alice would like to verify that she translated the text correctly using a third-party translation app on her mobile device. The app captures slides from the presentation, runs optical character recognition on each frame, and displays the translated text on her mobile device. However, the slides also contain sensitive corporate secrets (e.g., a table of earnings predictions) that the client does not want to leak. Both Alice and the client are uncomfortable that her translation app may forward the secret information to untrusted servers for processing.

(a) Scenario one: Presentation

Bob is video chatting on his laptop with collaborators. He would like to use his whiteboard to work out a problem, and hence would like his collaborators to see it. However, he has a confidential product roadmap on his whiteboard that his collaborators should not see. He also has a prototype model of a newly designed drone and a bottle of medication on his desk. Bob would like to show the prototype to his collaborators, but would be upset if his collaborators saw his medication.

(b) Scenario two: Video chat

Cathy relies on an augmented-reality app on her mobile device to superimpose indicators over her keyboard to help her input a complex password. While she is comfortable allowing the password app to view her keyboard, she does not want the app to capture her computer screen or anything else on her desk.

(c) Scenario three: Password entry

Figure 1: Hypothetical scenarios.

To prevent apps from accessing secret information, users put their device into a restricted privacy-mode. In privacy-mode, PrivateEye and WaveOff convert raw camera data into a stream of *public regions* defined by the privacy markers they detect. That is, when a user puts her device into privacy-mode, PrivateEye and WaveOff block all unmarked regions of the camera stream so that apps can only access regions that have been explicitly marked public. Using markers to define public regions was a key design decision. In our experience, computer-vision algorithms are not robust enough to handle practical considerations such as motion-induced blur and changing light conditions at usable framerates. Blocking unmarked regions provides strong assurances that apps cannot access sensitive information without unduly limiting app functionality.

We have integrated PrivateEye and WaveOff with a Nexus 5 smartphone running Android. In a 26-person user study with our prototype, participants successfully used PrivateEye and WaveOff to mark public regions containing QR codes, and reported that marking and scanning the QR code was easy and fast. Furthermore, using a benchmark of representative videos, our prototype blocked at least 99% of non-public regions while supporting at least 20 frames per second.

The rest of this paper is organized as follows. Section 2 motivates the problem of protecting visual secrets and our approach of using privacy markers. Section 3 provides a design overview. Section 4 describes our prototype implementations of PrivateEye and WaveOff. Section 5 describes the results of our user study as well as a comprehensive evaluation of our prototype's privacy-utility tradeoffs, and its performance. Related work is discussed in Section 6. Section 7 provides our conclusions.

2. USE CASES AND MOTIVATION

Figure 1 describes three scenarios that capture how camera-driven apps can put secret information at risk: a *presentation* containing sensitive material, a *video chat* with sensitive information visible in the background, and a *password-entry* app that should only view a small number of objects.

The simplest solution for protecting visual secrets is *coarse-grained blocking*. That is, users can turn off their cameras when sensitive information is in view. However, in all three of our scenarios, secret information and application-essential information are co-mingled. Alternatively, a system can take a more fine-grained approach to protect visual secrets. One fine-grained approach is to apply the principle of *least privilege* to visual content and give applications access to only the visual information required for their functionality [9, 10]. Under least privilege, applications must request access to classes of objects within a camera's view. If a request is granted, the system can use computer vision to ensure that only allowed objects of interest pass from a camera's view to an application.

Systems that take a least-privilege approach must anticipate the classes of objects that applications will want to access, and provide *recognizers* to accurately and efficiently detect these objects. For instance, a study of Kinect applications found that 87 only needed access to objects identified by four recognizers [9]. However, as illustrated by our use cases, one is unlikely to anticipate all classes of objects that mobile apps may need to access.

Thus, least-privilege systems that use predefined recognizers are likely to provide strong security, but support fewer applications. For example, in the "video chat" scenario, if the application can only access faces and hands, then it cannot capture more spontaneous moments, such as sharing an equation written on the whiteboard. Predefined recognizers will also fail when nuanced differentiation between objects

is required, such as between the sensitive and non-sensitive slides in the "presentation" scenario.

A second fine-grained approach is to *block secrets* by defining sets of sensitive objects and using computer vision to remove those objects from an application's view [4, 6, 16]. As with least privilege, these systems require designers to anticipate a large universe of objects, which vision algorithms must detect and block. For example, to protect a confidential roadmap from a video-chat application, the system would have to construct a recognizer for product roadmaps.

In light of these challenges, this paper expands on our earlier proposal of *privacy markers* [14], which provide a promising new approach to mitigating camera-based leaks. At a high level, privacy markers consist of two parts: (1) a simple interface for marking objects in the physical environment, and (2) device software for efficiently recognizing marked objects. PrivateEye and WaveOff are privacy-marker systems that take a least-privilege approach. When a user puts her device into privacy-mode, apps can view only marked objects through the camera. This approach provides solutions to each scenario described in Figure 1.

In the "presentation" scenario, Alice's client would enclose most of her slides in a special rectangle. The client trusts that Alice is using a mobile device equipped with PrivateEye that will only reveal the marked slides to her translation app. Unmarked slides containing secret information will be blocked from the app's view. In the "password" scenario, Cathy can use WaveOff on her mobile device to mark any keyboard she uses to enter a password. Thereafter, Cathy can safely allow the password-manager app to view the camera input and be assured that everything except the marked keyboard will be blocked from its view. Finally, in the "video chat" scenario, Bob can mark his face, the part of the whiteboard that he would like to share via video, and his prototype drone model using PrivateEye and WaveOff before starting the chat session so that only those objects are visible to his collaborators.

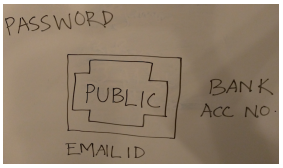
3. APPROACH OVERVIEW

This section provides an overview of our two privacy-marker systems, PrivateEye and WaveOff, including the attacker model and principles underlying their designs. We also summarize the systems' limitations.

3.1 Trust and attacker model

We assume that recording devices run a combination of trusted and untrusted software. The computer-vision software needed to recognize and track marked regions must be part of the trusted computing base. This software must reside in the camera subsystem of the platform, either integrated with the camera driver or with a trusted camera service.

We are primarily concerned with inadvertent leaks by untrusted, third-party software, such as video-chat and receipt-scanning apps. We assume that untrusted software can only access camera data through well known APIs defined by the device platform, and that privacy-marker software is properly isolated from third-party software.



(a)

(b)

Figure 2: (a) A region on a whiteboard marked as public using PrivateEye's marker. (b) A coffee mug marked as public using WaveOff.

Our trust model is based on settings such as an enterprise in which a company may purchase video-chat stations and employee equipment like laptops and smartphones. We cannot prevent a determined attacker from capturing secrets using a malicious recording device, such as an analog video recorder or a digital device that ignores users' markers.

3.2 Design principles

PrivateEye and WaveOff must be easy to use and efficiently block all non-public regions from a camera's view at runtime. The following design principles guided our work.

Avoid special equipment

Ease-of-use was a primary goal for PrivateEye and WaveOff. Before arriving at our current designs, we considered alternatives, such as affixing or projecting QR codes onto surfaces and objects. However, we rejected these approaches as being too inconvenient in practice. Ideally, a user should be able to mark an object or region without special equipment.

PrivateEye is used for marking content on two dimensional surfaces like whiteboards and presentations. Its marker is a pair of concentric rectangles delineating a public region as shown in Figure 2a. The inner rectangle has truncated corners to differentiate it from other unmarked objects with concentric rectangles (e.g., window frames). This marker is unusual and easy to draw, either by hand (e.g., using whiteboard markers) or using digital tools (e.g., in a presentation).

WaveOff is used for sharing objects that are difficult to physically mark, such as three-dimensional objects. To mark these objects, we take advantage of the recording device's screen by overlaying a bounding box on a live camera feed, as seen in Figure 2b. The marking UI is part of an app that is trusted to handle unmodified camera data.

Simplify recognition

A potential drawback of simplifying life for users is that it can create complexity for the vision algorithms needed to identify objects. Hence, we carefully designed markers for PrivateEye and WaveOff that play to computer vision's strengths.

The concentric rectangle marker in PrivateEye addresses the nearly impossible task of differentiating between pub-

lic and private text.[1] Public text can be enclosed within a marked region, and all other text is private by default.

At the same time, PrivateEye markers can be easily detected by vision algorithms due to the strong contrast of straight, dark lines on a light two-dimensional surface. Such features are apparent under low resolution and regardless of the lines' colors. Moreover, by choosing inner rectangles with truncated corners, the marker is easier to detect since detecting corners is more robust and faster than detecting edges. Furthermore, this design distinguishes our marker from other parts of a camera's view that contain concentric rectangles (e.g., window frames).

PrivateEye detects privacy markers using a combination of computer-vision algorithms. For each frame, it first detects edges in a frame using the well-known Canny algorithm [2]. After edge detection, the system detects contours in the edge-based image using Suzuki's algorithm [18]. The contours are stored in a tree hierarchy structure. Each contour has information about four other contours: (a) parent - the contour that encompasses the current contour, (b) previous - the previous sibling contour, (c) next - the next sibling contour, and (d) child - the contour which is completely inside the current contour. Upon detecting a contour that can be approximated as rectangle [5] (i.e., a convex area polygon with four vertices), PrivateEye searches its children contours. If PrivateEye finds a contour that has 12 corners with area at least 50% of the parent rectangle contour, the system concludes that it is a marked region.

WaveOff recognizes objects by extracting features from the bounded area of a camera's view to build a model of the objects within and stores the model in a database. We would like to note that WaveOff recognizes a specific object (e.g., Cathy's Mac keyboard) rather than a class of objects (e.g., any Mac keyboard). This simplifies recognition since the model can use all distinctive keypoints (including those that might not be present in other objects of the same type) without worrying about over-fitting. In order to build the model, WaveOff computes descriptors at selective keypoints in the marked region of the camera's view.

The keypoints are the distinctive regions (e.g., corners and edges) of the object and descriptors are the features at those keypoints (e.g., gradient and orientation). We use BRISK [12], a set of binary features, to compute keypoints and the corresponding descriptors. Computing and matching BRISK features is efficient and is ideal for realtime performance on low-power mobile devices.

WaveOff uses feature matching to detect the presence of a marked object in a given frame. For each frame, it computes the BRISK features and matches them with the features of models in the database. Since matching is independent for every object, they can be identified in parallel. WaveOff unblocks a region when at least 20% of its features match a model in the database.

[1]We do not utilize optical character recognition due to poor accuracy and computational intensity.

Track to mitigate recognition failures

Recognizing privacy markers on every incoming camera frame is computationally intensive and can cause a low frame rate. Furthermore, recognizers can fail if the camera's viewpoint changes significantly. Building recognizers for multiple viewpoints is costly and inconvenient. We address both issues by tracking features since tracking is robust to minor changes in viewpoints and faster than object recognition.

Once PrivateEye detects a marked region, it extracts prominent corners in the region (using [17]) and tracks these features using the Lucas-Kanade optical-flow-in-pyramids technique [21]. WaveOff also tracks object features across consecutive frames using Lucas-Kanade. While recognition is essential when an object appears in a stream for the first time, it is less helpful when the object is fully visible, because tracking alone is sufficient to identify the object's location in subsequent frames. Hence, we skip recognition altogether when tracking finds enough features (at least 75% of the features stored in the model).

However, when there are not enough features, tracking alone can introduce localization errors that propagate across subsequent frames. Hence, we use both matching and tracking to find a larger set of features when tracking alone is insufficient. We estimate an object's location in the camera's view using CMT [13].

Block under uncertainty

For privacy-sensitive applications, blocking the entire camera view might be acceptable when the system is uncertain about the presence of public regions. In fact, in our user study, we found that initially blocking entire frames for several seconds while scanning a QR code did not impact the utility of the app. Based on this observation, we take a conservative approach and only reveal a region when we are confident that it has been marked. However, blocking the entire frame under uncertainty may create a poor user experience when an object is blocked frequently after it appears. This can happen when a frame becomes blurry due to motion. To address this issue, during times of uncertainty, the system displays the previous frame (in which the object was visible) instead of showing nothing. This approach does not harm security because it does not reveal any new information, but it can help usability. Note, that if the view changes significantly (i.e., the object disappears), we quickly detect the change and stop showing the old frame.

3.3 Limitations

Though PrivateEye and WaveOff are robust in many settings, their reliance on standard computer-vision algorithms limit their security guarantees and can impinge on apps' utility.

First, both systems require users to mark public regions with a rectangle. As a result, it is possible for sensitive information to inadvertently appear within a marked area and cause a leak. For example, consider a presentation with slides that have been marked public. If the speaker or anyone else steps in front of the slides, then their identity will

be revealed regardless of whether this should be kept secret. Similarly, if a user marks an object with WaveOff as public, but a secret object is placed in front of it without completely occluding the public object, then an app may view the secret object. We believe that experience using privacy markers would reduce the likelihood of such incidents, but they could never be eliminated.

Second, a camera could zoom so far into a region marked by PrivateEye that the marker falls outside the camera's field of view. In this case, PrivateEye would block the content of the public region. While this would not be insecure, it would hurt the utility of an app that needs access to the blocked region. This could be particularly problematic for large, marked regions with very small details inside; the only way to take a picture would be from a far enough distance that the marker was visible, but at this distance little of the content may be legible.

Finally, while PrivateEye and WaveOff allow users to mark arbitrary objects, they only allow objects to be treated as public or private. Thus, when a device is in privacy-mode, an app can access *all* public regions. It is easy to imagine scenarios in which a user may want to restrict an app to a narrow subset of private objects, such as our hypothetical password-entry app. For WaveOff, one could provide a UI that allows users to specify which marked objects should be revealed to which apps. Unfortunately, it is much less clear how to support rich access-control policies for PrivateEye, such as making a document recordable for executives' smartphones but not engineers'. Increasing marker expressiveness would likely compromise usability and recognition performance (i.e., accuracy, precision, and efficiency), and understanding these tradeoffs is beyond the scope of this paper.

4. IMPLEMENTATION

We implemented PrivateEye and WaveOff by modifying the Android Open Source Project (AOSP) version of Android 5.1. Our prototype currently runs on a Nexus 5 smartphone. Before we describe the implementations of PrivateEye and WaveOff, we first provide some background on Android's camera subsystem. We then present the design of an initial implementation that recorded video at an unacceptable four frames-per-second (FPS). We then highlight the causes of this poor performance and describe techniques that allow us to record video at a median framerate of 20 FPS.

4.1 Android's camera subsystem

In Android 5, Google made significant changes to the camera subsystem to give apps more control over the camera feed. Figure 3 shows how the subsystem is split between hardware-dependent and hardware-independent software layers. The hardware-dependent layer consists of the camera device-driver that interacts directly with the physical camera, and a hardware abstraction layer (HAL) that implements a common interface for the camera service. Because the camera service is hardware independent, it is the ideal place to integrate PrivateEye and WaveOff.

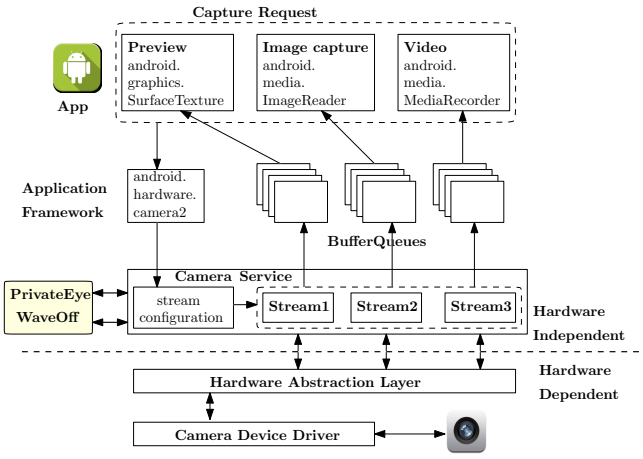


Figure 3: Android's camera subsystem. The colored box denotes PrivateEye and WaveOff, which intercept all incoming frames before they are passed to the application framework.

An app can submit a *capture request* to the camera service in one or more of the following modes: (a) preview, (b) image capture, and (c) video recording. In addition to a mode, each capture request describes a set of image attributes, such as resolution and pixel format. For instance, an app may request access to the camera in preview mode at a resolution of 1280×960, or in image-capture mode, with the image stored in a file at a resolution of 640×480.

The camera service creates one *stream* (an internal buffer) for each capture mode. The camera service and HAL can access all streams, and each stream is protected by its own lock. Multiple streams can be active at at time, e.g., a preview stream and a video-recording stream, and the active streams are configured according to their corresponding capture-request attributes. The application framework also creates a *BufferQueue* for each capture mode so that the camera service can send image data to an app. When the camera driver delivers a frame, HAL locks all active streams, copies image data to those streams according to their configuration parameters, and then releases the locks. To forward frames to an app, the camera service acquires all active streams' locks, copies image data from the streams to their corresponding BufferQueues, and releases all locks.

4.2 PrivateEye and WaveOff in Android

In this section, we present our Android implementations of PrivateEye and WaveOff. We start with a naive design that highlights the challenges of achieving realtime performance, and then describe how we overcame these challenges.

4.2.1 A simple implementation

Our initial implementation integrated PrivateEye and WaveOff into Android as follows: (1) the camera service locked each active stream and sequentially passed each stream's image data to PrivateEye or WaveOff; (2) PrivateEye or WaveOff detected any public regions and masked the rest of the

image; (3) the camera service copied each modified image to its corresponding BufferQueue and released all locks.

This straightforward implementation resulted in unacceptable video-recording performance. In particular, on our Nexus 5 we could only record video at a median rate of four FPS. This was due to two factors. First, the camera service held all active-stream locks while waiting for PrivateEye and WaveOff to identify public regions within a frame. This blocked the HAL for long stretches, and caused it to drop frames. Second, streams with different resolution and pixel format still contain most of the same visual information. Our initial implementation ignored similarities across streams, e.g., between a preview and video-recording stream, and needlessly analyzed each frame independently. Our current implementation addresses both of these problems.

4.2.2 Improved implementation

PrivateEye and WaveOff are separate modules loaded by the camera service when the camera is turned on. We rely on OpenCV for all computer-vision algorithms. Each module exposes an API of six calls to the camera service, which is described in Table 1.

Both PrivateEye and WaveOff modules maintain one queue for incoming frames, $InQueue$, and a second queue for processed frames, $OutQueue$. The camera-service thread blocks until it receives a frame from HAL, and then passes the frame to PrivateEye and WaveOff via a `processFrame` call. We currently set InQueue's maximum depth to 15, and calls to `processFrame` will add a new frame to InQueue as long as there is enough room. If there are already 15 frames in the queue, `processFrame` drops the new frame. An alternative approach would have been to make room for the new frame by dropping the oldest frame in InQueue, but we found that this made tracking objects across frames more difficult. All calls to `processFrame` return as soon as the passed-in frame has been copied onto InQueue or dropped.

To process frames in InQueue, PrivateEye and WaveOff each have a dispatch thread that blocks waiting for frames to be added to InQueue. After dequeueing a frame from InQueue, the dispatch thread hands off the frame for further processing. Processed frames are eventually forwarded back to the camera service by placing them in OutQueue.

After calling `processFrame`, the camera-service checks for processed frames by calling `isFrameAvailable`. If a frame is available in OutQueue, `isFrameAvailable` dequeues the frame and returns it to the camera service. The camera service then copies the processed frame to the appropriate BufferQueues. Allowing the camera service to accept new frames from HAL before older ones have been processed prevents the HAL from dropping frames. However, if PrivateEye and WaveOff cannot analyze frames fast enough, InQueue will fill up and cause the camera service to drop frames.

To make frame processing faster, PrivateEye is implemented as a four-stage pipeline, with a separate thread for each stage. The four stages are: (1) downsize a frame and convert it to grayscale, (2) detect marker contours, (3) track previously detected markers and merge them with newly detected markers, and (4) mask all non-public regions and add the final image to OutQueue.

WaveOff's performance depends on the number of object models in its database, and for each model WaveOff spawns one thread. Each thread is responsible for matching and tracking a specific public object. However, before tasking the object threads, WaveOff's dispatch thread dequeues a frame from InQueue and makes a single pass over the data to compute all image features. It then passes these features to the object threads before they begin their work. The last object thread to complete its analysis aggregates the results of the others to render the final frame.

Video recording

When multiple streams are active, e.g., the preview and video-recording streams, our initial implementation sequentially and independently processed a frame from each source. This effectively doubled the work that PrivateEye and WaveOff had to perform even though the visual content of each stream was nearly identical. To eliminate redundant analysis, PrivateEye and WaveOff only process frames from the preview stream. After processing a preview frame, they add the modified frame to OutQueue and save the coordinates of all public regions so that they can be applied to other frames. In particular, for non-preview streams, the camera service uses the `getLastRects` method to retrieve the coordinates of any public regions that were identified in the preview stream. It then masks the non-preview frame after adjusting for the stream's settings, such as its resolution.

In addition, to re-using processing results across multiple streams, PrivateEye and WaveOff applied several other optimizations. First, analyzing frames in the Nexus 5's original resolution of 1280×960 takes several seconds. Therefore, PrivateEye and WaveOff resize each frame to 400×240 and 320×240, respectively, before performing any compute-intensive operations.

In addition, PrivateEye and WaveOff operate on grayscale frames to reduce processing times. The camera-preview stream delivers frame data in the full-color YUV format, and OpenCV converts a YUV frame with resolution 1280×960 to grayscale in about 30 ms. However, for a 1280×960 image, the YUV format uses the first 1280×960 bytes to store an image's luminance data, which encodes a grayscale version of the frame. We found that simply copying the luminance data of a YUV frame into a separate buffer creates a grayscale version of the frame in under one ms.

Finally, PrivateEye and WaveOff avoid allocating and de-allocating memory whenever possible. The camera service pre-allocates all circular buffers, queues, and other data structures when it loads our modules. We only allocate new memory when a stream's configuration, such as its resolution, changes, and we only de-allocate memory when the camera is turned off.

Image capture

Capturing an image with an active preview stream is slightly different than video recording. When a user takes a picture, the HAL returns a JPEG-compressed byte stream rather than

Method	Description
processFrame	sends a camera frame data for processing
isFrameAvailable	checks if a frame is available for the delivery to application
getLastRects	returns a list of regions marked public in the last frame
addObjectModel	builds and adds model of the marked public object in model database
removeObjectModel	removes a model of the marked public object from the model database

Table 1: PrivateEye and WaveOff API for interacting with the camera service.

a YUV frame. The coordinates returned by `getLastRects` correspond to raw pixels, and thus the JPEG image must first be converted before it can be masked. After masking any non-public regions, the camera service converts the result back to a JPEG and forwards it to the appropriate BufferQueue.

AOSP provides jpeglib (version 6b) for encoding and decoding JPEG data. However, this version of the library only decodes and encodes JPEG data through the file system. Thus, we avoided costly system calls by adding two support functions to perform in-memory encoding and decoding.

Finally, to read and write to the image-capture JPEG stream in the camera service, we modified the Nexus 5 HAL implementation provided by LG. Each stream has a usage flag that determines how the stream will be accessed by different components in the camera subsystem. We added two flags to the JPEG stream: `GRALLOC_USAGE_SW_READ_OFTEN` and `GRALLOC_USAGE_SW_WRITE_OFTEN`. These flags direct all JPEG data to the camera service for reading and writing. These small changes were the only modifications we made to a hardware-dependent component.

5. EVALUATION

To evaluate PrivateEye and WaveOff, we sought answers to the following questions:

- What is the perceived burden of using PrivateEye and WaveOff?
- Do PrivateEye and WaveOff interfere with app functionality?
- How well do PrivateEye and WaveOff detect public regions, and under what conditions?
- How well do PrivateEye and WaveOff perform when the number of objects increases?
- What are the energy and performance overheads of PrivateEye and WaveOff?

To answer the first two questions we conducted a user study with 26 participants. The study was approved by our university Internal Review Board (IRB), and involved marking and scanning QR codes on flat and curved surfaces. We answered the next question by developing a benchmark with videos of typical settings and camera movements. We used these videos to measure how precisely, accurately, and efficiently our prototype would handle these scenarios. We evaluated scalability using experiments with multiple marked regions and objects. Finally, we answered the last question by measuring our prototype's resource usage with the camera preview enabled.

As mentioned previously, our prototype is based on AOSP for Android 5.1 and runs on a Nexus 5 smartphone.

5.1 User Study

To better understand PrivateEye and WaveOff's impact on usability we conducted a user study in which participants marked public regions containing QR codes and then used a third-party app to scan the codes.

5.1.1 Study design

The aim of our user study was to understand how privacy markers affect the experience of using a camera-enabled app. We randomized participants into two groups. Members of the *control group* performed tasks on a Nexus 5 running unmodified AOSP, and members of the *case group* performed tasks on a Nexus 5 running AOSP augmented with PrivateEye or WaveOff.

Both groups performed simple tasks with a QR-code scanning app. Marking and scanning a QR code provided a limited but informative context for understanding PrivateEye and WaveOff's usability. First, QR-code scanning is a common smartphone task that requires realtime image processing. Second, input images must be clear enough for accurate and fast scanning. Third, scanning codes allowed us to measure usability quantitatively (e.g., time to scan and scanning accuracy) and qualitatively (e.g., user-perceived ease and speed). We used the *Barcode Scanner*[2] app for our tests. At the time of testing, this app was freely available on the Google Play Store, had been downloaded more than 100 million times, and had received an average rating of 4.1 out of 5.

We initially asked each participant to complete a short pre-study questionnaire about their familiarity with smartphone apps and, in particular, camera-enabled apps. Then we explained the purpose of the study and described the tasks they would perform. Each control session involved using AOSP to scan a QR code affixed to a whiteboard and scanning a QR code affixed to a coffee mug. Case sessions were similar, except that participants marked the QR codes before scanning them. After completing these tasks, we asked each participant to fill out a post-study questionnaire about their experience. If a participant could not scan a QR code within one minute, we asked them to repeat the experiment at most three times.

To ensure that volunteers were comfortable performing their tasks, we showed them how to scan a QR code. We told them to start the app and point the camera at the QR code so that it stayed in focus in the center of the viewfinder. We also showed them how to adjust the position of the camera (e.g., near or far from the code) while scanning.

For PrivateEye, we showed volunteers a marker and demonstrated how to draw it on a blank sheet of paper. We stressed

[2]https://play.google.com/store/apps/details?id=com.google.zxing.client.android&hl=en

that the marker lines needed to be thick and continuous. Because participants had not seen the PrivateEye marker before, we asked them to practice drawing a marker on paper and pointed out any mistakes they made.

We similarly explained how to mark an object using WaveOff. First, we showed them how to mark a cup using the WaveOff UI. We emphasized that the marked object should be clearly visible and fully enclosed by the UI's bounding box. Once participants understood how to mark an object, we asked them to mark the coffee mug and to scan its QR code.

To measure how quickly and accurately participants scanned QR codes, we created a testing app. The app allowed users to launch *Barcode Scanner* via an Android Intent. After a successful scan, our testing app received the captured image. Using this app, we recorded the time to scan each QR code and whether the scan was successful.

We also measured the time participants took to mark a public region. For PrivateEye, we manually measured marking time with a stopwatch. For WaveOff, the marking UI logged the time taken to capture an image of the target object, to adjust the marker bounding box, to extract the object's features, and to store the object's model.

Finally, we designed a short post-study questionnaire that asked participants to respond to two statements and provided an optional section for general feedback. The questionnaire statements were '*Completing the study tasks was fast*' and '*Completing the study tasks was easy*'. Users reported their level of agreement with each statement on a scale from one (strongly disagree) to seven (strongly agree).

5.1.2 Recruiting participants

We recruited participants by sending messages to university mailing lists and posting on Facebook. We asked potential participants to fill out a demographic questionnaire on a website, and based on the completeness of the responses, we selected 26 volunteers for our study.

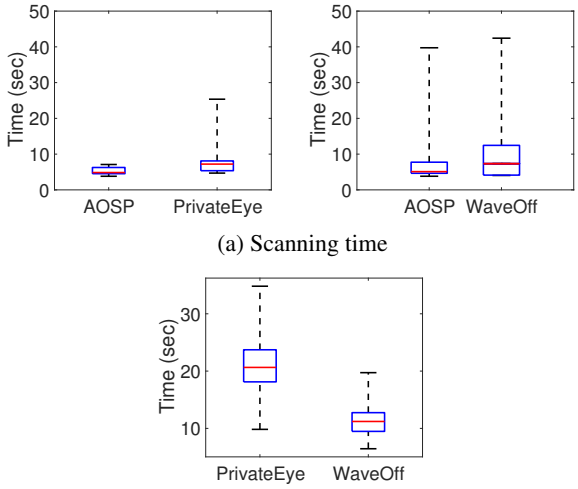
Among the 26 participants, 19 were male and seven were female. The median age was 27, with the youngest 23 and the oldest 53. Due to soliciting participation through university mailing lists and Facebook, 20 of the 26 had some kind of computer-science degree.

Many of our participants frequently used their smartphone camera. Seven reported using their smartphone camera at least ten times per week; eight reported video chatting at least twice per week; 14 reported using at least three different camera-enabled apps each month. Finally, many participants had used a smartphone to scan documents, with 16 reporting that they had scanned a check and seven reporting that they had scanned a receipt.

We randomly divided our 26 volunteers between two equally sized control and case groups. As an incentive to participate, we gave each volunteer a $10 Amazon gift card once they had completed the study.

5.1.3 Results

Figure 4b shows how long participants took to mark the regions around the QR codes. The median time taken to draw a marker on the whiteboard was 20.64 seconds, whereas the median time for marking a coffee mug was 11.2 seconds. While most participants felt that the marking scheme for WaveOff was simple, 20% of participants reported that the PrivateEye marker was complex. Two participants needed two attempts to correctly draw the PrivateEye marker.



(a) Scanning time

(b) Marking time

Figure 4: User-study results: (a) Time taken by users to scan a QR code with and without using our systems. (b) Time taken by users to mark a QR code with PrivateEye and WaveOff.

Each participant correctly scanned the QR codes and completed the study within 10 minutes. The 100% success rate indicates that PrivateEye and WaveOff preserve enough information in camera frames to support applications like QR-code scanning.

Figure 4a shows how long participants took to scan QR codes under PrivateEye and WaveOff compared to a baseline of unmodified AOSP. Our AOSP control group took a median of 4.9 seconds to scan a QR code on a whiteboard and a median of 5.5 seconds to scan a code on a mug. One member of the control group took nearly 40 seconds to scan the mug code because they initially held the camera too close.

Scan times for case-group members using PrivateEye and WaveOff were comparable to the baseline. For PrivateEye, the median scan time was 7.2 seconds, and for WaveOff the median scan time was 7.4 seconds. The main reason that case participants took longer is that PrivateEye and WaveOff block all camera input to an app until they have detected a marked region. As a result, users needed to guess the initial location of the QR codes before they became visible through the display.

Despite the additional time, participants rated the speed and ease of using PrivateEye and WaveOff on par with using AOSP. Figure 5 shows users' average levels of agreement with statements that scanning codes was fast and that scanning codes was easy. Users chose their level of agreement on a scale of one to seven, where one meant strong disagreement and seven meant strong agreement.

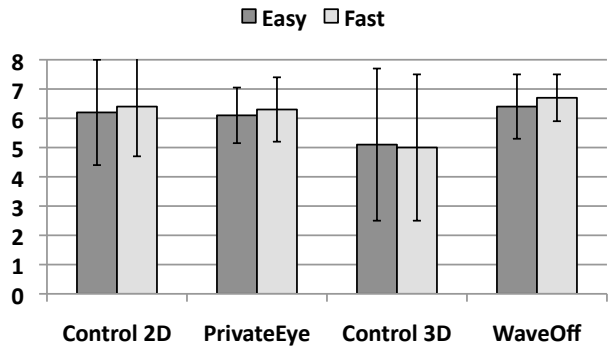


Figure 5: On a scale from one (strong disagreement) to seven (strong agreement), user-study participants' average levels of agreement that scanning QR codes was easy and fast. Error bars represent standard deviations.

When rating the speed of scanning the whiteboard code, the average levels of agreement were 6.4 (standard deviation 1.7) and 6.3 (standard deviation 1.2) for AOSP and PrivateEye, respectively. When rating the speed of scanning the mug code, the average levels of agreement were 5 (standard deviation 2.5) and 6.7 (standard deviation 0.8) for AOSP and WaveOff, respectively. When rating the ease of scanning the whiteboard code, the average levels of agreement were 6.2 (standard deviation 1.7) and 6.1 (standard deviation 0.9) for AOSP and PrivateEye, respectively. When rating the ease of scanning the mug code, the average levels of agreement were 5.1 (standard deviation 2.6) and 6.4 (standard deviation 1.1) for AOSP and WaveOff, respectively.

The high standard deviation in ratings for the control groups scanning under AOSP is due to two participants' requiring multiple attempts to scan the code. We found that in general, it is hard to scan a QR code on the curved surface if the camera is not positioned properly. This did not impact the WaveOff experiment because participants initially marked the object through the camera preview, and therefore were more careful when positioning the camera while scanning the code.

There were several notable results from our user ratings. First, PrivateEye's perceived ease-of-use and speed were statistically equivalent to unmodified AOSP. Second, for AOSP, users' felt that scanning a QR code on a flat surface was easier and faster than scanning a code on a curved surface. This makes sense since scanning a code on the mug using AOSP took longer, on average, than scanning a code on the whiteboard. Yet despite being quantitatively slower, WaveOff's perceived ease-of-use and speed were greater than PrivateEye's. Even more surprising, WaveOff's perceived ease-of-use and speed were greater than AOSP's!

We believe that these counter-intuitive results were due to users' mental model of the camera subsystem, and that WaveOff primed video streams for fast QR-code capture. Under WaveOff, the camera-preview screen was completely black until it recognized the coffee mug, and users likely attributed the blank screen to a camera delay rather than to WaveOff. Furthermore, the camera preview only appeared when a frame was clear enough for WaveOff to recognize the mug, which meant that the frame was also clear enough to scan the code. As a result, users likely scanned the QR code almost immediately after the camera preview appeared.

Note that PrivateEye also kept the screen blank until it recognized the whiteboard marker, but users rated PrivateEye the same as AOSP. This is because recognizing the PrivateEye marker was faster than recognizing either the mug or QR code, which caused a greater delay between the time the preview appeared and the time the QR code could be scanned than under WaveOff. As a result, WaveOff likely seemed faster and easier to use than either PrivateEye or AOSP.

5.2 Video benchmark

To characterize how well PrivateEye and WaveOff perform under a variety of settings and camera movements, we developed a simple video benchmark. Each video in the benchmark lasted 10 seconds, was shot with our Nexus 5 at a resolution of 1280×960 at 30 FPS, and captured public regions in controlled settings using predefined camera movements.

In particular, we shot videos of a public region on three two-dimensional surfaces and a three-dimensional object in three settings. To test two-dimensional surfaces we marked public regions on a whiteboard, a paper document, and in a presentation slide shown on a laptop screen. To test three-dimensional objects, we placed a white coffee mug alone in front of a white background, next to a soda can in front of a white background, and alone with a PC monitor and a keyboard in the near background.

We also designed three camera movements to understand how motion affected PrivateEye and WaveOff. Under the *still* movement, the camera was steady and remained focused on the marked region for the entire video. This movement approximated taking a picture. Under the *spin* movement, the camera rotated while keeping its view centered on the same region. This camera movement helped us understand orientation changes. Finally, under the *scan* movement, public regions moved in and out of the camera's view. For the first three seconds of these videos, a public region was centered. Then the camera panned to the right so that the public region disappeared from view. Lastly, the camera panned left so that the region returned to its original position, where it remained for final three seconds.

All together, our benchmark consisted of 18 videos: each two-dimensional surface captured under each camera movement, plus each three-dimensional setting captured under each camera movement. To test our PrivateEye and WaveOff implementations, we modified the Android camera subsystem to load pre-recorded videos from memory rather than from the HAL. After loading a video frame, we passed it along the same path that live frames pass, i.e., through our PrivateEye and WaveOff modules and ultimately to a video-recording app. Using this setup we evaluated PrivateEye and WaveOff's precision, recall, and performance.

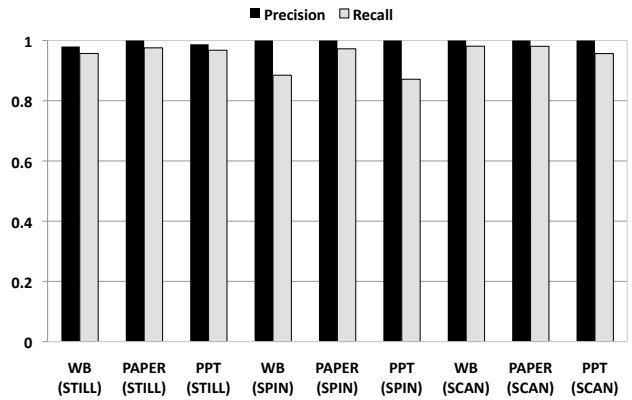


Figure 6: Median precision and recall for PrivateEye analyzing video frames of a public region on a whiteboard (WB), on a paper document (PAPER), and in a presentation slide on a laptop screen (PPT) under steady (STILL), rotating (SPIN), and panning (SCAN) camera movements.

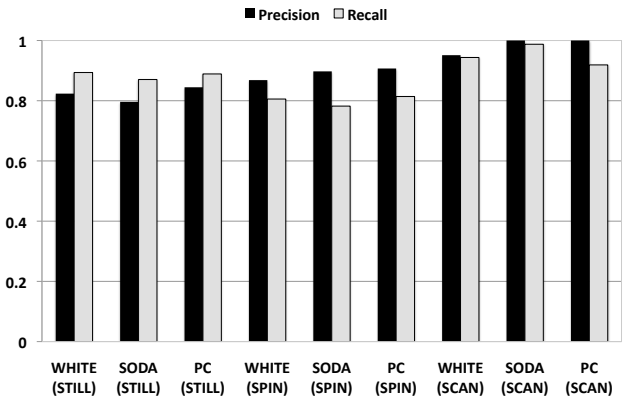


Figure 7: Median precision and recall for WaveOff analyzing video frames of a white coffee mug in front of a white background (WHITE), the mug next to a soda can in front of a white background (SODA), and the mug in front of a PC screen and keyboard (PC) under steady (STILL), rotating (SPIN), and panning (SCAN) camera movements.

5.2.1 Precision and recall

To calculate precision and recall, we partitioned each video frame into a grid of 5×5 pixel cells. Since we captured the original videos at a resolution of 1280×960 resolution, each frame contained 49,152 cells. We manually labeled all cells containing part of a marked region or object as *public*. We labeled all other cells *private*. Public cells should be revealed to applications, and private cells should be blocked from applications.

For each frame in the output videos, we analyzed which public cells PrivateEye and WaveOff revealed to compute precision and recall. We considered a cell blocked if *none* of it appeared in the frame; otherwise, we considered the cell to be revealed. More formally, let pb_t be the set of true public cells and pb_r be the set of cells revealed in a given frame. For all output videos, we calculated precision and recall for each frame as follows:

$$precision = \frac{|pb_t \cap pb_r|}{|pb_r|} \qquad recall = \frac{|pb_t \cap pb_r|}{|pb_t|}$$

Precision captures the percent of revealed cells that were truly public, and is a measure of a system's security. A perfectly secure system would reveal only true public cells, and thus have a precision of one. However, a system that blocked all cells would also be perfectly secure but would not support applications. Thus, precision must be considered in tandem with recall. Recall captures the percent of true public cells that a system revealed, and is a measure of how compatible the system is with applications. Similar to precision, if a system revealed all cells it would have perfect recall but provide no security. An ideal system would score highly on both precision and recall.

Figure 6 shows the median precision and recall under PrivateEye for frames in nine videos of two-dimensional surfaces. PrivateEye's lowest median precision was 0.98 for frames of a whiteboard with the camera still. In seven of the nine videos, PrivateEye exhibited a median precision of one. These results indicate that PrivateEye will provide strong security for many surfaces and under a variety camera movements.

It is worth noting that PrivateEye's median precision was greater than its median recall for all videos. This is a consequence of our design decision to prioritize security by revealing marked regions instead of blocking them. Our results show how blocking marked regions can compromise security. In particular, camera motion would cause systems that block marked regions to expose sensitive information, whereas PrivateEye exhibited a median precision of one for all spin and scan video frames.

It may seem counter-intuitive that PrivateEye's precision was lowest when the camera was still, i.e., 0.98 and 0.99 for the whiteboard and presentation slide, respectively. Across all videos, PrivateEye most often incorrectly revealed cells near the border of the marked region because marker lines were not perfectly straight, which led to slight differences between the marked region and PrivateEye's mask. In frames with motion blur, as in the spin and scan videos, border-cell features became fuzzy and did not register with our tracking algorithm. Still videos contained many more frames with clear marker boundaries, leading to lower median precision across the entire video.

PrivateEye's lowest median recall was 0.87 for the on-screen presentation slide with the camera rotating. In this case, the rotating camera created image blur that caused PrivateEye's tracking to incorrectly block cells just inside the public region. Nonetheless, in seven of the nine videos, PrivateEye exhibited a recall of 0.96 or greater. Overall, these results are highly encouraging and indicate that, despite providing strong security, PrivateEye is likely to reliably reveal public regions under most conditions.

Figure 7 shows the median precision and recall for frames under WaveOff. WaveOff's lowest median precision was

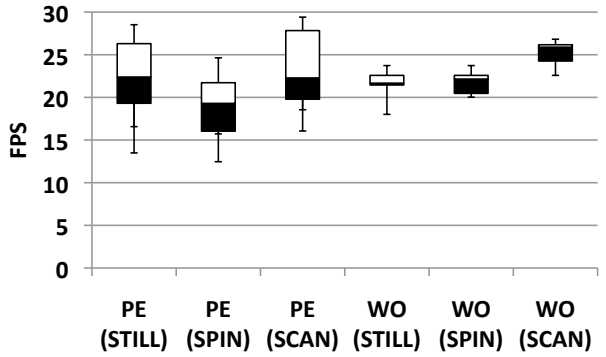


Figure 8: PrivateEye (PE) and WaveOff (WO) box plots for FPS under steady (STILL), rotating (SPIN), and panning (SCAN) camera movements.

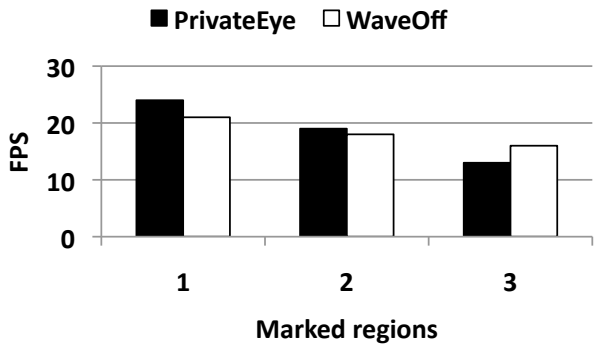


Figure 9: PrivateEye and WaveOff framerates with multiple marked regions.

0.79 for the still video when the coffee mug was next to a soda can. Precision scores for WaveOff in still videos with the mug front of a white background and in front of a PC were 0.82 and 0.84, respectively. As with PrivateEye, WaveOff's precision was lowest in still videos because they had no motion blur. However, mask mis-alignment was more acute with WaveOff than with PrivateEye because of the irregular shape of the coffee mug. WaveOff occasionally revealed cells close to the mug's handle when they should have been blocked. PrivateEye's rectangular marker was less prone to these errors.

WaveOff's lowest median recall scores were in spin videos. Its lowest median recall was 0.78 in the spin video with the mug next to a soda can. Recall was highest in scan videos because approximately 40% of the frames in these videos did not contain the mug. When the mug was absent, WaveOff correctly blocked all content and scored a perfect recall even though pb_t was empty. As with PrivateEye, median recall was lower in still videos because every frame contained the mug, and these frames provided more opportunities for WaveOff to incorrectly block cells near the border of the mug.

Overall, we were encouraged by WaveOff's benchmark results. Marking irregularly shaped three-dimensional objects is more challenging than marking two-dimensional regions with PrivateEye. In general, whenever WaveOff incorrectly blocked or revealed cells it was due to mask misalignment at the boundary of the coffee mug. Errors by PrivateEye and WaveOff were all localized to cells in the immediate vicinity of the marked regions. Consistent with our user-study, errors at the edge of an object seem unlikely to create major problems for applications or to leak sensitive information.

5.2.2 Performance

Supporting live camera feeds is critical for PrivateEye and WaveOff. Figure 8 shows box plots of PrivateEye and WaveOff's framerates on our video benchmark; the top whisker shows the max FPS achieved by a system for a collection of videos, the top of the white box shows the 75th percentile, the bottom of the white box shows the median, the bottom of the black box shows the 25th percentile, and the bottom whisker shows the minimum. The median framerate over all benchmark videos was 20 FPS and 22 FPS for PrivateEye and WaveOff, respectively. PrivateEye and WaveOff exhibited their highest median framerates on scan videos. The public regions and objects were absent for large portions of the scan videos, which eliminated the overhead of feature tracking on those frames. On the other hand, both systems exhibited their lowest median framerates on spin videos because camera rotation caused more tracking overhead. Finally, WaveOff achieved higher framerates than PrivateEye because WaveOff processes lower resolution images, i.e., 320×240 for WaveOff and 400×240 for PrivateEye. PrivateEye requires higher resolution images to reliably detect the concentric rectangle marker. Overall, these framerates are encouraging. Consistent with feedback from our user study, framerates between 20 and 25 FPS are acceptable for many camera-based applications.

5.3 Scalability

Our video benchmark evaluated PrivateEye and WaveOff using scenes with a single marked region or object. It is natural to ask how scenes with multiple marked regions and objects affect our systems' precision, recall, and performance. To answer this question, we recorded videos for PrivateEye with one, two, and three marked regions on a whiteboard, and a video for WaveOff with three objects (i.e., a soda can, green robot toy, and white coffee mug) in front of a white background. For WaveOff, we pre-computed models for all three objects, and loaded one, two, or three models at a time. The camera was still in all videos.

For both PrivateEye and WaveOff, the number of regions and objects did not have a significant impact on precision and recall. PrivateEye's average precision and recall were between 0.97 and 0.99 for all videos. WaveOff's recall declined from 0.95 to 0.90 after adding the toy model, but this was primarily due to its irregular shape. On the other hand, as Figure 9 shows, increasing the number of regions and objects decreased the framerates of both systems. The framerates for PrivateEye with one, two, and three public regions

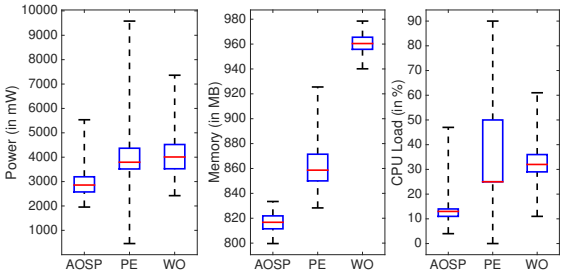


Figure 10: Power consumption, memory usage, and CPU load for AOSP, PrivateEye (PE), and WaveOff (WO) for a 60-second camera preview.

were 24, 19, and 13, respectively. The framerates for WaveOff with one, two, and three objects were 21 FPS, 18 FPS, and 16 FPS, respectively. These results suggest that PrivateEye and WaveOff are best suited to applications that require a small number of public regions per video frame.

5.4 Resource usage

Our final experiment was designed to measure the impact of PrivateEye and WaveOff on a mobile device's resource usage. We performed a simple experiment in which we turned on the camera and let it run for one minute in preview mode while measuring CPU load (in percentage), memory consumption (in MB) and power consumption (in mW). We took measurements every 100 ms using Trepn [8], a performance monitoring tool developed by Qualcomm. For this experiment, we set the smartphone on airplane mode to avoid measurement fluctuations due to network usage. We ran the experiment five times with unmodified AOSP, AOSP with PrivateEye, and AOSP with WaveOff, and rebooted our Nexus 5 between trials. For PrivateEye, the camera was focused on a public region, for WaveOff, the camera was focused on an object whose model was loaded in memory.

Figure 10 shows the power, memory, and CPU load results of our experiment. PrivateEye required a median of 40 MB more memory than AOSP. WaveOff's memory consumption was higher than PrivateEye's because it stores an object model in memory to perform matching. PrivateEye and WaveOff both increased the CPU load from a median of 11% to 26% and 31%, respectively. This CPU load was due to the detection and tracking algorithms applied on every frame during a camera session. As expected, power consumption exhibited a similar trend, with PrivateEye and WaveOff introducing median increases of 935 mW and 1150 mW, respectively. Overall, the additional resource usage of PrivateEye and WaveOff are reasonable, particularly since they only impose overhead when an app accesses the camera.

6. RELATED WORK

Several systems have applied the principle of least privilege to control fine-grained access to visual information [9, 20]. Recognizer [9] analyzes camera frames at runtime and limits third-party apps to accessing only the visual objects that they are authorized to view. SurroundWeb [20] limits 3D web browsers to skeleton views of rooms. An alternative but similar approach is to detect objects of interest and remove them [1, 4, 16]. Respectful Cameras use clothing such as special hats and vests to identify which users to remove from an image [16]. I-Pic uses a combination of short-range wireless signaling to disseminate policies and remote multiparty computation to transform captured images [1]. Privacy markers provide a more general way for users to specify what image data may be revealed to apps.

Another approach is to allow apps to manipulate transformed images through a high-level API rather than directly accessing them. Darkly [10] has some support for object-specific distortion (e.g., face morphing), but its primary means of restricting access to sensitive information are image-wide transformations (e.g., sketching) and limited access to the OpenCV API. However, because Darkly applies transformations to an entire image, it cannot support an arbitrary mix of high- and low-fidelity regions within an image as PrivateEye and WaveOff do.

Several projects [11, 19] protect visual content through sophisticated machine-learning tools. These systems require a great deal of training data and time, which limits their use to offline applications. PrivateEye and WaveOff both support realtime camera use.

Finally, many projects have tried to give users greater insight into and control over how apps use sensor data. Sensor-access widgets [7] graphically indicate when an app accesses sensor data, and SensorSift [6] automatically removes selected facial attributes, such as age, race, and expression, from images. ipShield [3] conveys privacy risks by listing inferences that could be drawn from sensor data. In world-driven access-control (WDAC) [15], real-world objects broadcast access-control policies to nearby recording devices. All of these projects are orthogonal to PrivateEye and WaveOff and could be easily integrated.

7. CONCLUSION

This paper made the case that *privacy markers* are a promising way to prevent third-party apps from inadvertently leaking visual information. We designed and implemented two privacy-marker systems, PrivateEye and WaveOff, that help users mark public regions in a camera's view and deliver only content within the public regions to apps. A user study with our prototype implementation revealed that users deemed camera operations with PrivateEye and WaveOff fast and easy-to-use. Other experiments with our prototype showed that PrivateEye and WaveOff provide strong security without compromising app functionality.

8. ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers and our shepherd Jakob Eriksson for their invaluable feedback. Our work was supported by Intel, Google, NSF award CNS-1253327, and DARPA and SPAWAR under contract N66001-15-C-4067.

References

[1] P. Aditya, R. Sen, S. J. Oh, R. Benenson, B. Bhattacharjee, P. Druschel, T. T. Wu, M. Fritz, and B. Schiele. I-pic: A platform for privacy-compliant image capture. MobiSys, 2016.

[2] J. Canny. A computational approach to edge detection. *TMAPI*, 1986.

[3] S. Chakraborty, C. Shen, K. R. Raghavan, Y. Shoukry, M. Millar, and M. Srivastava. ipshield: A framework for enforcing context-aware privacy. In *NSDI*, 2014.

[4] J. Chaudhari, S. Cheung, and M. Venkatesh. Privacy protection for life-log video. In *SAFE*, 2007.

[5] D. H. Douglas and T. K. Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: The International Journal for Geographic Information and Geovisualization*, 1973.

[6] M. Enev, J. Jung, L. Bo, X. Ren, and T. Kohno. Sensorsift: Balancing sensor data privacy and utility in automated face understanding. ACSAC, 2012.

[7] J. Howell and S. Schechter. What you see is what they get: Protecting users from unwanted use of microphones, cameras, and other sensors. In *Web 2.0 Security and Privacy*. IEEE, May 2010.

[8] Q. T. Inc. Trepn Power Profiler. https://developer.qualcomm.com/software/trepn-power-profiler. [Online; accessed 7-Dec-2015].

[9] S. Jana, D. Molnar, A. Moshchuk, A. Dunn, B. Livshits, H. J. Wang, and E. Ofek. Enabling Fine-Grained Permissions for Augmented Reality Applications With Recognizers. In *USENIX Security*, 2013.

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

[11] M. Korayem, R. Templeman, D. Chen, D. J. Crandall, and A. Kapadia. Screenavoider: Protecting computer screens from ubiquitous cameras. *CoRR*, 2014.

[12] S. Leutenegger, M. Chli, and R. Siegwart. Brisk: Binary robust invariant scalable keypoints. In *ICCV*, 2011.

[13] G. Nebehay and R. Pflugfelder. Clustering of Static-Adaptive correspondences for deformable object tracking. In *CVPR*, 2015.

[14] N. Raval, A. Srivastava, K. Lebeck, L. Cox, and A. Machanavajjhala. Markit: Privacy markers for protecting visual secrets. UbiComp '14 Adjunct, 2014.

[15] F. Roesner, D. Molnar, A. Moshchuk, T. Kohno, and H. J. Wang. World-driven access control for continuous sensing. Technical Report MSR-TR-2014-67, 2014.

[16] J. Schiff, M. Meingast, D. Mulligan, S. Sastry, and K. Goldberg. Respectful cameras: detecting visual markers in real-time to address privacy concerns. In *IROS*, 2007.

[17] J. Shi and C. Tomasi. Good features to track. 1994.

[18] S. Suzuki and K. Abe. Topological structural analysis of digitized binary images by border following. *Computer Vision, Graphics, and Image Processing*, 1985.

[19] R. Templeman, M. Korayem, D. Crandall, and A. Kapadia. PlaceAvoider: Steering first-person cameras away from sensitive spaces. In *NDSS*, 2014.

[20] J. Vilk, D. Molnar, E. Ofek, C. Rossbach, B. Livshits, A. Moshchuk, H. J. Wang, and R. Gal. Surroundweb : Mitigating privacy concerns in a 3d web browser. In *IEEE Symposium on Security and Privacy*, May 2015.

[21] J. yves Bouguet. Pyramidal implementation of the lucas kanade feature tracker. *Intel Corporation, Microprocessor Research Labs*, 2000.