Hidden in Plain Sight: Exploring Privacy Risks of Mobile Augmented Reality Applications

SARAH M. LEHMAN, Temple University, U.S.A.
ABRAR S. ALRUMAYH, Temple University, U.S.A.
KUNAL KOLHE, Stony Brook University, U.S.A.
HAIBIN LING, Stony Brook University, U.S.A.
CHIU C. TAN, Temple University, U.S.A.

Mobile augmented reality systems are becoming increasingly common and powerful, with applications in such domains as healthcare, manufacturing, education, and more. This rise in popularity is thanks in part to the functionalities offered by commercially available vision libraries such as ARCore, Vuforia, and Google's ML Kit; however, these libraries also give rise to the possibility of a *hidden operations threat*, that is, the ability of a malicious or incompetent application developer to conduct additional vision operations behind the scenes of an otherwise honest AR application without alerting the end user. In this paper, we present the privacy risks associated with the hidden operations threat, and propose a framework for application development and runtime permissions targeted specifically at preventing the execution of hidden operations. We follow this with a set of experimental results, exploring the feasibility and utility of our system in differentiating between user-expectation-compliant and non-compliant AR applications during runtime testing, for which preliminary results demonstrate accuracy of up to 71%. We conclude with a discussion of open problems in the areas of software testing and privacy standards in mobile AR systems.

CCS Concepts: • **Security and privacy → Mobile platform security**; *Vulnerability management*; Usability in security and privacy.

Additional Key Words and Phrases: augmented reality, mobile system security, user privacy

ACM Reference Format:
Sarah M. Lehman, Abrar S. Alrumayh, Kunal Kolhe, Haibin Ling, and Chiu C. Tan. YYYY. Hidden in Plain Sight: Exploring Privacy Risks of Mobile Augmented Reality Applications. *ACM Trans. Priv. Sec.* AA, B, Article CCC (August YYYY), 35 pages. https://doi.org/00.0000/1122334.4556677

1 INTRODUCTION

Augmented reality (AR) systems are pervasive sensing systems which leverage the contents and changes in the user's environment and behavior to generate virtual content such as visual elements, audio prompts, and even haptic feedback. This content is then integrated back into the user's experience of the real world, *augmenting* that experience with the additional virtual content. *Mobile* augmented reality systems are ones in which the system hardware is untethered and able to be

Authors' addresses: Sarah M. Lehman, Temple University, Philadelphia, Pennsylvania, U.S.A., smlehman@temple.edu; Abrar S. Alrumayh, Temple University, Philadelphia, Pennsylvania, U.S.A., abrar.alrumayh@temple.edu; Kunal Kolhe, Stony Brook University, Stony Brook, New York, U.S.A., kkolhe@cs.stonybrook.edu; Haibin Ling, Stony Brook University, Stony Brook, New York, U.S.A., hling@cs.stonybrook.edu; Chiu C. Tan, Temple University, Philadelphia, Pennsylvania, U.S.A., cctan@temple.edu.

2471-2566/YYYY/8-ARTCCC $15.00
https://doi.org/00.0000/1122334.4556677

carried or worn by the user as they move around their environment. The use of mobile AR systems to solve problems in commercial, industrial, and research settings has exploded in recent years, with the global market for augmented and virtual reality systems estimated to top 18 billion USD by the end of 2020 [70]. Originally the province of games and other "toy" applications [60, 102], mobile AR systems have become key tools in areas such as education [4, 14], manufacturing [30, 51, 97], tourism [16, 36, 84, 85], retail [19, 58], and military settings [3, 53]. Mobile AR systems have become particularly helpful for a range of tasks in the healthcare domain, where they have shown promise when assisting in surgical procedures [9, 59], and as tools for physical or clinical therapy in cases such as phantom limb pain [25] and exposure therapy for phobias [80]. AR systems have also been helpful in managing daily interactions for users with autism [17, 47] and dementia [33, 42, 92], by integrating context-aware prompts for social interaction or task completion into the user's view of the real world.

However, as mobile AR systems become more prevalent, there is a correspondingly increased concern over the privacy risks associated with these apps. The primary benefit of an AR application is the integration of virtual content directly into the user's experience of the real world, based on the user's behavior and the state of the surrounding environment. In order to do this, **mobile AR apps require direct access to live data streams**, such as the camera and microphone feeds. This level of access poses a significant privacy risk, as always-on pervasive sensors such as cameras and microphones can capture unintentionally privacy-sensitive content, both from the user and from bystanders in the user's vicinity. When the sensors are integrated into mobile systems, such as smartphones, tablets, and head-mounted displays (HMDs), the potential invasion of privacy is exacerbated due to the ability to change location and viewing angle freely within the environment. As serious as these privacy risks are, a blanket ban of such systems is infeasible, as this would hamstring vital systems such as those used for monitoring patients in assisted living centers [10].

This privacy threat is further amplified by the **emergence of powerful commercially-available libraries for AR and computer vision, which provide pre-trained machine learning modules out-of-the-box**, effectively eliminating any previous knowledge requirements and streamlining the development process for AR systems. Toolkits such as Google's ARCore [22], Apple's ARKit [21], Microsoft's Mixed Reality Toolkit for Hololens [52], and PTC's Vuforia plugin [88] for the Unity IDE [86], provide AR-specific functionality directly to their targeted platforms. Alternatively, more general-purpose libraries such as OpenCV [55], TensorFlow [81], and Google's MLKit [24] offer a broader range of vision and machine learning operations. *The availability of such tools enables developers to create AR applications without requiring extensive knowledge of computer vision or machine learning concepts*, thereby lowering the bar for malicious developers to collect personal data from their users while introducing limited processing overhead on their mobile platform of choice. Further, since the machine learning and computer vision operations are able to be conducted on-device, a malicious developer need only exfiltrate the *results* of such operations for aggregation and analysis, rather than a set of raw images or video clips; an example of this would be offloading the results of an image labelling operation in the form of an alphanumeric string array, a data packet with a footprint orders of magnitude smaller than a set of images.

Building on prior work [43], the objective of this paper is to understand the privacy risks associated with hidden operations within mobile augmented reality applications, given the capabilities of commercially available AR and vision libraries and the current permissions structures of mobile operating systems. For the purposes of this paper, we focus on mobile AR applications that run on smartphones, as this platform has the greatest number of users, and it can be used to simulate an HMD thanks to products such as the Google Cardboard [87]. As such, we make the following contributions:

(1) **We present the "hidden operations" privacy threat for mobile augmented reality systems**, in which a developer's ability to perform machine learning or computer vision operations behind the scenes of an otherwise honest app, without alerting the user, can result in the loss of users' personal information. We present this threat in the context of three different categories of commercially available AR and vision libraries: integrated AR libraries, function-level libraries, and general purpose libraries. We then examine the similarity of honest AR applications with those engaging in two different types of hidden operations: *complementary* (e.g. building on the advertised functionality of the application), and *orthogonal* (e.g. independent of the advertised functionality of the application).

(2) **We propose a new system for AR application development that OS and app marketplace publishers can employ to mitigate the hidden operations threat**. This system consists of three primary components: a trusted signing service for vision models, a trusted computing environment for executing those models on-device, and an expanded list of runtime permissions to communicate more effectively to end-users what vision operations an AR application is actually performing. We follow this proposal with a set of exploratory experiments to determine the feasibility and utility of our system in assisting with runtime testing efforts, and differentiating between honest and malicious AR applications.

The rest of the paper is organized as follows. Section 2 describes the current related work. Sections 3 and 4 respectively describe our threat model and the approach for inserting malicious logic into an AR application. Sections 5 and 6 describe our proposed system and exploratory experiments, respectively. Section 7 discusses limitations and open problems, and Section 8 concludes.

2 RELATED WORK

In this section, we discuss current work related to the problems of visual privacy for camera-based mobile and wearable systems, the context-sensitive nature of user and bystander privacy preferences, the use of machine learning concepts for testing mobile systems, and the difficulty of testing machine learning modules themselves.

2.1 Visual Privacy in Mobile and Wearable Systems

There has been extensive work recently in exploring visual privacy implications of mobile and wearable camera-enabled systems. Work by Koelle et. al. [38] proposes a gesture-controlled privacy framework; bystanders can provide a gesture to indicate whether the capturing device has permission to include their face in a photograph. Shu et. al. [66] provide a more robust contextually-sensitive framework to allow bystanders to control how they are captured by cameras, but only for individual images, rather than realtime video feeds as mobile AR systems require. However, neither of these solutions protect bystanders who are unaware or unable to provide a gesture, nor do they protect the end user from capturing privacy-sensitive content in the environment.

It is also possible for applications to capture sensitive information from the user's environment that the user would prefer to keep private, such as written or printed material, faces of unaware bystanders, and images taken in a bathroom or bedroom. Systems have been proposed to protect environmental information using markers such as near-infrared labels [44] as well as QR codes and other physical markers [28, 61, 62] to declare to the capturing device what can and cannot be recorded. Schiboni et. al. [63] address the visual privacy problem in a dietary monitoring system by manually modifying the camera viewing angle and field-of-view to limit the system's ability to capture bystanders. Ultimately, the main drawback of both gesture-based and marker-based access control systems is that they only work if the application developer is proactive in protecting user and bystander privacy and chooses to implement these solutions; these systems do nothing to protect against a malicious developer.

When focusing on malicious AR applications specifically, there has been some work on using the device's OS to limit the access that developers have to sensitive visual information. Two works by Jana et. al. [34, 35] create an intermediate layer between the camera and AR application code, but introduce a significant amount of overhead and limit the flexibility of honest app logic. More recent work by Lebeck et. al. [40] proposes the ARYA framework for AR output security, but their system targets visible output issues such as content occlusion, distraction, and physiological reactions such as motion sickness. Their system cannot defend against a malicious developer whose attacks take place completely behind the scenes.

One recent work by Srivastava et. al. [69] seeks to categorize camera-based applications by the operations performed on visual data, and how well that aligns with user expectations of application behavior. The key difference between this is work (known as "CamForensics") and ours is that we specifically focus on malicious developers of mobile AR applications leveraging commercially available vision libraries to conduct additional hidden operations behind the scenes of an otherwise honest application. While the CamForensics system can help identify what high level operations are being performed, it is unable to determine the *context* of such an operation, e.g. whether it is honest or malicious. This is especially important when the same high level operation (such as image labelling) is being used for both honest and malicious purposes within the same application.

2.2 Context-Sensitivity of User, Bystander Privacy Preferences

One of the principal methods of access control for mobile and wearable systems is for application developers to request permission from users to access sensitive features such as the camera and microphone at runtime. It is well-understood, however, that these types of permissions structures are deeply flawed; work by Felt et. al. [26] found that over a third of publicly available Android applications request more permissions than they need, with many issues stemming from developer confusion regarding the nature of each permissions group and what functions they represent. End users are similarly unaware of the scope of requested permissions [27]. Regardless of whether access to a given feature is truly necessary for an application, once it is provided by the user, it is not requested again. Recent work has demonstrated that this all-or-nothing approach to system permissions is unrealistic, particularly for vision-based systems [66, 69]. In a perfect world, applications would execute sensitive operations only in accordance with end-user expectations for a given scenario, a concept known as "contextual integrity" [54]. Wijesekera et. al. [90] showed that maintaining contextual integrity is not a simple problem; factors such as background operations, number of delivered notifications, the reason for the operation, and users' personal privacy concerns all play into whether a user would allow a given operation in the moment.

A survey was conducted by Akter et. al. [5] elicits privacy preferences related to persons with visual impairments using camera systems to supplement their vision, an important use case for mobile augmented reality. The authors surveyed both persons with visual impairments and those without (considered "bystanders"). The bystanders expressed discomfort that users of such systems would be able to learn things like facial expressions and behaviors that had the potential to be "misrepresented" by the system's underlying algorithms. In another work [6], Akter et. al. investigated how persons with visual impairments felt about unintentionally sharing privacy-sensitive images for different usage scenarios, such as different environments (home, office or restaurant) and with different categories of human assistants (friends, family, or crowd-worker). In this paper, the participants expressed vastly differing comfort levels with sharing captured images based on the environment, what the image contained, and who it was being shared with.

Liu et. al. [45] propose an alternate framework for setting system permissions; they used permissions settings gathered from existing smartphone users, and from those settings, compiled "profiles" representing meta-preferences of like-minded users. They then presented new users with an interactive tool, and predicted each user's preferences based on a few targeted questions. Of the permissions set by the assigned profiles, only 5% of settings were changed back by the users. This suggests that alternate methodologies, other than binary approval or rejection of individual permissions, can be useful and effective for end users.

2.3 ML-based Testing and Verification Methods for Mobile Platforms

In addition to researching new techniques for testing and verifying machine learning modules, there have also been recent efforts in applying machine learning concepts and techniques to distinguish between honest and dishonest applications on smartphones. Work by Burguera et. al. [11] monitors calls made by Android applications to the underlying Linux kernel, and applies K-means clustering in order to identify malicious applications. Other works have applied similar clustering approaches other information about the applications under test, such as the requested permissions [82] or sequences of API calls [31, 48]. One study by Afonso et al. [2] uses machine learning to classify an app as malicious or honest based on a dynamically generated feature vector by tracing API calls. However, the authors mention that their approach (at the time of publishing) could be detectable by said app, which may lead to a change in the app's behavior.

Using machine learning to categorize typical patterns and detect subsequent anomalies in runtime behavior is another popular research area. Abah et al. [1] created an anomaly/malware detection system which extracts features from running applications and uses these features to classify app as malicious or safe using a trained K-Nearest Neighbour classifier. Another study by Kurniawan et al. [39] analyzes phone sensor data for anomalies. The data they analyze is power usage, battery temperature, network, CPU, Bluetooth and other hardware resources that is tracked by a mobile device for anomalies. However, this approach uses the Android Malware Genome Project dataset; at the time of writing this paper, no such dataset exists for mobile AR applications. Other approaches utilize Generative Adversarial Networks (GANs) to preserve visual privacy and defend against attacks [93, 95, 96]. However, these approaches are focused on sanitizing datasets collected for offline processing, making them inappropriate for realtime augmented reality operations.

There has also been work to develop "trusted" neural networks which adhere to additional constraints beyond their initial functions. One approach proposed by Halliwell et. al. [32] enforces saliency constraints on convolutional neural networks; in doing so, developers can trust that the internal layers of the network are leveraging the correct pixels of an input image in order to make classification decisions. Ghosh et. al. [29] propose a parameterized approach to determine the "cost" of trusting a neural network's output for a given set of conditions. Their work leverages steering directions for autonomous vehicles, where a dual-headed network calculates a proposed steering angle for a set of traditional inputs (e.g. camera frames), as well as a "danger" score for violations of a set of safety constraints (e.g. minimum and maximum allowable angles). The developer is then free to adjust the weights between these results in order to maximize accuracy while keeping the danger score within acceptable limits. Approaches like these require the participation of the network developer; they can do nothing if the developer fails to incorporate these additional constraints into their network, either through unintentional omission or malicious intent. Additionally, such measures only increase confidence that the *function* of the neural network itself can be trusted. They cannot control or verify that the *context* under which the function is being executed is appropriate.

2.4 Testing Deep Learning Systems

Deep learning (DL) are one subset of machine learning systems that have gained in particular popularity thanks to their unique applications in domains such as autonomous vehicles [7]. However, these types of systems also carry great risk, as errors in controlling logic can lead both to personal injury or loss of property. As such, interest in *testing and verification of DL systems* has increased in recent years. (This contrasts with *machine learning-based testing*, in which machine learning concepts are applied to testing of traditional systems, discussed above in Section 2.3.) The challenge is that testing practices of traditional software systems do not translate directly to DL systems. Where traditional software systems are "logic-driven" (e.g. behavior is controlled by logic flows manually encoded by their programmers), DL systems are "data-driven", where behavior is governed by the architecture of the underlying neural network, the computations and activation functions being performed at each node, the weights between nodes, and the quality of the training data [100]. There are few "constituent parts" upon which to conduct unit testing, meaning the system must be tested as a whole [98]. However, simply feeding inputs to the system and evaluating the resulting output is an ineffective strategy; the search space of all possible inputs to such systems (as well as enumerating their respective expected outputs) is so large as to make brute-force testing prohibitively time- and resource-intensive.

Developing new strategies for testing deep learning systems, as well as metrics for measuring success, is a popular research area. Much effort has gone into developing metrics to represent logical coverage within a neural network. Proposed systems such as DeepXplore [56] and DeepTest [83] leverage a concept known as "neuron coverage", where inputs are scored according to how many neurons are activated in their processing. This concept was inspired by traditional software testing metrics such as "code coverage", in which testers seek to execute every line of code in a code base by the end of the test suite. Both approaches manipulate input images in order to maximize neuron coverage, but differ in how the results are evaluated. DeepXplore evaluates the results by cross-referencing with other DL systems, while DeepTest applies metamorphic relations to detect if the output is within acceptable limits. DeepGauge [49] builds on this idea, proposing the concepts of k-multisection neuron coverage (how well a given input covers the activation range of a given neuron) and top-k neuron coverage (a measurement of the most commonly activated neurons in a given layer of the network). These approaches, however, only consider how a system responds to a given input in isolation and not how the responses *differ* between inputs; to address this, Kim et al [37] propose a metric called "surprise adequacy" which measures how novel a given input is relative, not only to the network's training data, but to the set of inputs overall. Retraining the network with "surprising" inputs is then demonstrated to increase network accuracy.

There has also been interest in developing property-based verification methods for DL systems, beyond simply verifying the accuracy of outputs for a given set of inputs. Pei et al [57] treat the internal network of the system as a black box, validating a given input/output pair against a high-level safety property rather than trying to ascertain the nature of the network directly. Wang et. al. [89] apply symbolic linear relaxation and constraint refinement to validate the internal activation functions of a network in order to enforce safety properties. The developers of DeepMutation [50] utilize principles of mutation testing, applying mutations directly to the training data set, the network training program, and the resultant model in order to identify errant behavior. Test inputs are fed into both the original model and the set of mutated models; errors are detected when the output from a given mutated model for a given input differs from that of the original model.

While all of these approaches can provide some assurances into the ability of a given deep learning system to produce reliable output or to adhere to a given safety property, they all fail to address one particular problem. They do not ensure *contextual integrity*, that is, that the system

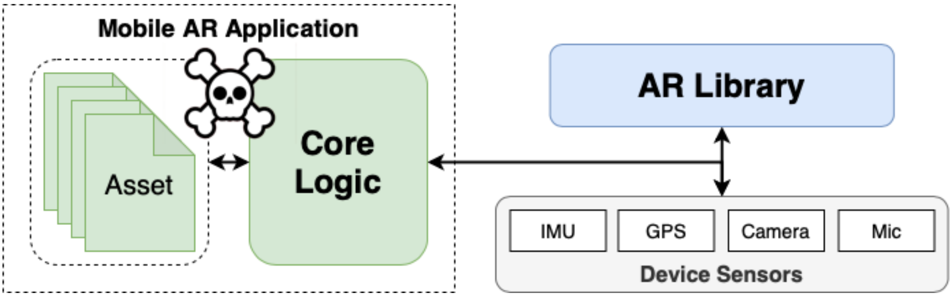


Fig. 1. System architecture for a mobile AR application leveraging a third-party AR library. The application is controlled by its core logic and any asset files needed to interface with the selected AR library. The nature of the asset files depends on the type of AR library being used, and could be anything from raw images, to databases of 3D object feature points, to pre-trained neural networks. The developer has complete control over the core logic and the asset files; once in place, they can be easily exploited to conduct additional hidden operations behind the scenes without the user's knowledge (discussed further in Section 4.2).

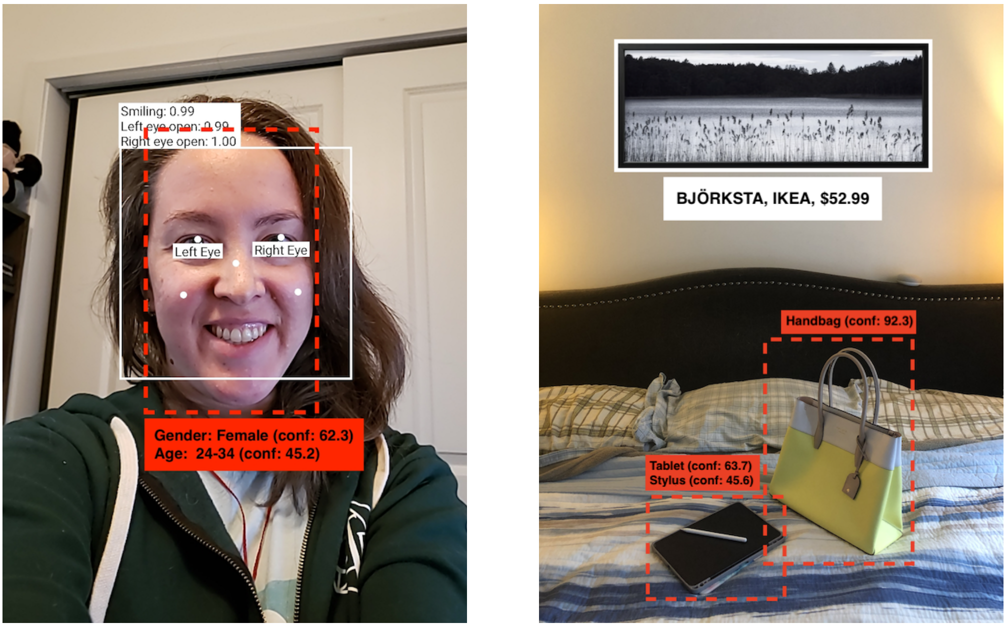
under test adheres to the user's expectation of appropriate behavior in a given scenario (discussed above in Section 2.2).

3 THREAT MODEL

While it is possible for an incompetent or unaware developer to violate user privacy unintentionally, for the purposes of this paper, we consider the adversary to be a *third party app developer who intentionally creates and distributes a malicious mobile AR application*. In order to distribute their application, the adversary must be able to publish the malicious AR app to the approved application marketplace for their platform of choice; this requires them to adhere to all standard conventions for system architecture, data, and asset storage, permissions, and security as dictated by the target operating system. In doing so, this demonstrates that, from a code inspection standpoint, the malicious application appears indistinguishable from an honest one.

Figure 1 demonstrates the system architecture for a mobile AR application. The AR application's core logic drives the primary functionality, such as managing the user interface and handling system events. The core logic interfaces with a third-party AR library, which encapsulates the computer vision and machine learning logic for the system to use. This interaction is managed by a set of asset files, which represent the targets that the AR library is trying to recognize. Asset files are unique to each AR library, and can take the form of raw images, compiled databases of feature points, neural networks, and more. The AR library connects to the device sensors (e.g. camera, microphone, etc.), using the provided asset files to trigger recognition events. The application core is then free to respond to these events in whatever way is appropriate, such as displaying a label or placing a 3D model into the user's view of the environment.

The adversary's objective is to use the AR app to stealthily collect information about the app user and their environment that they would otherwise be unable to obtain from a non-AR smartphone app [67, 68]. In other words, we do not consider scenarios such as side-channel attacks where the AR app attempts to collect GPS location, since this kind of information can already be obtained via other means. Rather, we focus on data which is obtainable through execution of machine learning logic by an AR application during runtime, that is, from the live camera feed. The adversary will attempt to stealthily collect this information through the use of **hidden operations** within in the AR app. We define "hidden operations" to mean any machine learning functionality that is outside the scope of what is communicated to and expected by the end user. Once the data is obtained, we



(a) Complementary (estimating age and gender)

(b) Orthogonal (inferring relative wealth)

Fig. 2. Example "hidden operation" attacks for (a) *complementary* [e.g. building on the advertised functionality] and (b) *orthogonal* [e.g. independent of advertised functionality] operations

assume that the adversary will be able to exfiltrate it without being detected by the user, using techniques such as piggybacking the information on top of routine software updates.

3.1 Threat Overview

The ability for developers of mobile augmented reality applications to perform "hidden operations", that is, unadvertised machine learning or computer vision operations on live camera frames, constitutes a serious threat to end-user privacy. Deng et. al. describe seven key threats to user privacy, known as the LINDDUN model [20]: Linkability (the ability to determine whether two items of interest are related in the system), Identifiability (the ability to positively identify an item of interest within the system), Non-repudiation (the ability to conclusively tie an actor back to an action taken within the system), Detectability (the ability to determine whether an item of interest exists within the system), information Disclosure (the ability to expose personal information that should not be accessible), content Unawareness (the inability of an end-user to fully know the extent of data collected or inferred by the system), and policy and consent Non-compliance (the ability of the system to go against advertised privacy policies and end-user expectations).

By conducting a hidden operation behind the scenes of the advertised AR application functionality, the developer engages in three of the LINDDUN privacy threats in particular: information disclosure, content unawareness, and policy and consent non-compliance. First, the system exhibits **information disclosure** by leveraging a legitimate data stream to potentially learn personal data that it should not have access to (such as the user's age, gender, or socio-economic status). The system also results in **content unawareness** by keeping secret the nature and extent of data gathered during the operation. The fact that the user is unaware of the hidden operation is fundamental to the operation's success. Finally, the system exhibits **policy and consent non-compliance** by exploiting the permissions granted for the advertised AR functionality to execute the additional hidden operations behind the scenes.

3.2 Assumptions and Example Scenarios

There are a number of ways in which the adversary can implement their hidden operations. In this paper, we only consider the adversary who utilizes *existing vision and machine learning libraries* to implement their hidden operations. This is because commercially available libraries provide the most convenient method for developing AR apps, since they require a much lower level of skill and domain knowledge compared to implementing such logic from scratch. Additionally, we assume that the adversary is *completing the majority of their machine learning logic locally on-device*; this is because augmented reality applications have a strict threshold of tolerable computational latency. Offloading the entirety of vision or machine learning operations to the cloud would incur too much latency to be feasible for realtime responsiveness as required for AR system output [46, 101]. Further, we assume that the hidden operations are also performed locally, as offloading frames for processing *en masse* risks getting caught by network monitors for transmitting large amounts of image or video data, without any guarantee that the files contain anything of interest.

The scope of the hidden operations privacy threat can be demonstrated through the following sample scenarios:

(1) **Complementary Hidden Operation**: Figure 2a demonstrates a complementary hidden operation, in which the additional logic being executed leverages and builds upon the existing, advertised functionality of the base application. Here, the adversary distributes an application such as Snapchat [78], which identifies and displays an overlay on top of faces. However, behind the scenes, they utilize a custom TensorFlow model to estimate the age and gender of observed faces without communicating so to the user. This is considered a “hidden operation” because age and gender estimation ostensibly have nothing to do with the advertised functionality of the application.

(2) **Orthogonal Hidden Operation**: Figure 2b demonstrates an orthogonal hidden operation, in which the additional logic being executed is completely independent of the application’s existing advertised functionality. Here, the adversary offers a retail application which detects vertical or horizontal planes to display virtual models of furniture, allowing the user to preview how the items would look in their home. However, under the covers, the adversary also leverages Google ML Kit’s coarse-grained image labelling functionality to determine when the camera view also contains interesting items such as electronics, handbags, or jewelry. Once detected, these frames can be offloaded for additional processing, in order to infer the user’s relative economic status.

In the following sections, we discuss a selection of possible implementations for hidden operations in mobile AR, and propose a design for improved testing and permissions of mobile AR apps.

4 HIDDEN OPERATIONS IN MOBILE AR APPLICATIONS

In this section, we present an overview of the process of building a mobile augmented reality application, as well as the steps involved in hiding additional logic behind the scenes using a selection of commercially available libraries. We also present a series of comparisons between honest and dishonest AR applications to demonstrate their similarity and the subsequent difficulty in distinguishing between them.

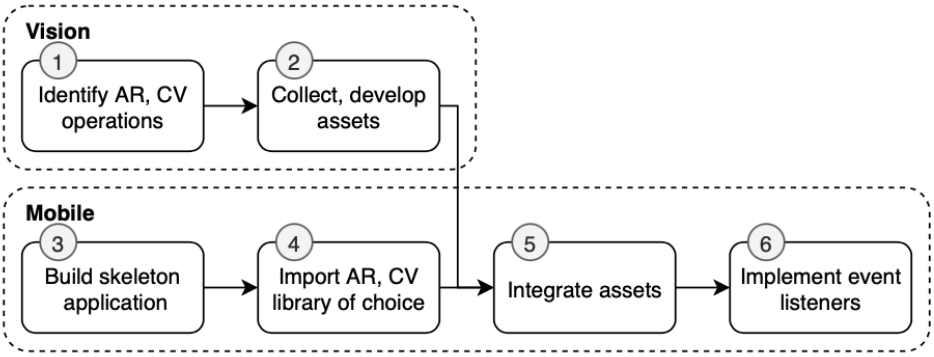

Vision
1
Identify AR, CV operations
2
Collect, develop assets
Mobile
3
Build skeleton application
4
Import AR, CV library of choice
5
Integrate assets
6
Implement event listeners

Fig. 3. High-level pipeline for building a mobile AR application. When an adversary wishes to incorporate additional dishonest logic, they must develop new vision assets (Steps 1 and 2), and integrate those assets and associated response logic into the existing app (Steps 5 and 6). Figures 4, 5, and 6 go into more detail of how these steps are done, where asset development and event response logic are marked (A) and (B) respectively.

4.1 Building Mobile AR Apps

Building a mobile AR application consists of two complementary pipelines, as described in Figure 3. The first pipeline reflects the development of computer vision functionality, while the second reflects the mobile application development. The first two steps of each pipeline can be executed in tandem with each other.

The first step in the vision pipeline is to identify the AR or CV operations that the application should support, and which AR or CV library will be utilized to implement them ①. Common vision operations include ground plane detection, facial recognition, object classification, and others. Once the desired vision operation and corresponding library been identified, the second step in the vision pipeline is to collect or develop the assets necessary to accomplish the selected operation ②. "Assets" are resource files utilized by the system during runtime to complete the desired operation, and can vary depending on the type of operation being executed as well as the library being used. For example, 2D image recognition with ARCore requires copies of the images to be recognized, while object classification with TensorFlow requires datasets of images representing the various classes to be recognized, which are then used to train a neural network.

The first step of the mobile pipeline consists of building the initial application skeleton ③, including establishing the system architecture, setting up scenes, connecting to device sensors, and other tasks. The second step involves importing the AR or CV library of choice ④, as well as supplementary tasks pertinent to the chosen library, such as registering a developer key. Once the vision assets have been assembled and the preliminary application developed, the developer can then integrate the assets into the application ⑤, typically by registering them with the library at application start-up. Finally, the developer can implement listeners to respond to results of the vision operations as they become available ⑥. Developers are free to respond to these results in whatever way makes sense for their application. Examples of potential responses include placing a graphical overlay in the user's view to highlight an object, placing a label to give more information, playing a sound, delivering haptic feedback, or nothing at all.

	APPROACH	LIMITATIONS
Integrated AR Libraries	Leverage common functionality provided directly by OS or platform publisher; configurable with custom asset files	Requires pre-existing knowledge of targets in order to select asset files; no ability to modify library functionality
Function-level Vision Libraries	Pre-packaged implementations for common functions (e.g. image classification) with limited customizability	Requires manual connection to camera and processing of frames; no support for dynamic offloading of operations
General Purpose Vision Libraries	Platform-agnostic support for advanced vision and machine learning operations with high degree of customizability and control	Requires high level of subject matter expertise, and manual collection of data sets for training and testing

Table 1. High-level summary of libraries that may be used when hiding malicious operations.

4.2 Hiding Additional Operations

We assume that the adversary is utilizing a commercially available vision library in order to build their AR application and implement their hidden operations. When implementing one of these operations, the adversary has three broad categories of libraries to pick from (as summarized in Table 1): integrated AR libraries; function-level computer vision and machine learning libraries; or general purpose vision and machine learning libraries. The developer may also combine any of these libraries together into a single app, both within and across categories. For example, an application which uses Vuforia to identify beverage labels can also leverage ARCore to detect and track face meshes, or an application which uses ARCore to place virtual superheroes on a tabletop could also use TensorFlow to identify other objects in the room and generate customized speech bubbles for the models. In doing so, the adversary can leverage the strengths of one type of library to compensate for the limitations of another. Next, we go into greater detail explaining the differences between these various vision libraries.

4.2.1 Integrated AR Libraries: Integrated AR libraries are fully encapsulated, local-processing libraries developed specifically by a given OS or platform provider in order to supply AR functionality. Examples of integrated AR libraries include Google's ARCore [22], Apple's ARKit [21], Microsoft's Mixed Reality Toolkit [52], and PTC's Vuforia [88]. Figure 4 shows the program loop of an application using an integrated AR library, such as ARCore. Being directly integrated with the target platform or IDE allows these libraries to abstract away many of the mechanical operations for the adversary; the library will automatically connect to the camera, perform target recognition and tracking, and invoke event listeners. The adversary is responsible only for supplying assets Ⓐ and registering event listeners to execute when a target's asset is recognized in the camera feed Ⓑ. The nature of the assets to be supplied depends on the library being used. ARCore supports recognition of elements such as the environmental ground plane (no assets required), and raw 2D images (such as collections of logos). Meanwhile, Vuforia recognizes not only the ground plane and collections of 2D images, but also 3D models of real-world objects.

Benefits: The primary benefit of integrated AR libraries is the speed and efficiency with which their functionality can be integrated into an application, due to the low burden of knowledge required from the adversary to utilize them. Additionally, because the recognition logic for these libraries depends only on these asset files, it is a relatively simple matter to update trackable targets

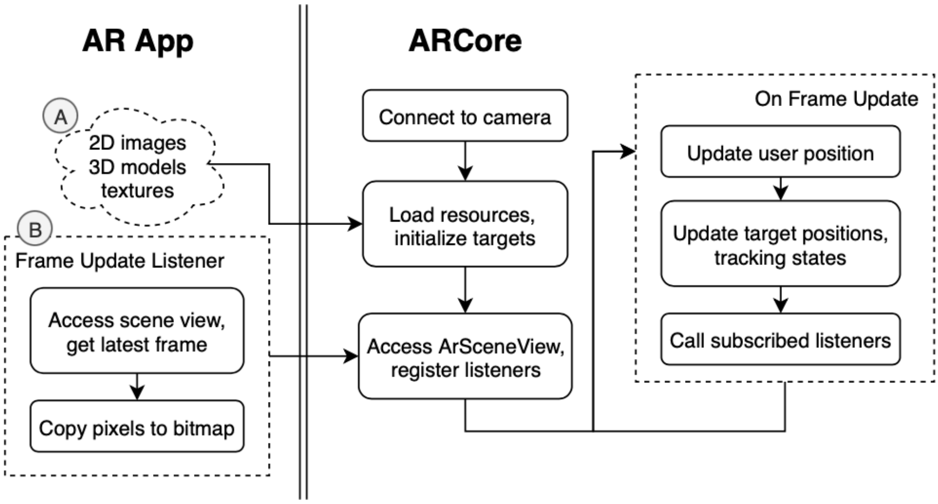

AR App
ARCore
A
2D images
3D models
textures
B
Frame Update Listener
Access scene view, get latest frame
Copy pixels to bitmap
Connect to camera
Load resources, initialize targets
Access ArSceneView, register listeners
On Frame Update
Update user position
Update target positions, tracking states
Call subscribed listeners

Fig. 4. Program loop for an integrated AR library (e.g. ARCore). While each integrated library has its own unique architecture, they all share similar high-level concepts.

on the fly by downloading new assets during runtime. The adversary can easily change their application logic by downloading new images (if using ARCore) or a new target database (if using Vuforia) from a back-end server, and resetting the library's collection of active assets.

Limitations: There are some limitations to using integrated AR libraries when implementing an AR application. First, the adversary is limited to using only that functionality that the library publisher provides at the time. There is no way for them to modify or expand that logic if it doesn't meet their needs. Second, the reliance on explicit asset files requires the adversary to know *exactly* what they will be recognizing during runtime; there is no support for class-based recognition.

4.2.2 Function-level Vision Libraries: In contrast to integrated libraries, function-level vision libraries are resources offered in a modular fashion around specific AR or CV operations with moderate levels of customizability. These libraries can either be executed locally or remotely, depending on the publisher's system architecture. Google's ML Kit [24] and Cloud ML APIs [18] are examples of local and remote function-level vision services respectively. Figure 5 shows the program loop of an application leveraging a function-level vision library such as ML Kit. Here, instead of providing raw asset files for the library to recognize, the adversary need only import the specific sub-library associated with the functionality they wish to utilize Ⓐ. Within the application's core logic, the adversary is required to provide considerably more system-level logic to perform such tasks as connecting to the device camera, processing frames, and passing them to the library to be recognized Ⓑ.

Benefits: The primary benefit of function-level vision libraries is in their considerably broader range of supported operations compared to integrated AR libraries. While integrated AR libraries are limited to operations such as ground plane detection and recognition of pre-loaded 2D and 3D targets, libraries such as Google's ML Kit offer more robust operations such as text recognition, face detection, barcode scanning, pose estimation, image labeling, object classification, and target tracking. Further, because the libraries are subdivided by functionality, only the portion relevant to the desired function need to be added to the target application. Because of this, the same library can be used to perform different operations; for example, the default image labeling library from

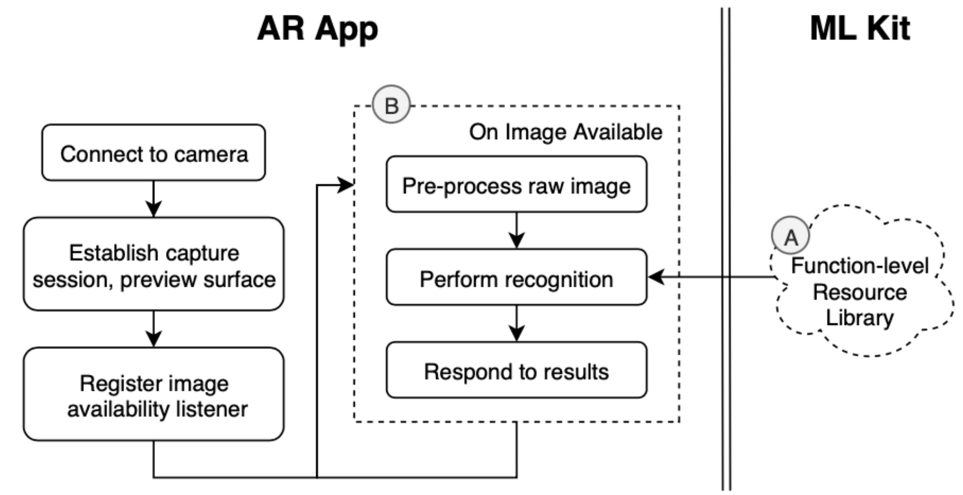

AR App
ML Kit
B
On Image Available
Connect to camera
Pre-process raw image
Establish capture session, preview surface
Perform recognition
Respond to results
Register image availability listener
A
Function-level Resource Library

Fig. 5. Program loop for function-level vision library (e.g. Google ML Kit). While each function-level library has its own unique architecture, they all share similar high-level concepts.

Google's ML Kit recognizes more than 400 class-level labels [23], allowing the adversary to use the same logic whether they want to recognize animals such as cats, birds, or dogs, or "luxury" items such as sunglasses, jewelry, or computers.

Limitations: Similar to integrated libraries, the functionality of function-level libraries is limited to what the publisher provides. While the range of functionality offered by libraries such as Google MLKit is more robust than libraries such as ARCore, and in some cases *does* provide some level of customization for a given operation, the adversary is still limited in what they can do. Further, at the time this paper was written, no commercially available function-level libraries offer support for dynamic offloading of operations; depending on the library, all operations are either fully local or fully remote. Therefore, this category of libraries is a poor choice for an adversary who desires the flexibility of a hybrid processing solution.

4.2.3 General Purpose Vision Libraries: General purpose vision libraries such as Tensorflow [81], OpenCV [55], and SciKit-Learn [64] are fully open-ended third-party libraries which support more advanced levels of vision and machine learning operations. Figure 6 shows the program loop for an application utilizing one of these general purpose libraries (such as Tensorflow). The adversary must first collect a sufficient body of data samples for their chosen operation. They then select a base algorithm, then trains, tests, and validates the model before exporting it into a mobile-compatible format Ⓐ. In this way, general purpose libraries build on the structure of function-level libraries by allowing the adversary to fully customize the recognition logic in whatever way they wish. In the application code, they must connect to the device camera, extract frames from the feed, perform pre-processing, and pass them to the model to be processed Ⓑ.

Benefits: The primary benefits of using general purpose vision libraries center on the adversary's ability to fully customize their model according to their own system's needs. These libraries can be used when publishers of the other two library categories do not offer an adversary's desired functionality out-of-the-box. Examples of this include the publisher offering a given function that is trained for an unhelpful dataset, offering a given function remotely when the adversary needs to run it locally, or not providing the functionality at all. Further, if they discover a scenario in which

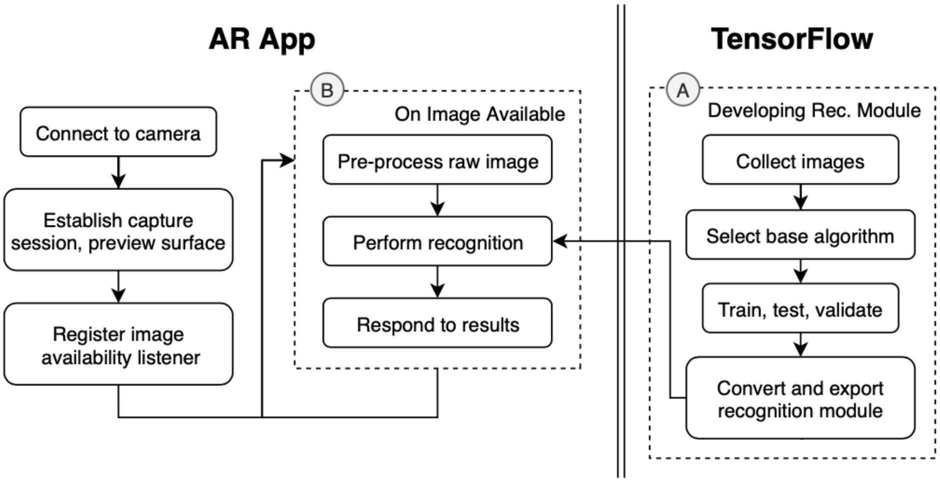


Fig. 6. Program loop for general purpose vision library (e.g. Tensorflow). While each general purpose library has its own unique architecture, they all share similar high-level concepts.

their system is performing poorly, they can retrain and redeploy their vision model to improve performance, while another developer using an integrated or function-level library would have to make do with whatever functionality was commercially available.

Limitations: The primary limitation of general purpose vision libraries is that they are much more difficult to use than integrated AR libraries or function-level vision libraries, requiring a much higher level of familiarity with computer vision concepts in order to develop a working model. Therefore, this type of library is *not* recommended for an adversary who is not comfortable with more hands-on computer vision or machine learning tasks, or one who is unable to collect sufficient data samples for testing and training. This is because the work of training a custom vision module to a high level of accuracy can be quite demanding in terms of time, storage and processing requirements, especially with environmental variations such as low light or a partially blocked line of sight. This makes the process of updating a model on the fly quite impractical when running locally. Finally, general purpose libraries are typically quite monolithic in structure compared to the other two categories, which can contribute further to storage and resource drain on-device.

4.3 Comparing Honest and Dishonest Applications

There are currently no publicly available, open source examples of known malicious AR applications; therefore, we developed three prototype applications to investigate the extent to which honest and dishonest MAR applications differ from each other. The applications were fully implemented in Java for the Android operating system as proofs-of-concept for increasing levels of maliciousness. The first application (A1) represents the "honest" application, in which the only operations being executed are in keeping with advertised functionality, and are thoroughly communicated to the end users. For the purposes of this paper, we designed A1 to recognize faces and place an overlay on the camera feed, outlining the primary landmarks of the recognized face (e.g. eyes, mouth, etc.). The second application (A2) builds on the first, supplementing the honest logic with a *complementary hidden* operation, that is, one that builds upon the existing honest logic without informing users of its existence or purpose. A2 calculates the age and gender of the detected faces, but provides no indicator to the user of the operation. Finally, the third application (A3) supplements the honest

PERMISSION	LEVEL	A1	A2	A3	MATCH
Internet	Normal			X	100%
Access Network State	Normal			X	96.8%
Camera	**Dangerous**	**X**	**X**	**X**	**95.1%**
Write External Storage	**Dangerous**	**X**	**X**	**X**	**91.9%**
Read External Storage	Dangerous		X		74.2%
Read Phone State	Dangerous		X		29%

Table 2. OS permissions requested by adversarial application prototype versions: (A1) honest, (A2) complementary hidden op., and (A3) orthogonal hidden op. with corresponding match percentage among publicly available AR applications on Google Play Store (n = 62)

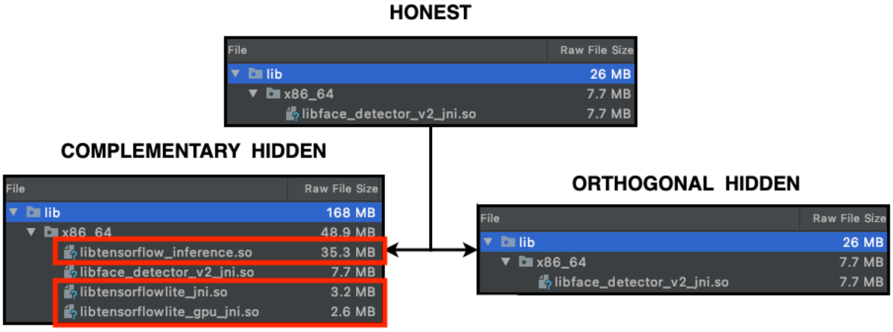
logic with an *orthogonal hidden* operation, that is, one that is completely independent of the existing honest logic, but is similarly undisclosed to the end users. A3 performs text recognition on frames taken from the live feed, an operation which is performed regardless of the results returned from the primary logic.

4.3.1 Permissions. All three of the applications described above were developed with the same set of explicitly requested permissions, e.g. access to the camera and write permissions to external storage. During compile time, the various resource libraries incorporated into the different code bases add their own requested permissions to the application manifest. The final lists of requested permissions for each application version are reflected in Table 2, where the boldface lines are developer-requested and the rest are library-requested.

Because it is important for prototype applications to reflect real-world conditions, we conducted a survey of AR applications available on the Google Play Store and compared their lists of requested permissions with the ones requested by our honest and dishonest prototypes. Since the Google Play store does not support filtering or sorting of applications, we had to use the pre-filtered lists provided by Google under the “Augmented Reality” category - “AR Apps”, [71], “AR Games”, [72], and “Best of AR” [73]. To filter the raw corpus of applications down to a manageable size, we went through each list and filtered out duplicates. After the first pass, we were left with 713 unique applications. For our second pass, we considered only those applications with a million or more downloads, yielding a final count of 62 unique applications. For each of these, we downloaded the APK, analyzed it using Android Studio’s APK inspector, and extracted the lists of requested permissions from the applications’ manifest files, as summarized in Table 2. (It should be noted that three of the applications did not request the camera permission - Google Photos, [74], Google Spotlight Stories, [75], and Mondly [77]. We believe that these applications were either misclassified by the app store, or the “AR” features are performed on media already present on the user’s phone and not explicitly retrieved from the camera.)

Of the permissions requested by our various applications, all but one are also requested by more than 90% of commercially available AR applications on the Google Play Store. (“Write External Storage” is a super-permission of “Read External Storage”; granting of write permissions also grants read permissions by default.) Based on these results, we conclude that the sets of permissions requested by our applications are reasonable, and reflect the types of permissions typically requested by commercially available apps.

4.3.2 Manual Inspection. In addition to requested permissions, we also performed manual inspection of the different applications to attempt to identify differences between versions. We considered the following: package size, asset and library artifacts, and impacts to the runtime UI.



HONEST

File	Raw File Size
▼ lib	26 MB
▼ x86_64	7.7 MB
libface_detector_v2_jni.so	7.7 MB

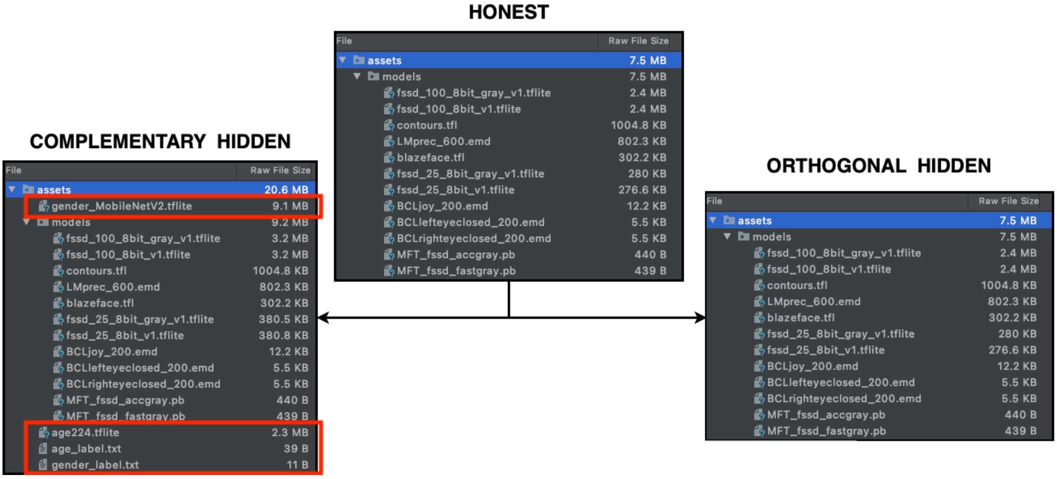
COMPLEMENTARY HIDDEN

File	Raw File Size
▼ lib	168 MB
▼ x86_64	48.9 MB
libtensorflow_inference.so	35.3 MB
libface_detector_v2_jni.so	7.7 MB
libtensorflowlite_jni.so	3.2 MB
libtensorflowlite_gpu_jni.so	2.6 MB

ORTHOGONAL HIDDEN

File	Raw File Size
▼ lib	26 MB
▼ x86_64	7.7 MB
libface_detector_v2_jni.so	7.7 MB

Fig. 7. Lib artifacts extracted with Android Studio APK Inspector for adversarial applications. Additional artifacts introduced by hidden operations are outlined in red.



HONEST

File	Raw File Size
▼ assets	7.5 MB
▼ models	7.5 MB
fssd_100_8bit_gray_v1.tflite	2.4 MB
fssd_100_8bit_v1.tflite	2.4 MB
contours.tfl	1004.8 KB
LMprec_600.emd	802.3 KB
blazeface.tfl	302.2 KB
fssd_25_8bit_gray_v1.tflite	280 KB
fssd_25_8bit_v1.tflite	276.6 KB
BCLjoy_200.emd	12.2 KB
BCLlefteyeclosed_200.emd	5.5 KB
BCLrighteyeclosed_200.emd	5.5 KB
MFT_fssd_accgray.pb	440 B
MFT_fssd_fastgray.pb	439 B

COMPLEMENTARY HIDDEN

File	Raw File Size
▼ assets	20.6 MB
gender_MobileNetV2.tflite	9.1 MB
▼ models	9.2 MB
fssd_100_8bit_gray_v1.tflite	3.2 MB
fssd_100_8bit_v1.tflite	3.2 MB
contours.tfl	1004.8 KB
LMprec_600.emd	802.3 KB
blazeface.tfl	302.2 KB
fssd_25_8bit_gray_v1.tflite	380.5 KB
fssd_25_8bit_v1.tflite	380.8 KB
BCLjoy_200.emd	12.2 KB
BCLlefteyeclosed_200.emd	5.5 KB
BCLrighteyeclosed_200.emd	5.5 KB
MFT_fssd_accgray.pb	440 B
MFT_fssd_fastgray.pb	439 B
age224.tflite	2.3 MB
age_label.txt	39 B
gender_label.txt	11 B

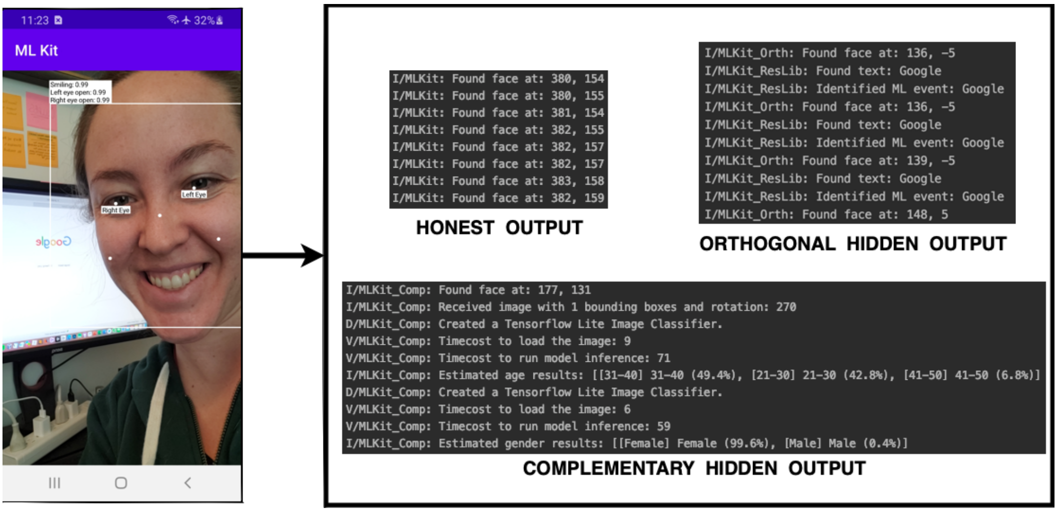
ORTHOGONAL HIDDEN

File	Raw File Size
▼ assets	7.5 MB
▼ models	7.5 MB
fssd_100_8bit_gray_v1.tflite	2.4 MB
fssd_100_8bit_v1.tflite	2.4 MB
contours.tfl	1004.8 KB
LMprec_600.emd	802.3 KB
blazeface.tfl	302.2 KB
fssd_25_8bit_gray_v1.tflite	280 KB
fssd_25_8bit_v1.tflite	276.6 KB
BCLjoy_200.emd	12.2 KB
BCLlefteyeclosed_200.emd	5.5 KB
BCLrighteyeclosed_200.emd	5.5 KB
MFT_fssd_accgray.pb	440 B
MFT_fssd_fastgray.pb	439 B

Fig. 8. Asset artifacts extracted with Android Studio APK Inspector for adversarial applications. Additional artifacts introduced by hidden operations are outlined in red.

Package Size: The size of an application's installation file can be an indicator of how much local processing the application does. In Android, this installation file is known as the APK (or Android PacKage). When considering the applications listed by the Google Play Store under the "Augmented Reality" category (e.g. "AR Apps", [71], "AR Games", [72], and "Best of AR" [73]), the average APK size is 50.1 MB with MIN = 1.2MB, MAX = 539MB, and STD_DEV = 32.8MB (n=713). This represents a wide range of "normal" package sizes, with the majority coming in under 100MB. The corresponding package sizes for our adversarial applications are 38.5MB for (A1), 201.4MB for (A2), and 38.9MB for (A3), with A1 and A3 coming in well below the average. Even A2, with a package size considerably above average, has a package size well below the largest commercially available app (size 539MB). Based on this, we conclude that our package sizes are reasonable, with no observable deviation or abnormality compared to package sizes of commercially available MAR applications.

Asset and Library Artifacts: Android Studio provides a packet inspector to analyze APK files. When we use this to open the compiled APKs, we see that the lib and assets directories make up



```
I/MLKit: Found face at: 380, 154
I/MLKit: Found face at: 380, 155
I/MLKit: Found face at: 381, 154
I/MLKit: Found face at: 382, 155
I/MLKit: Found face at: 382, 157
I/MLKit: Found face at: 382, 157
I/MLKit: Found face at: 383, 158
I/MLKit: Found face at: 382, 159
```

HONEST OUTPUT

```
I/MLKit_Orth: Found face at: 136, -5
I/MLKit_ResLib: Found text: Google
I/MLKit_ResLib: Identified ML event: Google
I/MLKit_Orth: Found face at: 136, -5
I/MLKit_ResLib: Found text: Google
I/MLKit_ResLib: Identified ML event: Google
I/MLKit_Orth: Found face at: 139, -5
I/MLKit_ResLib: Found text: Google
I/MLKit_ResLib: Identified ML event: Google
I/MLKit_Orth: Found face at: 148, 5
```

ORTHOGONAL HIDDEN OUTPUT

```
I/MLKit_Comp: Found face at: 177, 131
I/MLKit_Comp: Received image with 1 bounding boxes and rotation: 270
D/MLKit_Comp: Created a Tensorflow Lite Image Classifier.
V/MLKit_Comp: Timecost to load the image: 9
V/MLKit_Comp: Timecost to run model inference: 71
I/MLKit_Comp: Estimated age results: [[31-40] 31-40 (49.4%), [21-30] 21-30 (42.8%), [41-50] 41-50 (6.8%)]
D/MLKit_Comp: Created a Tensorflow Lite Image Classifier.
V/MLKit_Comp: Timecost to load the image: 6
V/MLKit_Comp: Timecost to run model inference: 59
I/MLKit_Comp: Estimated gender results: [[Female] Female (99.6%), [Male] Male (0.4%)]
```

COMPLEMENTARY HIDDEN OUTPUT

Fig. 9. Runtime UI screenshot of adversarial prototypes with corresponding log output

the bulk of our storage consumption. In Android, the `lib` folder contains executable logic from an application's imported resource libraries, and is populated by the library publishers. Conversely, the `asset` folder is used as raw storage for resource files for the system to use during runtime, and can be populated by both the developer and the library publishers.

Figures 7 and 8 describe the contents for each of these folders respectively, where the additional artifacts introduced by the hidden operations are outlined in red. We can observe from these figures that the only the *complementary* hidden operation introduced new artifacts into either the `assets` or `libs` directory. Even considering these artifacts, they are not particularly informative; the shared operations files in the `lib` directory are general-purpose only, and the `asset` files are completely under the control of the adversary, who can name them whatever they want. For this scenario, there is nothing stopping the adversary from choosing completely misleading names for their model and label file assets.

Runtime UI: A runtime screenshot and samples of the corresponding output for each application are shown in Figure 9. For each of the applications, the same runtime UI is displayed to the user - that of the dynamic overlay displayed on top of a detected face. There are no additional indications made to the user that other operations are being performed. However, if we examine the applications' runtime logs, we can see output from the various dishonest operations that differ from the honest application. The *complementary* dishonest application outputs additional information about the estimated age and gender of the detected face, while the *orthogonal* dishonest application performs text recognition. However, because current commercial vision toolkits and mobile operating systems place no requirements on developers to make any sort of indication to users when vision or machine learning operations are being performed, the dishonest applications are able to perform these operations without making any changes to the runtime UI.

4.4 Results

Based on the similarity comparison above, we make the following conclusions. First, we observe that **the increasing availability and maturity of commercially available vision libraries makes it quite simple to conduct hidden machine learning operations without informing the end user**. Commercially available vision libraries give an adversary a host of options to incorporate into their application, catered to their needs and pre-existing subject matter expertise. These powerful operations can be used for a variety of different purposes, none of which are required to be consistent with the application's advertised functionality, or to be communicated to the end user. As demonstrated above, in the case of low-overhead functionality such as text recognition, the introduction of new libraries may not even introduce new artifacts into the APK's `assets` or `libs` directories. Additionally, because these operations are being conducted in the background, there are no impacts to the runtime UI of the application to alert the end user to their presence. This has the effect of lowering the bar to create AR applications in the first place, while simultaneously making it easier to violate the privacy of end users by exploiting these same libraries.

Second, we see that the **OS-level permissions structures currently provided by mobile platforms are insufficient to protect against hidden machine learning operations**. All three of the adversarial applications described above, honest and dishonest alike, achieved their functionality using the same list of permissions, e.g. access to the camera and write permissions to storage, which were requested by more than 90% of AR applications from the Google Play store. Access to a given permission is requested on first execution of an application after install; once granted, it is never requested again, and the application has full access to any and all features related to that permission. Thus, an application will request its permissions once, and then have no subsequent accountability on when or why it is accessing that functionality. Further, contemporary mobile operating systems make no requirements on developers either to request permission to perform machine learning operations, or to make any visual or auditory indications to users when those operations are being performed. Based on this, in current mobile operating systems, there is no way to tell an honest application from a dishonest one based on the permissions they request.

Based on these observations, in the following sections, we propose and perform exploratory experiments on a system which seeks to mitigate the hidden operations threat. This system addresses the problem first by preventing developers from executing unvetted vision logic, and second by increasing user awareness of application behavior during runtime.

5 SYSTEM DESIGN

One of the primary challenges of contemporary mobile augmented reality systems is that there is no dedicated representation for computer vision or machine learning operations within the existing permissions structures of mobile operating systems; as such, there is no requirement for developers to communicate with end users regarding their use of computer vision and machine learning operations. Our proposed solution focuses on incorporating dedicated steps for verifying machine learning components into the application development process, and on expanding the existing permissions structures of mobile operating systems to specifically target vision operations. In doing so, we will remove developers' ability to execute computer vision operations without supervision, and improve end user awareness when such operations are performed.

5.1 Proposed System

Our proposed system design consists of three principal components. The first component is a **trusted verification and signing service for computer vision models**. This service would be responsible for vetting vision functionality before it is allowed to be executed on the local device; it would offer a repository of pre-vetted, commonly used models (similar to the function-level organization of libraries such as Google's ML Kit), and provide verification services for developers to submit their own custom models built with general purpose libraries such as TensorFlow. Signed versions of the vetted models would then be available for download, along with a unique key

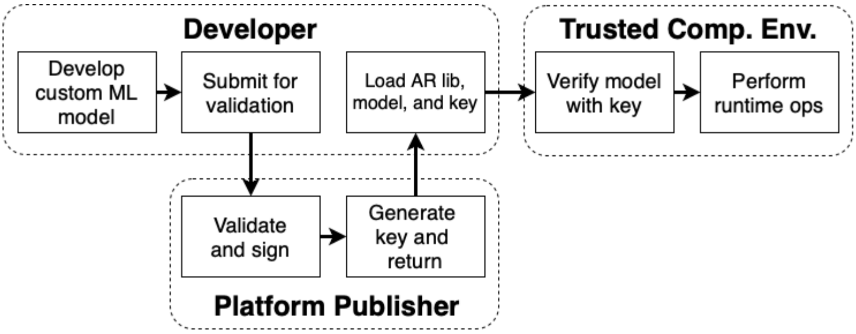


Fig. 10. Proposed workflow for custom model validation and trusted runtime operations.

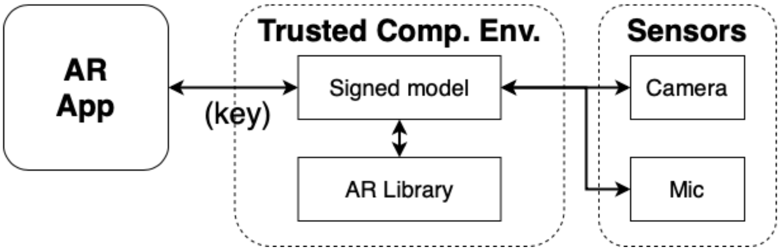


Fig. 11. Proposed system architecture for AR application with encapsulated machine learning operations

for each developer to use in authentication, thereby preventing the developer from modifying or replacing the model after download.

Figure 10 demonstrates the proposed workflow by which a developer would submit a custom model for verification. This workflow assumes that, in addition to the model itself, the developer also provides all relevant meta-data describing the model's purpose, operations, and targets to be recognized. The trusted service provider (such as Google for Android or Apple for iOS) would then perform their own independent validation, verifying that the model adheres to expected functionality and performs all advertised functions. Once the model has been verified, the service provider would return the signed version of the model as well as a unique key to the developer.

Once the developer has downloaded their model and associated key, they would then utilize our second principal component: a **trusted computing partition on the local device in which to run the signed vision model**, as shown in Figure 11. During runtime, the application would interface with the trusted partition to request operations from the custom model, providing the unique key to authenticate the request. The trusted partition would then interface with privacy-sensitive sensors such as the camera and microphone before returning the results to the application. By inserting this trusted computing layer in between the application and privacy-sensitive sensors such as the camera, the developer is able to perform approved operations using the signed model, but is unable to access the data feeds directly, removing their ability to perform additional operations on raw camera frames.

Finally, the device OS would leverage our system's third and final component, an **expanded permissions structure to communicate desired computer vision operations** to end-users at runtime. This component leverages the meta-data associated with the application's signed vision models and exposed by the trusted signing service. As such, the permissions could be structured in a number of different ways. For example, if an application is using pre-existing functionality from trusted function-level libraries such as Google's ML Kit, the OS publisher might require the developer to request permission from the user to execute that particular function (e.g. text recognition or image labelling). Alternatively, if the developer is using a signed custom model, the OS publisher could provide the end-user with the results of that model's verification process before deciding whether to permit the application to continue. The high level operation permission can also be supplemented with a brief explanation of *why* the functionality is being requested. These prompts would supplement or replace the overly vague and unhelpful request for wholesale access to the device's camera, and will provide end-users with the opportunity to reflect and decide if an application's request for a particular functionality is acceptable before engaging in use of the app.

5.2 Potential Verification Methods

There are a wide variety of established methods for testing and verifying expected behavior of software systems and machine learning modules, which the trusted signing service publisher could utilize. Examples include but not limited to such approaches as static and dynamic analysis, and metamorphic testing. The purpose of this section is not to identify the perfect verification method, but to explore how different verification methods could be incorporated into this system.

5.2.1 Static and Dynamic Analysis. Static and dynamic analysis are traditional techniques used to identify malware. Static analysis methods achieve this by creating call graphs representing the control and data flows elicited by inspecting the application's raw code files. Unfortunately, this method of analysis is inherently difficult for AR systems, due to a number of factors. As noted by Chaulagain et. al. [12], it is highly computationally expensive, suffers from high rates of false alarms, and can fail when the adversary is highly skilled in evasive code writing. In contrast to static analysis, dynamic analysis examines the behavior of the application at runtime, by feeding in particular inputs and monitoring the resulting outputs. However, dynamic analysis is only as good as the quality of the selected inputs; poor input selection can result in missed code execution paths and therefore incomplete testing. This problem is particular important for AR systems, since it is impossible to achieve 100% input coverage for a vision-based system.

Trusted signing service providers employing static and dynamic analysis can utilize our system in several ways. First, they can force developers to provide **explicit declarations of what their applications are recognizing**. This could take such forms as itemized lists from predetermined categories, collections of explicit 3D models or 2D image files, or a written description from which scope can be derived. By providing this explicit information in addition to the application's raw code files, the signing service publisher can focus its testing efforts, either to verify that a given application *does* recognize what it claims to, or that it *doesn't* recognize a particular sensitive category (such as faces or text) in which it claims to have no interest.

Unfortunately, picking inputs for runtime testing is only half the battle; the signing service publisher must also be able to tell if an application's response to a given input is honest or malicious. In order to do this using machine learning, a dataset must be available for training and testing such solutions. However, at the time of this paper's writing, no such dataset exists for mobile AR applications. Therefore, the publisher can also use our system to generate traces of such runtime information as on-board resource consumption, library function calls, network transmissions, display frame rate, and other data points. These traces can then be compiled into a **dataset for developing machine-learning-based detectors of behavioral anomalies** for AR systems. Using this dataset, the signing service publisher will be able to determine, over time, when a given application's behavior strays outside the realm of normality for a given input-output pair.

5.2.2 Metamorphic Testing. Metamorphic testing was developed to help address two problems: the "oracle problem" and the "reliable test set problem" [8, 15]. The "oracle problem" refers to the challenge of determining whether the result of a given test case is expected and therefore successful. While this may seem like a straightforward problem, in many real life systems, there is no definitive oracle for a given test set, or the resources required to fully execute the oracle may be prohibitive. The "reliable test set problem", on the other hand, refers to the difficulty in selecting a subset of tests to reliably infer system correctness when exhaustively executing all test cases is impossible. Under metamorphic testing, testers develop *metamorphic relations*, or properties that describe correct system behavior and are exhibited by a small set of "source" inputs (and their resulting system outputs). Testers then iteratively modify and expand the set of source inputs, verifying the relationships between the resulting outputs to make sure they adhere to the metamorphic relation. This allows testers to leverage a subset of the full test set to draw conclusions about the state and reliability of the system as a whole.

Mobile AR systems are particularly susceptible to both the reliable test set problem and the oracle problem because of both the massive search space for image and video inputs, and the fact that, for a given input provided to the system, the resulting output may be honest *or* dishonest, depending on the context and end-user expectations. Therefore, a signing service publisher leveraging metamorphic testing could require the developer to submit **meta-data that facilitates the construction of metamorphic relations** for use in testing. This meta-data might include the target recognition information described above, as well as mathematical equations. Work by Tian et. al. [83] shows an example of this, where minor variations in predicted steering angle for an autonomous vehicle are governed by a mean-square error calculation across subsequent input images. Signing service publishers can identify high-level system properties (such as steering angle) that are crucial to system behavior, and work with developers to construct corresponding relations.

5.3 Feasibility and Benefits

The system proposed above is high-level and somewhat abstract; this is because we recognize the scope and complexity of addressing the hidden operations threat in full is too great for this paper to cover satisfactorily. Our goal in proposing this system is not to present a single panacea solution, but rather to identify components that can be developed in synergy to mitigate the threat and its various facets. Full realization of this proposed system will require multi-disciplinary contributions from the worlds of computer vision, machine learning, systems security, and human-computer interaction.

The ultimate goal of the proposed system is to limit a developer's ability to execute vision or machine learning logic without alerting the end user. It does this through two main approaches. The first approach is by **preventing developers from executing unvetted vision logic**. By forcing developers to use trusted services to implement vision functionality (either by utilizing an existing function-level library or submitting a custom model for intense validation), we remove the adversary's ability to execute custom machine learning logic without supervision. This mitigates the LINDDUN threat of information disclosure by adding another layer of review to the app's machine learning components, preventing them from gathering more data from the user than what is permitted.

As mentioned previously in Sections 2.3 and 2.4, the testing of machine learning components and the systems that use them is a difficult problem [98, 100]. There are strategies that a given library publisher can employ to ensure the technical correctness of a given machine learning module [37, 49, 50, 56, 83], but these strategies cannot guarantee that an application developer will not use the module in ways contrary to user expectations. The second approach is, therefore, **to increase user awareness of application behavior during runtime** through the use of meta-data exposed

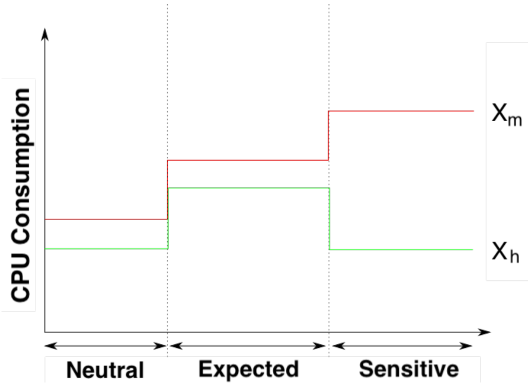


Fig. 12. The ideal case plot of X_{honest} vs $X_{malicious}$

by the trusted signing service and increased granularity of requested permissions. By providing dedicated permissions for computer vision functions and requiring end-users to explicitly approve or deny such permissions, we remove the adversary's ability to perform such operations without informing the user. This mitigates the LINDDUN threats of user unawareness and policy and consent non-compliance by ensuring that the user is as informed as possible about the true nature of an application's background operations. Once alerted to suspicious application behavior, it is then up to the user to respond appropriately, such as changing his or her own behavior or removing the application entirely.

6 EXPERIMENTAL RESULTS

Even if platform publishers implement our proposed system for application development, there is still a need for testing and verification after development is complete, such as before an application is published to a given app marketplace. Given that one of the benefits of our proposed system is the *ability to identify what an AR application is recognizing*, we are interested in determining whether this knowledge can be exploited to verify whether an AR application is executing more vision operations than it advertises, that is, conducting a hidden operations attack. We therefore conduct a series of experiments to estimate the effect that this additional knowledge can have on runtime testing practices for AR systems, in order to determine whether our proposed system shows promise for differentiating between honest and malicious AR applications.

We base our experiments on the intuition that, for a set of applications that ostensibly recognize the same honest inputs, we can expect to see low resource consumption when presented with inputs that the application does *not* recognize, and an increased level of resource consumption when presented with an input that the application *does* recognize. This intuition is reflected in Figure 12, in which inputs are divided into three categories: `neutral` (e.g. inputs that neither application recognizes), `expected` (e.g. inputs that correspond to both applications' advertised functionality), and `sensitive` (e.g. inputs which the malicious application recognizes but the honest application does not). When presented with different inputs at runtime, for the `neutral` category of inputs, resource consumption for both the honest and malicious applications should be relatively low. Conversely, when presented with inputs from the `expected` category, we expect to see a spike in resource consumption from both apps. However, when presented with `sensitive` inputs, we expect the honest application to return to a "neutral" level of resource consumption while the malicious application's levels remain high. In this section, we present two series of evaluations to

	NEUTRAL	EXPECTED	SENSITIVE
Baseline	Shoes	Single-face portraits	Street signs
Face Recog. Stress Test	Shoes	Group portraits	Street signs
Text Recog. Stress Test	Shoes	Single-face portraits	Wikipedia articles
Worst Case	Shoes	Group portraits	Wikipedia articles

Table 3. Input combinations for exploratory test conditions

verify this expected behavior first for the prototypes we developed in Section 4.3, and second for a group of commercially available third-party applications.

6.1 Test Set-up

Each application-under-test (AUT) was executed on a smartphone placed in a dock facing a computer screen. This method was utilized to in order to engage the full camera frame delivery pipeline, rather than feeding a video or set of raw image files directly to the application. The computer then played a video showing a rotation of input images. The input video was split into three phases, showing one minute each of `neutral` images (e.g. recognized by none of the applications), then `expected` images (e.g. inputs that match the applications' advertised functionality), and finally `sensitive` images (e.g. inputs that are potentially privacy-invasive and have nothing to do with the applications' advertised functionality). For all AUTs, the `expected` functionality was facial recognition, while the `sensitive` functionality was one of either age and gender estimation, or text recognition.

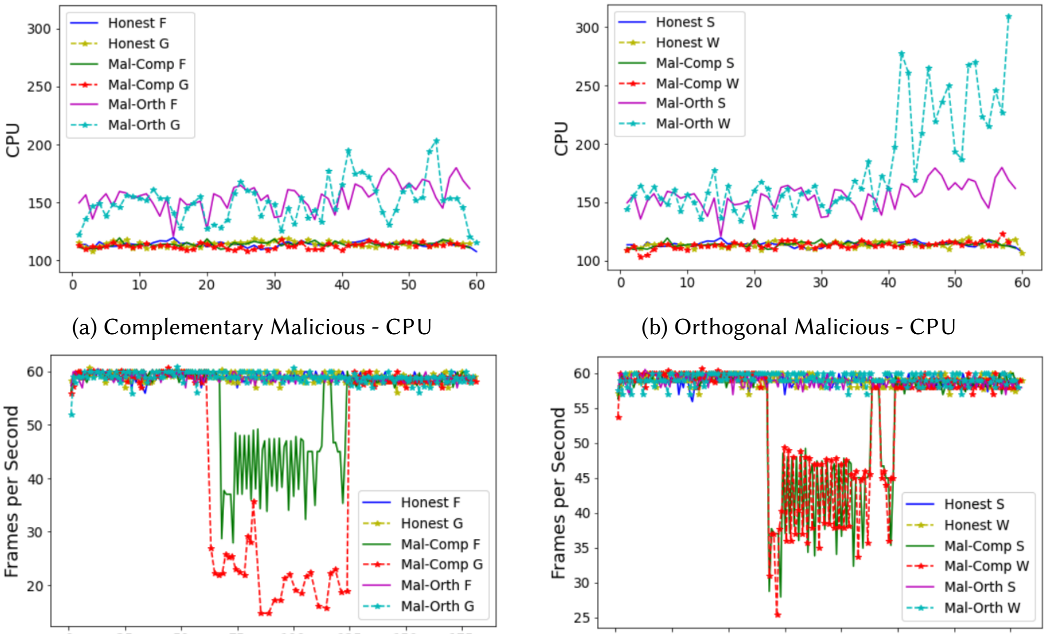
We tested four conditions: baseline, worst case, and stress tests for the facial and text recognition conditions. Input videos were generated for each condition as described in Table 3. Results were collected and averaged over five trials from each application for each video. All tests were executed using a Samsung Galaxy S9 smartphone (Qualcomm SDM845 Snapdragon 845 processor, 4 GB memory), and a 15in. MacBook Pro laptop (3.1 GHz i7 processor, 16 GB DDR3 memory). We simulated the proposed inspector system through a series of command-line scripts, executed through the Android Debug Bridge (adb) and the hardline connection between the smartphone and computer. Using this script suite, we collected a spectrum of runtime traces (CPU, memory, and storage utilization; display frame rate; active process information, etc.) as the application-under-test processed the input video. These traces were then utilized for analysis. All code files and data collection scripts used for these evaluations are publicly available on GitHub[1].

6.2 Evaluating Adversarial Applications

Our first set of tests utilized the adversarial applications described in Section 4.3, where A1 is the honest application, performing the expected operation α (e.g. facial recognition). A2 and A3 are dishonest applications, performing *complementary* and *orthogonal* hidden operations β (age and gender estimation) and γ (text recognition) respectively, in addition to α.

We analyzed the resource consumption traces to determine whether any behavioral differences could be observed for the various categories of inputs. First, we examined the traces over time for different stress test conditions to determine if we could manually observe any changes in behavior. Next, we performed statistical analysis to determine the similarity of traces collected from different apps for the same sequence of inputs. Finally, we performed K-means clustering to group traces by input category, in the attempt to distinguish malicious apps from honest.

[1] https://github.com/lehmansarahm/MAR-Security



(a) Complementary Malicious - CPU

(b) Orthogonal Malicious - CPU

(c) Complementary Malicious - Frame rate

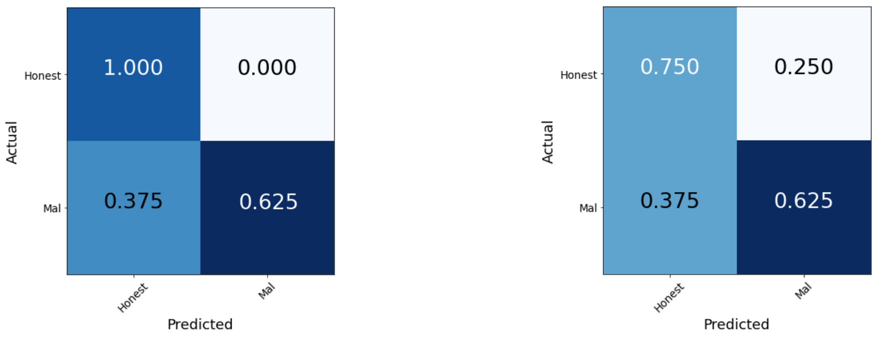
(d) Orthogonal Malicious - Frame rate

Fig. 13. Runtime traces of adversarial applications for different stress test conditions

6.2.1 Runtime Resource Consumption. We performed two stress tests to determine if we could induce a resource consumption spike for known malicious inputs, the results of which are shown in Figure 13. The first test focused on the malicious application performing the *complementary hidden operation*, that is, age and gender recognition for detected faces (A2). For this test, we considered two groups of images for the `expected` phase of the input video: single-face portraits ("F" condition) and photos of large groups ("G" condition). The second test focused on the malicious application performing the *orthogonal hidden operation*, that is, text recognition regardless of the results of the face detection output (A3). For this test, we considered two groups of images for the `sensitive` phase of the input video: street signs ("S" condition) and Wikipedia articles ("W" condition).

Impacts to runtime CPU consumption for the two stress tests are shown in Figures 13a and 13b. We can observe that the honest and complementary malicious applications exhibit relatively consistent levels of CPU consumption, regardless of the changes made to the input video. However, the orthogonal malicious application reflects a significant increase in CPU consumption for the base case, which is then further exacerbated by the stress test to almost twice the original consumption level during the `sensitive` input phase.

Impacts to the display frame rate for the two stress tests are shown in Figures 13c and 13d. Here, we observe that the honest and orthogonal malicious applications maintain relatively consistent display frame rates, regardless of input changes. However, the complementary malicious application experiences a sharp decrease in frame rate during the `expected` input phase. This effect is further magnified during the corresponding stress test, where the frame drops to one-half of the original levels during the same phase.



(a) Two-sample T-test

(b) Two-sample KS-Test

Fig. 14. Accuracy of statistical analysis for adversarial applications with a p-value threshold of 5%, where the results of each cell are the percentage of honest / malicious samples that received a given prediction.

Honest App	**Malicious App**
Neutral ≡ Sensitive	Neutral ≢ Sensitive
Neutral ≢ Expected	Neutral ≢ Expected
Expected ≢ Sensitive	Expected ≢ Sensitive

Table 4. Comparison of hypotheses for various apps

6.2.2 Statistical Analysis. Our second approach was to compare behavior of an application between the different input phases. We utilized a statistical approach to check whether the CPU distributions for the various categories of input are the same or different according to the intuition described above, that is, that an honest application will consume "neutral" levels of resources when processing sensitive inputs, while a malicious application will consume more. The expected comparison results are summarized in Table 4, where the malicious application is characterized by a significant difference in resource consumption between `neutral` and `sensitive` inputs.

We utilized two homogeneity tests to test this approach: two-sample Student's T-test (T-test), and two-sample Kolmogorov–Smirnov test (KS-Test). These tests mainly are used to determine whether two populations have different means. The *two-sample T-test* tries to find whether the means of two distributions can be similar within a margin of error depending on the distribution of both those populations. We are assuming both populations (CPU value measurements at two sections of the video) have equal variance; hence we use the *independent* (or *unpaired*) T-test. We also assume that the CPU values are distributed as normal, around a stable CPU value with fluctuations around some mean for each segment. The *two-sample KS-Test* tries to find whether the two populations have a different distribution as well as a different mean. If the difference is above a certain threshold, we say that the populations are not from the same distribution. In the case where both populations have different distributions but similar means, the KS-test will say that they are different distributions. Compared to this, the T-test can only detect difference of mean of the populations, not the difference in distribution of samples.

Next, we conducted our analysis by first aggregating the sets of five trials for each application processing each input video, for a total of twelve sets. The CPU consumption for each set was averaged and smoothed using an exponentially weighted moving average. The T-test and KS-test were then performed for each pair of input phases reflected in Table 4; the application was marked

	A1			A2			A3		
	Neut.	**Exp.**	**Sens.**	**Neut.**	**Exp.**	**Sens.**	**Neut.**	**Exp.**	**Sens.**
Precision	0.19	0.29	0.54	0.87	0.56	0.65	0.62	0.62	1.00
Recall	0.25	0.30	0.35	0.65	0.70	0.65	0.40	1.00	0.74
F-Score	0.22	0.29	0.42	0.74	0.62	0.65	0.48	0.77	0.85
Accuracy	0.30			0.67			0.71		

Table 5. K-Means classification results for adversarial applications.

as "malicious" if we observed that the three populations were not statistically homogeneous as evidenced by a low p-value (below the 0.05 threshold).

The results of our statistical analysis are shown in Figure 14. Both approaches demonstrated significant success in identifying the honest applications, and moderate success in identifying malicious applications. The most correctly-identified adversarial application was the orthogonal malicious version which performed text recognition on all frames, regardless of the presence of faces. In this respect, both approaches were able to identify three out of four orthogonal malicious traces, compared to two out of four complementary malicious traces.

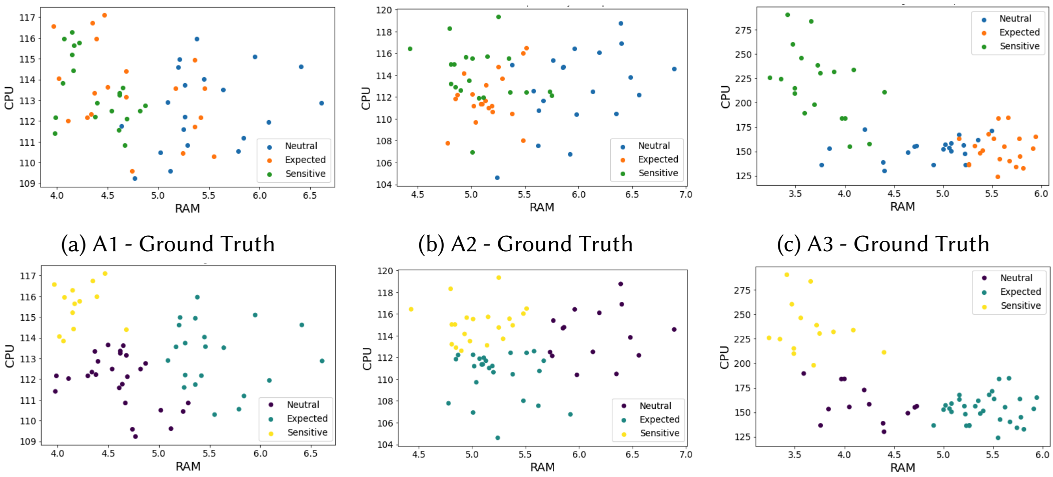
6.2.3 Clustering. Our final experiment seeks to understand whether resource consumption traces can be grouped based on the type of input being processed. To do this, we used a clustering technique. Clustering is based on unsupervised learning, and is used to partition data into groups ("clusters") depending on the similarity between data. Our intuition in this case was that the resource consumption for each "phase" of testing (e.g. when the app is processing neutral, expected, or sensitive inputs) would be sufficiently different for an algorithm to assign a given reading to one of these three known groups. For this evaluation, we used the "worst case" traces (described in Table 3) as these reflected the most dramatic differences in resource consumption by phase.

We used the resource consumption data collected from the various applications during runtime (specifically CPU and RAM) and applied the K-Means algorithm. K-means is one of the most common clustering algorithms provided by the machine learning library Scikit-Learn [64], and is an iterative algorithm that aims to partition data into k clusters. K-means clustering is a vector quantization algorithm, and as such, does not require training as neural networks do. Instead, we manually configured the number of expected clusters (k=3) to reflect the groups of `neutral`, `expected`, and `sensitive` inputs, and let the algorithm decide how best to assign the data points. The types of inputs were pre-selected based on the publicly advertised functionality of the application, and resulted in roughly the same number of collected readings per app; since the focus of this paper is on assisting researchers and commercial entities such as application marketplace providers to more effectively test applications such as this, we judge that these are reasonable test conditions.

The results of our clustering efforts are reflected in Figure 15. The ground truth resource consumption plots for the different adversarial applications are shown in Figures 15a, 15b, and 15c. The results of our algorithm's efforts in assigning those data points to clusters are shown in Figures 15d, 15e, and 15f. The overall accuracy of these clusters is summarized in Table 5, where the highest accuracy of cluster assignment (71%) is demonstrated by the orthogonal malicious application (A3).

6.3 Evaluating Commercial Applications

Our second round of evaluations considered a group of commercially available third-party applications. Three applications were selected that explicitly exhibit facial recognition: Instagram [76], Snapchat [78], and SweetCam [79]. These applications were selected due to their popularity (over 1 billion downloads for both Instagram and Snapchat, and over 100 million downloads for SweetCam)



(a) A1 - Ground Truth

(b) A2 - Ground Truth

(c) A3 - Ground Truth

(d) A1 - Predicted Clusters

(e) A2 - Predicted Clusters

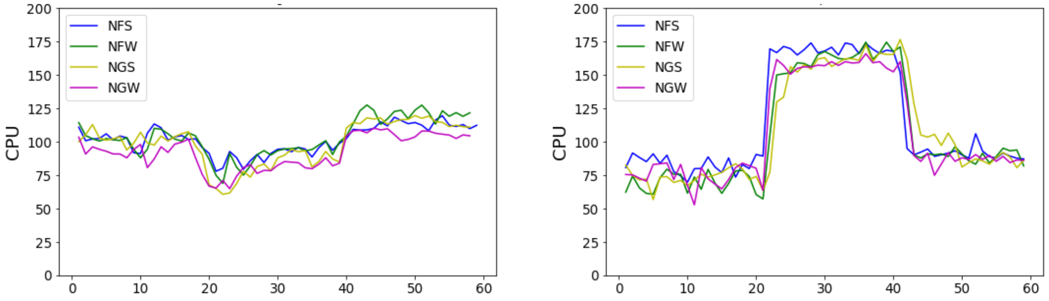
(f) A3 - Predicted Clusters

Fig. 15. K-Means clusters for adversarial applications when recognizing different classes of inputs. Cluster predictions were made using an out-of-the-box instance of SKLearn's K-means clustering algorithm for three clusters built around 10 randomly selected candidate centroid seeds.

and high reviews (3.8 stars with 120 million reviews for Instagram, 4.3 stars with 26 million reviews for Snapchat, and 4.4 stars with 930 thousand reviews for SweetCam). **While we don't believe any of these applications to be malicious, we do expect them to adhere to the behavioral patterns described in Figure 12**, where there are noticeable changes to resource consumption patterns when the application is processing an expected target, compared to "baseline" levels of consumption for both neutral and sensitive inputs.

The test conditions were the same for the commercial applications as the adversarial applications. Four input videos were used (Table 3) to cover the baseline (denoted as "noise-faces-signs" (NFS)), worst case (denoted as "noise-group-wiki" (NGW)), and stress test conditions (denoted as "noise-group-signs" (NGS) for faces and "noise-faces-wiki" (NFW) for text). Each application was configured to place a dog overlay (e.g. ears and nose) onto a person's face when recognized. The applications were run on a smartphone positioned in a dock in front of a computer screen playing the input video in order to test the full camera pipeline. The smartphone was connected to the computer through a hardline, which collected resource consumption traces. CPU and memory consumption were tracked; display framerate was not trackable without either modifying the original application or significant instrumentation to the device. Results were averaged across five trials for each application processing each input video (n=20 trials per app).

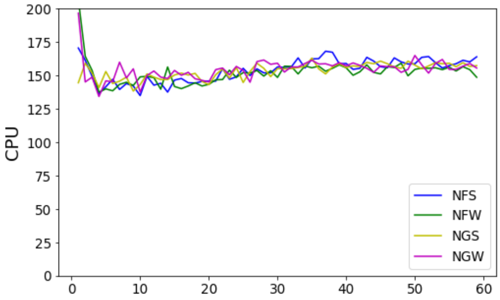
6.3.1 Conformant Resource Consumption. Figure 16 shows the runtime resource consumption of two of our commercial applications: Instagram and Snapchat. Regardless of test condition, both applications show relatively stable CPU consumption during the `neutral` and `sensitive` phases of testing (first third and final third respectively), but show different patterns during the `expected` phase (middle third) when the video was displaying faces (memory consumption showed no response to inputs and so was not included in the figure). While Snapchat exhibits exactly the expected behavior, Instagram actually *drops* slightly in CPU consumption during this phase. Our intuition is that the internal logic for tracking an already-recognized face and placing the selected overlay was a comparatively lightweight function compared to scanning and detecting



(a) Instagram CPU Consumption

(b) Snapchat CPU Consumption

Fig. 16. Resource consumption for commercial apps that conforms to our intuition.



(a) SweetCam CPU Consumption

Fig. 17. Inconclusive resource consumption for commercial app

faces in the first place. Instagram also consumed a mostly consistent amount of memory over time, while Snapchat increased memory consumption during the `expected` phase, but released it during the `sensitive` phase once faces were no longer detected. Regardless, both applications exhibit a noticeable difference in resource consumption during the `expected` input phase.

6.3.2 Inconclusive Resource Consumption. Contrary to the first two apps, the third commercial application-under-test did *not* exhibit the expected behavior. Figure 17 shows the runtime resource consumption for SweetCam, where there was no noticeable change in resource consumption during the `expected` input phase. Indeed, CPU consumption remained consistently high regardless of the test condition. Our intuition in this scenario is that the application is continually performing face recognition or other resource-intensive operations on every frame, regardless of what was previously recognized.

6.4 Results Summary

Given the proposed system described in Section 5, application marketplace publishers will have access to additional knowledge of what runtime operations a given AR application is supposed to be performing. In this section, we presented examples of how this knowledge can be leveraged to instigate noticeable behavioral changes during runtime in malicious applications. For the malicious prototypes, we were able to induce visible spikes in the resource consumption traces through stress testing of recognized target density, and classify the honest / dishonest nature of a given application with relative accuracy using both statistical analysis and K-means clustering (62.5% and 67-71% respectively). Based on the results for the prototypes with known hidden operations (A2 and A3), we conclude that **the additional meta-data provided by our proposed system shows promise in assisting with testing efforts when differentiating between honest and dishonest AR applications**.

An interesting observation is that the tests conducted above demonstrated varying levels of success in identifying the malicious applications, depending on the type of hidden operation being conducted. The hidden operations of the *complementary* malicious application were more visible in runtime monitoring of the display frame rate; the hidden operations of the *orthogonal* malicious application were more visible in the CPU traces. Both applications responded to stress testing to induce runtime resource consumption spikes, and displayed success in both detection by statistical analysis and K-means clustering. While the base prototype (A1) did not present observable changes in runtime behavior and scored lowest of the three in K-means clustering, we do not consider this to be problematic. An application which consumes consistent amounts of resources regardless of input (such as the commercial application, SweetCam) can be flagged for additional testing specifically because of its resistance to change.

A final observation is that, when applying our intuition to commercial applications, the expected behavior changes were not observed for *all* of the tested apps; some applications exhibited noticeable change to runtime behavior in response to different inputs, while others did not. While the commercial applications are presumed honest, the heterogeneity of runtime behavior patterns both supports our intuition and encourages future study on those applications which exhibit inconclusive patterns. Based on these results, and the varied success of the detection methods for the malicious prototypes, we can conclude that **no single test is a catch-all solution; a variety of testing strategies must be incorporated to detect hidden operations in AR.** The best approach for this remains an open problem. In the next section, we discuss the limitations of our proposed design, and explore open problems for future research.

7 OPEN PROBLEMS

Our proposed solution assumes that there is a pre-existing list of known privacy sensitive inputs to use during runtime. While developers can certainly create such a list for their own use, to the best of our knowledge, there are no widely acceptable datasets of communally-approved privacy sensitive images. There are also no widely used conventions for describing the contents of the datasets used to train AR models, of quantifying the quality of a dataset in representing a category of inputs. Based on these limitations, we have identified two main open areas for future explorations.

7.1 Industrial Conventions, Standards for Privacy in Camera-based Systems

The success of the system proposed above depends on the mobile and machine learning communities coming together to develop standards and conventions for dealing with privacy problems in mobile AR and other camera-based systems. Here, we present a number of open problems in this area.

- **Ontology of AR Capabilities:** There currently exists no standardized way to describe the capabilities of an AR library, custom vision model, or similar system. Further, there is no standardized way of measuring or representing the corresponding privacy sensitivity or security threat level associated with these capabilities. Thus, developing and standardizing such an ontology of vision-based capabilities and their associated privacy implications remains an open problem. Inspiration can be taken from such models as LINDDUN [20] and LINDDUN GO [94], with particular focus on elicitation and mitigation strategies for threats such as Information Disclosure, Content Unawareness, and Policy / Consent Non-compliance.

- **Improved Understanding of End-User Preferences, Context-sensitivity:** As we develop the ontology of vision-based capabilities, we must also improve our understanding of end-user privacy preferences for these capabilities, and the associated contexts under which a capability would be considered privacy-sensitive or not. Some prior work has been done in this area [13, 41], but not, to our knowledge, as it relates to the context-sensitivity of individual capabilities of vision-enabled systems. The personal and context-sensitive nature of user privacy preferences is well-documented [90, 91, 99], as are the challenges experienced by users in interpreting and understanding the nature of mobile systems' permissions requests [65, 90], but exactly how users' preferences on the application of background operations in mobile AR change over time remains unknown.
- **Improved Permissions Structures for Mobile App Development:** Finally, improving the permissions structures of mobile operating systems as they relate to machine learning and computer vision operations remains an open problem. The ontology of vision-based capabilities, their associated threat levels, and the corresponding usage contexts must be translated into a format that is compatible with mobile application development. However, asking for permission on application install or a function's first use does not take the changing nature of future usage contexts into account, while asking for permission every time the function is used quickly leads to notification fatigue [90, 91, 99]. Therefore, context-aware access control methods that move beyond simply requesting at install time should be explored, as well as methods for communicating to end-users how a particular capability is being used during runtime.

7.2 Improving Software Testing Methods for Mobile AR

As observed previously, some tests are better suited to detecting different kinds of hidden operations over others. Even within the same application, performance and behavior can change, depending on the type of input being provided. Using the CPU and memory data traces collected from the commercial applications in Section 6.3, we applied the K-means clustering approach from Section 6.2.3 to see if the applications which demonstrated conformant behavior also responded well to clustering. Table 6 summarizes the results. The accuracy for assigning data points to the appropriate clusters (e.g. neutral, expected, sensitive) was understandably much lower for Instagram and SweetCam than for the malicious prototypes, as these apps exhibited relatively consistent resource consumption, regardless of the input condition. However, Snapchat, which exhibited the most promising resource consumption trends during runtime, failed utterly during clustering, with an accuracy of only 5%. Our intuition in this case is that, while the application exhibited the expected behavior for `expected` inputs when looking at the CPU trace on its own, minor lags in either the application's response to an input phase change or the trace collection script in logging those changes were enough to throw off the clustering assignments.

It should be emphasized that *the goal of this paper is not to propose a single one-size-fits-all solution*; rather, we have identified a range of application behaviors and identified approaches that show promise in identifying different patterns. There remains, however, much work to be done in improving these processes. Here we present a number of open problems related to improving software testing methods, given the unique nature of mobile augmented reality systems and their runtime behaviors.

- **Input Selection:** It is impossible to achieve 100% input coverage with a vision-based system [8, 15]. Therefore, selecting a particular subset of inputs that can reliably determine the honest or malicious nature of a given mobile AR application remains an open problem. The

	Instagram			**Snapchat**			**SweetCam**		
	Neut.	**Exp.**	**Sens.**	**Neut.**	**Exp.**	**Sens.**	**Neut.**	**Exp.**	**Sens.**
Precision	0.27	0.73	0.00	0.00	0.04	0.10	1.00	0.14	1.45
Recall	0.35	0.95	0.00	0.00	0.05	0.11	0.15	0.10	1.00
F-Score	0.30	0.83	0.00	0.00	0.05	0.10	0.26	0.12	0.62
Accuracy	0.45			0.05			0.41		

Table 6. K-Means clustering results for CPU-memory consumption of commercial applications

impact of the order of selected inputs as well as the transfer-ability of input sets between similar applications should also be explored.

- **Test Case Selection:** Because various testing approaches have exhibited different levels of success in identifying different types of hidden operations, the selection of which test cases to execute against a given application remains an open problem. This is particularly important as we cannot assume the presence of a baseline ‘honest’ version of an application against which to compare. The efficacy of stress testing in order to trigger anomalies and system responses to inter- and intra-class inputs can also be explored.
- **Result Verification:** Finally, once inputs have been selected and the test cases have been executed, reliably interpreting the results remains an open problem. Failure to induce a resource consumption spike during runtime does not preclude the presence of hidden operations; the behavior could be the result of an honest application, a malicious application with highly effective obfuscation logic, or simply picking the wrong inputs. Therefore, better methods of verifying the results of such test instances should be explored.

8 CONCLUSIONS

In this paper, we have presented the *hidden operations privacy threat* for mobile AR systems, a scenario in which a malicious developer executes additional hidden vision operations behind the scenes of an otherwise honest mobile AR application without alerting the end-user. We presented the risks involved in this scenario as a result of the capabilities of commercially available AR and vision libraries, and presented our proposed system for mitigating these risks. We also conducted a series of experiments to determine whether the proposed system shows promise in assisting with AR system testing efforts. In the future, we would like to explore the classification of permissions and system capabilities as they relate to privacy-sensitive functions of AR libraries and custom vision models.

ACKNOWLEDGEMENTS

Dr. Ling’s support was provided in part by NSF grant number 2006665.

REFERENCES

[1] Joshua Abah, Victor Waziri, Muhammad Abdullahi, Arthur U.M, and Adewale O.S. A machine learning approach to anomaly-based detection on android platforms. *International Journal of Network Security & Its Applications*, 7:15–35, 11 2015.

[2] Vitor Monte Afonso, Matheus Favero de Amorim, André Ricardo Abed Grégio, Glauco Barroso Junquera, and Paulo Lício de Geus. Identifying android malware using dynamically obtained features. *Journal of Computer Virology and Hacking Techniques*, 11(1):9–17, Feb 2015.

[3] Kunjal Ahir, Kajal Govani, Rutvik Gajera, and Manan Shah. Application on virtual reality for enhanced education learning, military training and sports. *Augmented Human Research*, 5(1):7, 2020.

[4] Murat Akçayır and Gökçe Akçayır. Advantages and challenges associated with augmented reality for education: A systematic review of the literature. *Educational Research Review*, 20:1–11, 2017.

[5] Taslima Akter, Tousif Ahmed, Apu Kapadia, and Swami Manohar Swaminathan. Privacy considerations of the visually impaired with camera based assistive technologies: Misrepresentation, impropriety, and fairness. In *The 22nd International ACM SIGACCESS Conference on Computers and Accessibility*, pages 1–14, 2020.

[6] Taslima Akter, Bryan Dosono, Tousif Ahmed, Apu Kapadia, and Bryan Semaan. " i am uncomfortable sharing what i can't see": Privacy concerns of the visually impaired with camera based assistive applications. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020.

[7] Think Autonomous. Deep learning in self-driving cars. https://www.thinkautonomous.ai/blog/?p=deep-learning-in-self-driving-cars. Accessed: 2021-10-31.

[8] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE transactions on software engineering*, 41(5):507–525, 2014.

[9] H Borgmann, M Rodríguez Socarrás, J Salem, I Tsaur, J Gomez Rivas, E Barret, and L Tortolero. Feasibility and safety of augmented reality-assisted urological surgery using smartglass. *World journal of urology*, 35(6):967–972, 2017.

[10] An Braeken, Pawani Porambage, Andrei Gurtov, and Mika Ylianttila. Secure and efficient reactive video surveillance for patient monitoring. *Sensors*, 16(1):32, 2016.

[11] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 15–26. ACM, 2011.

[12] Dewan Chaulagain, Prabesh Poudel, Prabesh Pathak, Sankardas Roy, Doina Caragea, Guojun Liu, and Xinming Ou. Hybrid analysis of android apps for security vetting using deep learning. In *2020 IEEE Conference on Communications and Network Security (CNS)*, pages 1–9. IEEE, 2020.

[13] Andrew Tzer-Yeu Chen, Morteza Biglari-Abhari, I Kevin, and Kai Wang. Context is king: Privacy perceptions of camera-based surveillance. In *2018 15th IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS)*, pages 1–6. IEEE, 2018.

[14] Peng Chen, Xiaolin Liu, Wei Cheng, and Ronghuai Huang. A review of using augmented reality in education from 2011 to 2016. In *Innovations in smart learning*, pages 13–18. Springer, 2017.

[15] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, TH Tse, and Zhi Quan Zhou. Metamorphic testing: A review of challenges and opportunities. *ACM Computing Surveys (CSUR)*, 51(1):1–27, 2018.

[16] Namho Chung, Heejeong Han, and Youhee Joun. Tourists' intention to visit a destination: The role of augmented reality (ar) application for a heritage site. *Computers in Human Behavior*, 50:588–599, 2015.

[17] David F Cihak, Eric J Moore, Rachel E Wright, Don D McMahon, Melinda M Gibbons, and Cate Smith. Evaluating augmented reality to complete a chain task for elementary students with autism. *Journal of Special Education Technology*, 31(2):99–108, 2016.

[18] Google Cloud. Machine learning apis | cloud apis. https://cloud.google.com/apis#section-6. Accessed: 2020-12-05.

[19] Scott G Dacko. Enabling smart retail settings via mobile augmented reality shopping apps. *Technological Forecasting and Social Change*, 124:243–256, 2017.

[20] Mina Deng, Kim Wuyts, Riccardo Scandariato, Bart Preneel, and Wouter Joosen. A privacy threat analysis framework: supporting the elicitation and fulfillment of privacy requirements. *Requirements Engineering*, 16(1):3–32, 2011.

[21] Apple Developers. Arkit. https://developer.apple.com/arkit/. Accessed: 2020-09-04.

[22] Google Developers. Arcore. https://developers.google.com/ar/. Accessed: 2020-09-04.

[23] Google Developers. Image labeling | ml kit. https://developers.google.com/ml-kit/vision/image-labeling. Accessed: 2020-12-05.

[24] Google Developers. Ml kit. https://developers.google.com/ml-kit. Accessed: 2020-12-05.

[25] Justin Dunn, Elizabeth Yeo, Parisah Moghaddampour, Brian Chau, and Sarah Humbert. Virtual and augmented reality in the treatment of phantom limb pain: A literature review. *NeuroRehabilitation*, 40(4):595–601, 2017.

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

[27] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the eighth symposium on usable privacy and security*, pages 1–14, 2012.

[28] Md Sadek Ferdous, Soumyadeb Chowdhury, and Joemon M Jose. Analysing privacy in visual lifelogging. *Pervasive and Mobile Computing*, 40:430–449, 2017.

[29] Shalini Ghosh, Amaury Mercier, Dheeraj Pichapati, Susmit Jha, Vinod Yegneswaran, and Patrick Lincoln. Trusted neural networks for safety-constrained autonomous control. *arXiv preprint arXiv:1805.07075*, 2018.

[30] Mar Gonzalez-Franco, Rodrigo Pizarro, Julio Cermeron, Katie Li, Jacob Thorn, Windo Hutabarat, Ashutosh Tiwari, and Pablo Bermell-Garcia. Immersive mixed reality for manufacturing training. *Frontiers in Robotics and AI*, 4:3, 2017.

[31] Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. Checking app behavior against app descriptions. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1025–1035. ACM, 2014.

[32] Nicholas Halliwell and Freddy Lecue. Trustworthy convolutional neural networks: A gradient penalized-based approach. *arXiv preprint arXiv:2009.14260*, 2020.

[33] Jason Hayhurst. How augmented reality and virtual reality is being used to support people living with dementia—design challenges and future directions. In *Augmented reality and virtual reality*, pages 295–305. Springer, 2018.

[34] Suman Jana, David Molnar, Alexander Moshchuk, Alan Dunn, Benjamin Livshits, Helen J Wang, and Eyal Ofek. Enabling fine-grained permissions for augmented reality applications with recognizers. In *Presented as part of the 22nd {USENIX} Security Symposium ({USENIX} Security 13)*, pages 415–430, 2013.

[35] Suman Jana, Arvind Narayanan, and Vitaly Shmatikov. A scanner darkly: Protecting user privacy from perceptual applications. In *2013 IEEE symposium on security and privacy*, pages 349–363. IEEE, 2013.

[36] Timothy Jung, M Claudia tom Dieck, Hyunae Lee, and Namho Chung. Effects of virtual reality and augmented reality on visitor experiences in museum. In *Information and communication technologies in tourism 2016*, pages 621–635. Springer, 2016.

[37] Jinhan Kim, Robert Feldt, and Shin Yoo. Guiding deep learning system testing using surprise adequacy. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1039–1049. IEEE, 2019.

[38] Marion Koelle, Swamy Ananthanarayan, Simon Czupalla, Wilko Heuten, and Susanne Boll. Your smart glasses' camera bothers me! exploring opt-in and opt-out gestures for privacy mediation. In *Proceedings of the 10th Nordic Conference on Human-Computer Interaction*, pages 473–481, 2018.

[39] H. Kurniawan, Y. Rosmansyah, and B. Dabarsyah. Android anomaly detection system using machine learning classification. In *2015 International Conference on Electrical Engineering and Informatics (ICEEI)*, pages 288–293, 2015.

[40] Kiron Lebeck, Kimberly Ruth, Tadayoshi Kohno, and Franziska Roesner. Arya: Operating system support for securely augmenting reality. *IEEE Security & Privacy*, 16(1):44–53, 2018.

[41] Kiron Lebeck, Kimberly Ruth, Tadayoshi Kohno, and Franziska Roesner. Towards security and privacy for multi-user augmented reality: Foundations with end users. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 392–408. IEEE, 2018.

[42] Sarah Lehman, Jenna Graves, Carlene Mcaleer, Tania Giovannetti, and Chiu C Tan. A mobile augmented reality game to encourage hydration in the elderly. In *International Conference on Human Interface and the Management of Information*, pages 98–107. Springer, 2018.

[43] Sarah M Lehman, Abrar S Alrumayh, Haibin Ling, and Chiu C Tan. Stealthy privacy attacks against mobile ar apps. In *2020 IEEE Conference on Communications and Network Security (CNS)*, pages 1–5. IEEE, 2020.

[44] Dong Li, Danhao Guo, Weili Han, Hao Chen, Chang Cao, and Xiaoyang Sean Wang. Camera-recognizable and human-invisible labelling for privacy protection. In *2016 12th International Conference on Mobile Ad-Hoc and Sensor Networks (MSN)*, pages 365–369. IEEE, 2016.

[45] Bin Liu, Mads Schaarup Andersen, Florian Schaub, Hazim Almuhimedi, Shikun Aerin Zhang, Norman Sadeh, Yuvraj Agarwal, and Alessandro Acquisti. Follow my recommendations: A personalized privacy assistant for mobile app permissions. In *Twelfth Symposium on Usable Privacy and Security ({SOUPS} 2016)*, pages 27–41, 2016.

[46] Jianhui Liu and Qi Zhang. Code-partitioning offloading schemes in mobile edge computing for augmented reality. *IEEE Access*, 7:11222–11236, 2019.

[47] Runpeng Liu, Joseph P Salisbury, Arshya Vahabzadeh, and Ned T Sahin. Feasibility of an autism-focused augmented reality smartglasses system for social communication and behavioral coaching. *Frontiers in pediatrics*, 5:145, 2017.

[48] Songhao Lou, Shaoyin Cheng, Jingjing Huang, and Fan Jiang. Tfdroid: Android malware detection by topics and sensitive data flows using machine learning techniques. In *2019 IEEE 2nd International Conference on Information and Computer Technologies (ICICT)*, pages 30–36. IEEE, 2019.

[49] Lei Ma, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu, et al. Deepgauge: Multi-granularity testing criteria for deep learning systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 120–131, 2018.

[50] Lei Ma, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Felix Juefei-Xu, Chao Xie, Li Li, Yang Liu, Jianjun Zhao, et al. Deepmutation: Mutation testing of deep learning systems. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*, pages 100–111. IEEE, 2018.

[51] Sotiris Makris, Panagiotis Karagiannis, Spyridon Koukas, and Aleksandros-Stereos Matthaiakis. Augmented reality system for operator support in human–robot collaborative assembly. *CIRP Annals*, 65(1):61–64, 2016.

[52] Microsoft. Getting started with mrtk | mixed reality toolkit documentation. https://microsoft.github.io/MixedRealityToolkit-Unity/Documentation/GettingStartedWithTheMRTK.html. Accessed: 2020-10-30.

[53] Alejandro Mitaritonna, María José Abásolo, and Francisco Montero. An augmented reality-based software architecture to support military situational awareness. In *2020 International Conference on Electrical, Communication, and Computer Engineering (ICECCE)*, pages 1–6. IEEE, 2020.

[54] Helen Nissenbaum. Privacy as contextual integrity. *Wash. L. Rev.*, 79:119, 2004.

[55] OpenCV. Opencv library. https://opencv.org. Accessed: 2020-09-26.

[56] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *proceedings of the 26th Symposium on Operating Systems Principles*, pages 1–18, 2017.

[57] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. Towards practical verification of machine learning: The case of computer vision systems. *arXiv preprint arXiv:1712.01785*, 2017.

[58] Atieh Poushneh and Arturo Z Vasquez-Parraga. Discernible impact of augmented reality on retail customer's experience, satisfaction and willingness to buy. *Journal of Retailing and Consumer Services*, 34:229–234, 2017.

[59] Philip Pratt, Matthew Ives, Graham Lawton, Jonathan Simmons, Nasko Radev, Liana Spyropoulou, and Dimitri Amiras. Through the hololens™ looking glass: augmented reality for extremity reconstruction surgery using 3d vascular models with perforating vessels. *European radiology experimental*, 2(1):2, 2018.

[60] Philipp A Rauschnabel, Alexander Rossmann, and M Claudia tom Dieck. An adoption framework for mobile augmented reality games: The case of pokémon go. *Computers in Human Behavior*, 76:276–286, 2017.

[61] Nisarg Raval, Animesh Srivastava, Ali Razeen, Kiron Lebeck, Ashwin Machanavajjhala, and Lanodn P Cox. What you mark is what apps see. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, pages 249–261, 2016.

[62] Franziska Roesner, David Molnar, Alexander Moshchuk, Tadayoshi Kohno, and Helen J Wang. World-driven access control for continuous sensing. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1169–1181, 2014.

[63] Giovanni Schiboni, Fabio Wasner, and Oliver Amft. A privacy-preserving wearable camera setup for dietary event spotting in free-living. In *2018 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, pages 872–877. IEEE, 2018.

[64] Scikit-Learn. Machine learning in python. https://scikit-learn.org/stable/. Accessed: 2020-10-30.

[65] Bingyu Shen, Lili Wei, Chengcheng Xiang, Yudong Wu, Mingyao Shen, Yuanyuan Zhou, and Xinxin Jin. Can systems explain permissions better? understanding users' misperceptions under smartphone runtime permission model. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[66] Jiayu Shu, Rui Zheng, and Pan Hui. Cardea: Context-aware visual privacy protection from pervasive cameras. *arXiv preprint arXiv:1610.00889*, 2016.

[67] Raphael Spreitzer, Felix Kirchengast, Daniel Gruss, and Stefan Mangard. Procharvester: Fully automated analysis of procfs side-channel leaks on android. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 749–763, 2018.

[68] Raphael Spreitzer, Gerald Palfinger, and Stefan Mangard. Scandroid: Automated side-channel analysis of android apis. In *Proceedings of the 11th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, pages 224–235, 2018.

[69] Animesh Srivastava, Puneet Jain, Soteris Demetriou, Landon P Cox, and Kyu-Han Kim. Camforensics: Understanding visual privacy leaks in the wild. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, pages 1–13, 2017.

[70] Statista. Global augmented/virtual reality market size 2016-2024. https://www.statista.com/statistics/591181/global-augmented-virtual-reality-market-size/. Accessed: 2020-12-12.

[71] Google Play Store. Ar apps. http://bit.ly/GooglePlay_ArApps. Accessed: 2020-10-30.

[72] Google Play Store. Ar games. http://bit.ly/GooglePlay_ArGames. Accessed: 2020-10-30.

[73] Google Play Store. Best of ar. http://bit.ly/GooglePlay_BestOfAr. Accessed: 2020-10-30.

[74] Google Play Store. Google photos. https://play.google.com/store/apps/details?id=com.google.android.apps.photos. Accessed: 2020-11-14.

[75] Google Play Store. Google spotlight stories. https://play.google.com/store/apps/details?id=com.google.android.spotlightstories. Accessed: 2020-11-14.

[76] Google Play Store. Instagram - apps on google play. https://play.google.com/store/apps/details?id=com.instagram.android. Accessed: 2021-06-15.

[77] Google Play Store. Mondly. https://play.google.com/store/apps/details?id=com.atistudios.mondly.languages. Accessed: 2020-11-14.

[78] Google Play Store. Snapchat - apps on google play. https://play.google.com/store/apps/details?id=com.snapchat.android. Accessed: 2021-06-15.

[79] Google Play Store. Sweet face camera - live face filters for snapchat - apps on google play. https://play.google.com/store/apps/details?id=com.ufotosoft.justshot. Accessed: 2021-06-15.

[80] Carlos Suso-Ribera, Javier Fernández-Álvarez, Azucena García-Palacios, Hunter G Hoffman, Juani Bretón-López, Rosa M Banos, Soledad Quero, and Cristina Botella. Virtual reality, augmented reality, and in vivo exposure therapy: a preliminary comparison of treatment efficacy in small animal phobia. *Cyberpsychology, Behavior, and Social Networking*, 22(1):31–38, 2019.

[81] TensorFlow. Open-source machine learning framework. https://www.tensorflow.org. Accessed: 2020-09-04.

[82] Peter Teufl, Michaela Ferk, Andreas Fitzek, Daniel Hein, Stefan Kraxberger, and Clemens Orthacker. Malware detection by applying knowledge discovery processes to application metadata on the android market (google play). *Security and communication networks*, 9(5):389–419, 2016.

[83] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th international conference on software engineering*, pages 303–314, 2018.

[84] M Claudia tom Dieck and Timothy Jung. A theoretical model of mobile augmented reality acceptance in urban heritage tourism. *Current Issues in Tourism*, 21(2):154–174, 2018.

[85] Iis P Tussyadiah, Timothy Hyungsoo Jung, and M Claudia tom Dieck. Embodiment of wearable augmented reality technology in tourism experiences. *Journal of Travel research*, 57(5):597–611, 2018.

[86] Unity. Unity3d game engine. www.unity3d.com. Accessed: 2020-09-04.

[87] Google VR. Google cardboard. https://arvr.google.com/cardboard/. Accessed: 2020-12-12.

[88] Vuforia. Augmented reality framework. https://www.vuforia.com. Accessed: 2020-09-04.

[89] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. Efficient formal safety analysis of neural networks. *arXiv preprint arXiv:1809.08098*, 2018.

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

[91] Primal Wijesekera, Arjun Baokar, Lynn Tsai, Joel Reardon, Serge Egelman, David Wagner, and Konstantin Beznosov. The feasibility of dynamically granted permissions: Aligning mobile privacy with user preferences. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 1077–1093. IEEE, 2017.

[92] Dennis Wolf, Daniel Besserer, Karolina Sejunaite, Matthias Riepe, and Enrico Rukzio. care: An augmented reality support system for dementia patients. In *The 31st Annual ACM Symposium on User Interface Software and Technology Adjunct Proceedings*, pages 42–44, 2018.

[93] Zhenyu Wu, Zhangyang Wang, Zhaowen Wang, and Hailin Jin. Towards privacy-preserving visual recognition via adversarial training: A pilot study. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 606–624, 2018.

[94] Kim Wuyts, Laurens Sion, and Wouter Joosen. Linddun go: A lightweight approach to privacy threat modeling. In *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 302–309. IEEE, 2020.

[95] Zuobin Xiong, Zhipeng Cai, Qilong Han, Arwa Alrawais, and Wei Li. Adgan: Protect your location privacy in camera data of auto-driving vehicles. *IEEE Transactions on Industrial Informatics*, 2020.

[96] Zuobin Xiong, Wei Li, Qilong Han, and Zhipeng Cai. Privacy-preserving auto-driving: a gan-based approach to protect vehicular camera data. In *2019 IEEE International Conference on Data Mining (ICDM)*, pages 668–677. IEEE, 2019.

[97] AWW Yew, SK Ong, and AYC Nee. Towards a griddable distributed manufacturing system with augmented reality interfaces. *Robotics and Computer-Integrated Manufacturing*, 39:43–55, 2016.

[98] Jie M Zhang, Mark Harman, Lei Ma, and Yang Liu. Machine learning testing: Survey, landscapes and horizons. *IEEE Transactions on Software Engineering*, 2020.

[99] Shikun Zhang, Yuanyuan Feng, Lujo Bauer, Lorrie Faith Cranor, Anupam Das, and Norman Sadeh. “did you know this camera tracks your mood?”: Understanding privacy expectations and preferences in the age of video analytics. *Proceedings on Privacy Enhancing Technologies*, 2021(2):282–304, 2021.

[100] Tianyi Zhang, Cuiyun Gao, Lei Ma, Michael Lyu, and Miryung Kim. An empirical study of common challenges in developing deep learning applications. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, pages 104–115. IEEE, 2019.

[101] Wenxiao Zhang, Bo Han, and Pan Hui. On the networking challenges of mobile augmented reality. In *Proceedings of the Workshop on Virtual Reality and Augmented Reality Network*, pages 24–29, 2017.

[102] Ágnes Zsila, Gábor Orosz, Beáta Bőthe, István Tóth-Király, Orsolya Király, Mark Griffiths, and Zsolt Demetrovics. An empirical study on the motivations underlying augmented reality games: The case of pokémon go during and after pokémon fever. *Personality and individual differences*, 133:56–66, 2018.