AFRL-RI-RS-TR-2016-111

SPECIALIZED BINARY ANALYSIS FOR VETTING ANDROID APPS USING GUI LOGIC

UNIVERSITY OF MARYLAND

APRIL 2016

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2016-111 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /
MARK K. WILLIAMS
Work Unit Manager

/ S /
WARREN H. DEBANY, JR.
Technical Advisor, Information
Exploitation & Operations Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
APRIL 2016	FINAL TECHNICAL REPORT	NOV 2013 – DEC 2015

4. TITLE AND SUBTITLE

SPECIALIZED BINARY ANALYSIS FOR VETTING ANDROID APPS USING GUI LOGIC

5a. CONTRACT NUMBER
FA8750-14-2-0039

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
61101E

6. AUTHOR(S)

Atif Memon

5d. PROJECT NUMBER
APAC

5e. TASK NUMBER
MA

5f. WORK UNIT NUMBER
RY

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

University of Maryland
College Park, MD 20742-3255

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RIGB
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSOR/MONITOR'S REPORT NUMBER
AFRL-RI-RS-TR-2016-111

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This project develops a suite of specialized analysis techniques for vetting Android applications to confirm the presence or rule out the absence of malice. The definition of malice of interest is an inconsistency between the action taken by the app and the user expectation of what the app is doing. These techniques enable security analysts to quickly vet any given Android app even if the source code is unavailable. These techniques make it possible to vet a large number of Android apps in a timely and cost-effective manner.

15. SUBJECT TERMS

Malware Detection for Android applications, Binary Analysis

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			**MARK K. WILLIAMS**
U	U	U	SAR	43	19b. TELEPHONE NUMBER (*Include area code*) **(315) 330-4560**

Contents

List of Figures

List of Tables

1 SUMMARY

This project develops a suite of specialized analysis techniques for vetting Android applications to confirm the presence or rule out the absence of malice. The definition of malice of interest is an *inconsistency* between the action taken by the app and the user expectation of what the app is doing. These techniques enable security analysts to quickly vet any given Android app even if the source code is unavailable. These techniques make it possible to vet a large number of Android apps in a timely and cost-effective manner.

The project develops 4 complementary techniques. First is based on detecting inconsistencies between (a) the public information of an app and its user interface (UI), (b) the UI and the actual behavior, or (c) the public information and the actual behavior. For example, turning on the camera (actual behavior) when there is no button related to camera operations is a potential security indicator. Second is based on combining information extracted from an app's source code and marketplace webpage to identify correlated variables and validate an app's quality properties such as its intended behavior, trust or suspiciousness. This work involved analysis of artifacts such as the GUI text, user ratings, app description keywords, permission requests, and sensitive Android library invocations. Third, the project builds a mental model that a GUI user creates during app usage; a model that implicitly informs the user of the software designer's intent. Specifically, the model focuses on an important question used for security assessment of Android apps: "What permission-sensitive behaviors does this app exhibit?" Finally, we use static analysis to connect security sensitive operations to GUI components.

The project experimentally evaluated each of the above techniques. Experimental results show that each of our approaches has complementary strengths, and together, form a powerful paradigm for Android app security assessment using UI logic.

2 INTRODUCTION

The use of mobile devices has created an enormous and lively marketplace for application development and deployment [11, 41, 42, 58]. A developer can create a mobile application (or *app*) and provide it for users to download with only a small monetary investment [20, 47]. Apps exist to help users with almost any conceivable task. For instance, to track finances, provide meta-data about products when shopping, manage health, or play games (either alone or with friends). Millions of apps are found in marketplaces like the Google Play store [3], yet this type of distributed development comes with some downsides [12, 35, 53, 64]. For the user who is concerned about application quality, it is difficult to discern what an app actually does [49], or to know if one can trust that it won't perform malicious acts [21, 43, 65]. If an unexpected behavior occurs, it may not be clear if this is intended or is a fault – without a formal specification [36, 69], it is not possible to reliably check the behavior of these apps [44]. DoD app marketplaces are no exception.

Android leads the US market with a market share of 60% (Dec. 2015), which is projected to reach 68% in 2016. The number of Android apps available is well over 1.5 Million. Currently, it is not possible to confirm the absence of hidden malice in these apps. As a result, organizations that require high security standards, such as government agencies and banks, often take the pessimistic approach of assuming most of the apps are unsafe unless proven otherwise. Employees of these organizations are restricted to only a handful of apps vetted by security analysts. This restriction has adverse effects on employee productivity and happiness. An employee may want to use a newly released app that offers many attractive timesaving features to boost productivity. Another employee may want to use an email app she is familiar and happy with rather than using the one officially sanctioned. Both employees must seek approval from security analysts but both are likely to be disappointed. An organization may not have enough security analysts to vet every single app employees want to use. Third-party app developers may not always cooperate by providing source code for security analysts to examine. "Your approval request is denied due to security concerns" is often the most convenient and safest response, at the unfortunate loss in productivity and happiness.

This project develops a suite of specialized analysis techniques for vetting Android apps to confirm the absence of malice. These techniques enable security analysts to quickly vet any given Android app even if the source code is unavailable. The definition of malice of interest is an *inconsistency* between the action taken by the app and the user expectation of what the app is doing.

Example 1 (Expectation Inconsistency)**.** A camera app is designed to take pictures. There are two execution paths in this app:

Normal The user presses a button labeled "Take Picture," then the app captures a pictures tags it with the time and location. The user presses a button labeled "Upload Picture," and the app uploads the picture to a desired cloud service.

Malicious The app also takes a picture quietly in the background and uploads to a central server *without the user's knowledge.*

Such a malicious app is particularly pernicious because the app does not seem malicious from a permission audit perspective. In order to carry out the malicious behavior, this app must request the permission to take a picture (`android.permission.CAMERA`) and to upload (`android.permission.INTERNET`), but these requests appear legitimate from the normal execution path.

The key novelty of this project is the use of high-level reasoning based on the GUI design logic of an app to enable a security analyst to diagnose and triage the potentially *sensitive* execution paths of an app.

Levels of Inconsistency We have identified three-levels of logical inconsistencies:

Event-level inconsistency A sensitive operation (e.g., taking a picture) is not trigged by user action on a GUI component.

Layout-level inconsistency A sensitive operation is triggered by a GUI component that may be deceptive to user due visibility or layout.

Semantic-level inconsistency A sensitive operation is triggered by a GUI component that may be deceptive to use due to a semantic inconsistency in the labels (e.g., a "Save" button triggers picture taking).

This project develops 4 complementary techniques to detect the above levels of inconsistency. The first uses the fact that a user typically learns about an app from the app's public information (while deciding whether to install it), from the app's UI (while exploring the UI), and from the app's actual behaviors (while using it). Users may become confused or surprised if there are inconsistencies between (a) the public information and UI, (b) the UI and the actual behavior, or (c) the public information and the actual behavior. For example, turning on the camera (actual behavior) when there is no button that says SNAP (UI) is a potentially confusing and even malicious inconsistency. We present a methodology for automatically detecting inconsistencies in Android apps with respect to permissions and similarity. We report results on a large corpus of 178,765 apps.

Second, to combine information extracted from an app's source code and marketplace webpage to identify correlated variables and validate an app's quality properties such as its intended behavior, trust or suspiciousness. This work involved analysis of one or two artifacts such as the GUI text, user ratings, app description keywords, permission requests, and sensitive API calls. Previous such approaches have made assumptions about how the various artifacts are populated and used by developers, which may lead to a gap in the resulting analysis. We take a step back and perform an in-depth study of 14 popular apps from the Google Play Store. We have studied a set of 16 different artifacts for each app, and conclude that the output of these must be pieced together to form a complete understanding of the app's true behavior. We show that (1) developers are inconsistent in where and how they provide descriptions; (2) each artifact alone has incomplete information; (3) different artifacts may contain contradictory pieces of information; (4) there is a need for new analyses, such as those that use image processing; and (5) without including analyses of advertisement libraries, the complete behavior of an app is not defined. In addition, we show that the

number of downloads and ratings of an app does not appear to be a strong predictor of overall app quality, as these are propagated through versions and are not necessarily indicative of the current app version's behavior.

Our third technique outputs a mental model that a GUI user creates during app usage; a model that implicitly informs the user of the software designer's intent. Our model specifically considers an important question used for security assessment of Android apps: "What permission-sensitive behaviors does this app exhibit?" Our assessment is based on the comparison of 2 mental models of 12 Android apps – one derived from the app's usage and the other from its public description. We compare these two models with a third, automatically derived model – the permissions the app seeks from Android. Our results show that the usage-based model provides unique insights into app behavior. This model's consistency with other behavioral information about the app may be used in security assessment. Finally, we develop specialized static analysis to connect GUI actions to the underlying source code.

These techniques make it possible to vet a large number of Android apps in a timely and cost-effective manner. Organizations will no longer need to tradeoff productivity and happiness for security.

3 METHODS, ASSUMPTIONS, AND PROCEDURES

3.1 Descriptor-Based Analysis

Android uses a permission system to protect the device's resources and user's data. There are many apps that ask for unexpected permissions. Why does this app ask for the camera permission but never mention a word about it? Why does this app ask for the camera permission but never invoke any API method to operate the camera? Is this a bug, malice, or a feature? These are potential inconsistencies that often confuse users.

Within a group of apps, there may exist some apps significantly different from the rest. As an example, consider a set of apps grouped together because they all require camera permissions. After analyzing a large number of these apps, we may spot a common pattern exhibited by the majority of these apps. The pattern may be that an app's public description tends to contain words such as capture, take picture, or photo. A user might be confused when he finds an app that says nothing publicly about using cameras even though it explicitly requires camera permissions. Inconsistency may occur between various levels. Consider an app whose description includes words like take a picture but it doesn't include anything at the UI level. Additionally, the UI may include "take a picture" button, yet that button doesn't trigger the phone's camera. As a result of this inconsistency, users may be surprised, annoyed, confused, or even harmed. We studied two questions regarding inconsistencies in Android apps:

1. How can we compute a descriptor to provide a comprehensive and comparable view of an app?

2. How can we apply such descriptor to discover inconsistencies between an app's public features, UI features, and code features?

To address these questions, we took a data-driven approach. We collected a corpus of 178,765 Android apps from the Google Play Store. We computed descriptors of Android apps by combining features extracted at three levels. At the public level, we considered an app's online description, package name, category, ratings, install size, and other publicly available information. At the UI level, we considered an app's user interface's text and layout. At the code level, we considered the methods declared, invoked and the connections between event handlers (e.g., onClick) and library calls (e.g., takePicture), using program analysis.

We begin our analysis of inconsistencies in Android apps by computing a descriptor for each app. The role of a descriptor is to give a comprehensive and comparative view of an app. A descriptor should comprehensively capture as many aspects about an app as possible. It should also allow efficient and effective comparison. We propose a novel descriptor composed of features extracted at three levels.

Three-level features are extracted to form a descriptor of each app. *Public Features* are derived from public information about an app and visible to users before the app is installed. We extracted public features about an app from the details published on the Google Play Store. We used Google-Play-Crawler,[1] an unofficial open source API for the Google Play Store, to download APK files, collect package names, reviews, permissions, title, creator, and number of downloads for each application from the Google Play Store. NOKOGIRI[2] was used to gather further information from each app's Google Play Store web page including: app description, category, rating, date published, Play Store URL, price, version, operating system, ratings count, content rating, developer URL, install size, and downloads count text.

User Interface (UI) Features are derived from the user interface of an app, including the text and layout. They are visible only to users who have installed and are using the app. We extracted two types of user interface features: text and layout. We downloaded each app and saved it as an APK file (Android Package). We used an open source reverse-engineering tool, to unpack the APK file into a directory tree of program files that make up the app. To extract layout features, we examined the res/layout directory for layout files. We parsed them all using a custom XML parser and collected all the widgets we encountered as features. To extract text features, we parsed three string resources files, (strings.xml, arrays.xml, and plurals.xml). We extracted strings from layout files when strings are hardcoded into layout files.

Code Features are derived from the disassembled and decompiled code of an app. We are interested in the following code features: (a) an app's own methods, (b) the Android library methods an app invokes, and (c) the connection between a user event handler (e.g., onClick) and the method triggered by the event (e.g., takePicture). These features are not visible to users but can be inspected by an expert program analyst to examine the app's actual behavior. We extracted three types of code features: declared methods, invoked methods, and pairs. We used apktool to unpack an app's APK file.

We located the file classes.dex, which bundles the binary code of all the classes of an app into a single file. We used smali to disassemble classes.dex into individual files, one per class, in a human-readable assembly-like format. To extract declared method features, we looked for the pattern .method [declared name] in every smali file and collected all occurrences. To

[1]https://github.com/Akdeniz/google-play-crawler

[2]http://nokogiri.org

extract invoke method features, we looked for the pattern .invoke-virtual and .invoke-public. To extract method pair features, we built a call graph for each app using the WALA program analysis framework. WALA takes Java classes.dex into Java bytecode. Our call graphs are object-sensitive with unlimited context sensitivity for container classes.

3.2 App Artifact Synthesis

We now describe a synthesis-based approach that we developed as there are no formal requirements that must be adhered to when marketing an app to a marketplace [42, 59, 64], and no set standards for how to describe and document exactly how an app is supposed to behave [45, 62]. The marketplaces provide ample space for developers to upload this information, but as we have learned, the structure for documentation is loose and its use open to choice and style [46, 66]. For instance, a developer can write a complete natural language description of their app, or they can provide screenshots or videos – or use a mixture [25]. They might instead, rely on the crowd and provide only a brief description, expecting users to comment on the other behavior within their reviews [37, 60, 67]. Some hints can be inferred about an application's intention [24], such as the request for user permissions [28] to utilize particular resources (e.g. a camera), however, some apps use alternative mechanisms that rely on other apps (and their permissions) to access these resources (e.g. an app might use a camera Intent to utilize a camera via another app and this would not require explicit user permissions [61]), or they may over request, meaning that the user has no real idea of what the actual behavior will be [27]. Having such flexibility makes the release of new apps easy on the developer, however it means that users have incomplete information, and may end up surprised later. Even the use of a simple mechanism such as user reviews seems to be open to interpretation, since as we have learned, reviews from one version of an app appear to follow it to the next version. New faulty behavior, or malicious Intent will not be exposed [48].

Recent research has focused on ways to automatically extract particular types of app behavior, such as those that might violate user privacy or that may indicate the existence of malware or stealthy methods [21, 39]. Other research has looked at ways to measure the amount of trust one should have in an app (by rating developers, comments, etc.) [26]. But most of these approaches use limited information. For instance, there has been work on finding mismatches between the online descriptions with the requested permissions [7, 33]. While this provides some notion of intended vs. hidden behaviors, we have learned that neither information source gives complete and accurate information. Other research has focused on comparing descriptions vs. API calls, however there is often a lot of *dead code* in the form of API calls leading to a large overapproximation [27]. Other analyses remove advertisement or analytic libraries [34, 56], for simplification. We have learned that the external libraries often have their own behavior, and require their own permissions – and that they should not be ignored. Finally, there has been work on comparing user ratings, descriptions and permissions, but this is still only a partial view [37, 67]. While all of these approaches extract important behavioral information, and each provides a useful analysis technique, they often target specific data elements which may expose only a subset of app behavior.

Several researchers have started to analyze various app artifacts with focus on secu-

rity, quality and reliability. Qu et al. [55] examine the app's *Description* and *Permission* requests. Their analysis tries to determine whether text in the app description provides any indication for why the app needs a specific permission (they considered only 3 permissions: READ_CONTACTS, READ_CALENDAR, and RECORD_AUDIO). They selected from the top 500 apps specifically for each of the 3 permissions of interest, and analyzed each description with respect to only one permission for which the app was selected. In later work [63], they check for consistency between *Description* and *Permissions*, which they call description-to-permission fidelity. Their results show that the description-to-permissions fidelity is generally low on Google Play with only 9.1% of applications having permissions that can all be inferred from the descriptions. They hypothesize that new or individual developers—those not from a software company—may fail to completely discuss the need for some permission in the app description.

Gorla et al. [33] make use of app *Description* and *API usage*. Their goal is to identify suspicious/outlier apps in respect to API usage. They first examine the description text to cluster descriptions into topics, and then use API calls to reclassify apps as normal or abnormal depending on whether their API usage falls in "normal" behavior, where normal is defined by other apps in the cluster. They mention several weaknesses with the app artifacts in their dataset: code dominated with advertisement frameworks because they used free apps; use of certain words in the description text led to misclassification; analysis of code artifacts may be incomplete because of their inability to follow *reflective* calls and recover source code from obfuscated binaries.

Chia et al. [21] examine *User Ratings*, *Description*, and *Permissions* to determine that the current forms of community ratings used in app markets today are not reliable indicators of privacy risks of an app. Their large-scale study provided evidence indicating attempts to mislead or entice users into granting permissions. In terms of app artifacts, they conclude that popular and free apps request more permissions than they need. Hence, studies that rely only on such apps may unknowingly skew their analysis of permissions.

The work by Dini et al. [26] uses the largest variety of app artifacts to evaluate app "trust." They label an app as trusted, untrusted, or deceptive, based on a weighted formula that takes *Developer Rating*, *Number of Downloads*, *Market Name*, *User Rating*, *Number of Crashes*, and *Battery Consumption*. Because of their focus on trust, they ignore app functionality, and consequently code artifacts. Their weighted function gives high weightage to *Developer Rating* (to reward historically trusted developers), with the result that poor quality apps by a trusted developer may be ranked as trusted.

Other researchers have examined only code artifacts to assess security and trust. Huang et al. [39] compare text strings used in the user interface of an app with API calls to determine whether the API usage is consistent with user expectation. We have learned that developers may not always use the .xml files, and instead, choose to hardcode the strings into the Java source code, or use images for buttons. Moreover, they may not use all the strings in the .xml files.

Fuchs et al. [30] extract security specifications from manifests that accompany such applications, and check whether data flows through those applications are consistent with the specifications. Linares-Vásquez et al. [50] analyze how the fault- and change-proneness of APIs used by 7,097 (free) Android apps relates to its lack of success. They used the apps' average ratings as a measure of app success; the number of bug fixes in a particular

version as a measure of fault proneness; and the number of code changes at method level in a particular version as a measure of fault proneness.

Barrera et al. [16] analyze the Android permission model to investigate how it is used in practice and to determine its strengths and weaknesses, and usage patterns by apps. Peng et al. [57] use probabilistic models to assign each app a risk score based on its requested permissions.

Finally, some researchers have used non-code and non-developer artifacts to assess quality. Galvis et al. [31] process user comments to extract the main topics mentioned as well as some sentences representative of those topics. This information can be useful for requirements engineers to revise the requirements for next releases. Results show that the automatically extracted topics match the manually extracted ones, while also significantly decreasing the manual effort. Fu et al. [29] identify users' major concerns and preferences of different types of apps through user ratings and comments. Malmi [51] show how the quality of an app, as reflected in how people start to use it, is linked to the popularity of the app. They also show the connection between app popularity and the past popularity of other apps from the same publisher and find a small correlation between the two.

In our analysis, we ask what information should be utilized and pieced together to complete the puzzle of an apps true specification. We believe that this question is important, yet has not been previously answered. We present an in-depth case study on 14 highly downloaded apps from differing domains. We quantify the different sources of information that the developers use and look for inconsistencies between them.

Unlike traditional software applications that come bundled with extensive user guides, reference manuals, troubleshooting guides, etc., mobile apps are rarely supplied with such detailed artifacts. Instead, where they lack in providing extensive documentation and manuals, they attempt to make up for with a variety of concise artifacts, such as *Description*, *Reviews*, *What's New*, *Number of Installs*, *Platform Requirements*, *Content Rating*, *Permissions*, *Vendor Name*, *Category*, *Developer Contact*, *screenshots*, *Demonstrational Videos*, and *Size*. End users browse through these artifacts before purchasing an app from marketplaces such as Google Play, Amazon Appstore for Android, and the Apple App Store.

App code binary updates are frequent and are typically downloaded and installed automatically unless they require user consent or have been disabled. The app's corresponding artifacts may be updated in the marketplace, but this responsibility of keeping an app's artifacts consistent with each other as well as providing accurate information is distributed across multiple stakeholders, namely developers, users, and marketplace managers.

We analyze a subset of Android app artifacts to understand and compare what they tell us about how an app behaves. While some of these (rating, description, videos) are straightforward to understand, several others (permissions, Intents) require background in Android app structure, operation, and security features. We summarize some of the fundamental concepts needed to understand our study and results.

An end user downloads and installs an Android *.apk* file from a marketplace, which is actually a zip compression package containing several code artifacts needed to install and execute the app. Of interest to us are the *Classes.dex* file, *res/* folder, and *AndroidManifest.xml* file.

• *Classes.dex* is a java byte code file in Dalvik executable format, generated after compilation from java source, and used by the Dalvik virtual machine in Android. There are various tools

that can disassemble and decompile .dex files such as dexdump, dex2jar and apktool. These allow .dex files to be viewed on their byte- and source-code levels. For example, we obtain the following decompiled code segment for one app:

```
# virtual methods
.method public onClick(Landroid/view/View;)V
  .registers 6
  .parameter "v"
  .prologue
  .line 58
...
invoke-virtual {v0,v1,v2,v3},Landroid/hardware/Camera;->
takePicture(Landroid/hardware/Camera\$ShutterCallback;
Landroid/hardware/Camera\$PictureCallback;
Landroid/hardware/Camera\$PictureCallback;)V
  .line 59
  return-void
.end method
```

We show the recovered signature of the invocation of Camera.takePicture() and its invoking onClick() method. Such information may be mined to determine the resources (e.g., the Camera API) used by an app.

• The *res/* folder contains all the resources: an image resource, layout resources, launcher icons, and string resources. All this information may be used to better understand the app, e.g., by reading its strings, and examining its icons and GUI widgets.

• The *AndroidManifest.xml* file presents essential information about the app to the Android system. Of importance to us is it declares the permissions the app must have in order to access protected parts of the API and interact with other applications. For example, to take a picture, using the aforementioned *Camera.takePicture()* method, the app needs to declare the following permission in its AndroidManifest.xml file:

```
<uses-permission android:name="android.permission.CAMERA"/>
```

Such permissions are used by Google Play to inform end users about an app's access to resources and its capabilities.

As discussed above, an app may use a camera on an Android device by invoking the Camera API and adding a *uses-permission* block in the AndroidManifest.xml file. An alternative way is using the camera via an *Intent* (for our purpose, an Intent is an inter-app message) to invoke an existing Android camera app. A camera Intent makes a request to capture a picture or video clip through an existing camera app. The app using the Intent mechanism does not need to request permission in the AndroidManifest.xml file. The developer simply has to create an Intent object that requests an image or video, set a resulting image/video file location, and to "send" the Intent, using code much like the following:

```
// create Intent to take a picture and return control to the calling application
Intent intent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
// create a file to save the image
```

```
fileUri = getOutputMediaFileUri(MEDIA_TYPE_IMAGE);
// set the image file name
intent.putExtra(MediaStore.EXTRA_OUTPUT, fileUri);
// start the image capture Intent
startActivityForResult(intent, CAPTURE_IMAGE_ACTIVITY_REQUEST_CODE);
```

After the Intent starts, the Camera app user interface appears on the device screen and the user can take a picture or video. When the user finishes taking a picture or video (or cancels the operation), the system returns control to the calling app.

Although much of the work on understanding app artifacts is new, there is related work on the topic of general program understanding using reverse engineering [22], which identifies software artifacts in the subject system, and then aggregates these artifacts to form more abstract architectural models [52]. Some of this work has also been applied to specific domains such as the web [8] and service-oriented software [32] that are constantly expanding from simple systems (simple web-sites and message passing) toward the construction of full-fledged applications. A richer set of artifacts is available when understanding code by programmers for maintenance and evolution of large-scale code [68]. As end users provide online feedback (in the form of accessible reviews), these have also been mined and summarized [38]. Our own recent work on understanding GUI behaviors from test execution [9, 10] is rooted in test case repair and combinatorial coverage [23].

We complement prior work by providing an in-depth study that is conducted from a more holistic view. We identify the pieces of the puzzle that contribute to an apps overall behavior and security.

3.3 Role of Static Analysis

Static analysis concerns the inference of possible program behavior from the source code of the program. For each of the levels of logical inconsistencies, a security analyst needs to be able to connect sensitive operations to the GUI components that trigger them. Thus, the central role of static analysis in our project is this connection of sensitive operations to GUI components. Other tools in our project (described separately) leverage the static analysis information to interface with the security analyst (e.g., to flag potential semantic inconsistencies in button labels).

We specifically focused on three tasks:

- Task 1: Developing Static Analysis Techniques for Vetting Sensor Operations
- Task 2: Developing Static Analysis Techniques for Vetting Input Operations
- Task 3: Developing Static Analysis Techniques for Vetting Network Operations

We addressed these tasks together by considering the *sensitive* operation of interest (sensor, input, or network) as an input to the tool. We have thus focused our efforts on progressively improving our ability to give precise GUI component to sensitive operation information.

3.3.1 Static Analysis for GUI Logic

Android applications are executed in an event-driven environment governed by the Android framework. Apps are developed against the framework by extending special framework classes like `Activity` and `Service` and overriding pre-determined callback methods like `onCreate` and `onDestroy`. This model of development is quite different than standard Java applications, so there are a number of technical challenges that we have addressed to be able to perform static analysis of Android applications for GUI logic.

In a standard batch application, there is a single `main` method. In Android, however, the application implements a number of callback methods that are then invoked by the Android framework, so there is no single `main` method but rather a set of callback methods that we call *entry points*. Each entry point handles some *event*. From each entry point, we derive control-flow and call graphs, which are standard program representations. Thus, the baseline fact that we generate via static analysis is a *permission path*:

Definition 1 (Permission Path)**.** Given a sensitive method $C_n.sensitive$, a *permission path* is a path in the intra-event call graph from an entry point method $C_1.entry$. Schematically,

$$C_1.entry \rightarrow C_2.m_2 \rightarrow \cdots \rightarrow C_n.sensitive \ .$$

Or we may simply be interested in the permission pair

$$(C_1.entry, C_n.sensitive) \ .$$

False Positives and False Negatives Since sound static analysis is concerned with over-approximating program behavior, it is possible that some permission paths that our tool outputs does not correspond to actual behavior (i.e., is a *false positive* or a *false alarm*). Our approach is to examine successive approaches for refining our statically-derived information to successively eliminate false alarms. Our approach builds a semantic call graph using inclusion-based points-to analysis, so this initial phase already mitigates some sources of false alarms due to dead code.

While the static analysis approaches we apply are over-approximate, there are two main sources of *false negatives* or *missed alarms*. First, if the input app is a binary, then we apply existing tools for decompilation to Java bytecode. If these tools are unsound, then this may be a source of missed alarms. Second, the Android framework is complex and uses both reflection and native methods, which results in potentially missing behavior that could lead to missed alarms. We discuss in Section 3.3.3 our work in mitigating this source of missed alarms.

3.3.2 Permission-Sensitive Methods

From a usability perspective, it is simpler for the analyst to specify the Android permissions of interest (e.g., `android.permission.CAMERA`) than a sensitive method (e.g., `android.hardware.Camera: void <init>()`). We permit the analyst to instead specify a permission of interest by using a mapping from permissions to corresponding permission-sensitive methods. We obtain this mapping from the PScout tool [14].

3.3.3 Android Analysis Infrastructure

Our static analysis tools are built on the WALA [40] analysis framework for Java bytecode. We support analyzing applications from binary `apk` packages by using the third-party tools `apktool` [5] for extracting code from the package and `dex2jar` [6] for decompiling Dalvik bytecode to Java bytecode. It is known that `dex2jar` may not always produce sound Java bytecode [54], so issues in decompiling using `dex2jar` may result in missed alarms.

The second issue with respect to missed alarms mentioned above concerns static analysis of the Android framework. The Android framework makes heavy use of reflection. Reflection is a notoriously difficult issue for static analysis because of the need to model that reflection—otherwise code invoked via reflection is (unsoundly) seen as unreachable. In particular, application entry points are invoked by the framework using reflection, and thus, without special handling of reflection, none of the application code would be seen as reachable. In order to perform GUI logic analysis, we needed to invest effort in this problem, which resulted in the creation of Droidel [19]. Droidel has been open sourced and has already been adopted by other researchers at IBM Research and the University of Texas at Austin.

Prior approaches to Android static analysis have addressed this problem by developing models of the Android framework to serve as a replacement for the Android code itself. Because modeling is labor intensive, almost all models are client-specific, meaning the model only aspects of interest to the client analysis (e.g., taint flow for taint analysis). Thus, we were unable to reuse prior models for our GUI logic analysis and thus needed to create Droidel. Droidel is different from prior models in that it aims to be a general approach to the framework modeling issue—at least with respect to resolving reflective calls in Android. We compared Droidel with the model from an existing state-of-the-art Android analysis tool on a set of seven open source apps and observed significantly fewer missed methods in call graph construction (see [19] for details).

Droidel also provides infrastructure for working with Android GUI components (called `Views`), including reasoning about layout inflation and manifest-registered callbacks.

3.3.4 Precision Improvements

In the remainder of this section, we discussed static analysis approaches that we have taken to improve the quality of the permission path results we produce, particularly with respect to false alarms.

3.3.5 View Refinements

Given a permission path to a GUI component handler entry point (e.g., `onClick`), it is non-trivial to identify the corresponding GUI component. Specifically for example, an `onClick` entry point is a method on an app class that implements a `View.OnClickListener` interface with the following signature:

```
public void onClick(View v);
```

When this callback is invoked, the GUI component is passed as the `v` parameter. For GUI logic analysis, we need to be able to determine what are the possible `Views` on which this

`onClick` may be invoked. Unfortunately, the baseline information via a point-to analysis is typically extremely imprecise, yielding essentially all `Views` of the application.

To obtain more precise information about GUI components and associated callbacks, we apply further static analysis to try to determine which `Views` can have which listeners registered. Obtain such information in a precise manner is challenging because a listener can be attached and detached from a `View` programmatically in addition to manifest declarations. In on-going work, we address this problem by applying backwards symbolic reasoning techniques [17, 18]

3.3.6 Activity Transitions

In addition to GUI component handler entry points, another kind of entry point is a *lifecycle* callback. A lifecycle callback is an app-implemented method on a core Android component, such as `Activity.onCreate`. An `Activity` is an Android component that corresponds roughly to a screen, and an Android app can consist of multiple `Activities`.

A permission path that begins from, for example, an `Activity` lifecycle entry point like

$$A_1.\texttt{onCreate}$$

is not directly associated with a GUI component. However, a common idiom is that the triggering of GUI component from another `Activity` A_2 causes the creation of A_1. Thus to provide more precise permission path information, we need to relate the creation of `Activities`. To do so, we apply a separate static analysis to generate such relations in the form of an activity-transition graph, which was defined by prior work in the literature [15].

4 RESULTS AND DISCUSSION

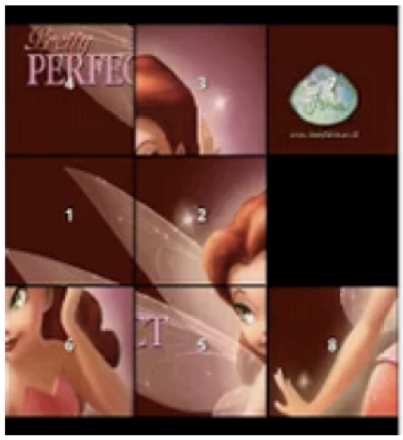
4.1 Findings from Descriptor-Based Analysis

To evaluate our descriptor based approach, we collected our own corpus of Android apps. This corpus consists of 178,765 apps published on the Google Play Store. We extracted the public features of all 178,765 apps to perform our analysis. Among these apps, 153,294 were free. We took a sample of 84,405 apps (about 50%). We downloaded, unpacked, disassembled, and decompiled them. Each app yielded about 1,000 program files we must process. From these files, we extracted user interface and code features. We stored all extracted features using MongoDB, a NoSQL database optimized for big data analytic. The total size of raw data is about 4TB.

4.1.1 Camera Permission Inconsistencies

In our corpus of 178,765 apps, we found 17,739 (9.9%) apps requiring camera permissions. At the public level, we used all these apps as positive examples. We randomly selected the same number of apps that do not require camera permissions as negative examples. We trained a model for positive camera permission apps based on Maximum entropy. The training accuracy was 98.7%. We then applied the classifier to the 17,739 apps that require camera

permissions. Among these apps, 307 were classified as not requiring camera permissions. These apps present inconsistency that could not fit the model. At the user interface level, we analyzed a subset of 7,816 apps requiring camera permissions. We repeated a similar training process. The training accuracy was 92.9%. 498 were classified as inconsistent. We now discuss three types of camera inconsistencies we discovered.



TinkerBell Puzzle

Public:
Fun & Addicting Puzzle Game Create Puzzles from TinkerBell Photos Challenge & Compete with your friends ...

User Interface:
Tinkerbell Puzzle All photos are collected from the search engines ... Take Picture ... New Game High Scores Settings New Best Time ...

Figure 1: Example of Inconsistency between Public and Interface

Inconsistency between Public (P) and Interface (I): An app is inconsistent between its public information and interface if the interface presents certain sensitive features that are not disclosed on the app's public page in the app market. TinkerBell Puzzle (Figure 1) is an example we discovered that exhibits this type of inconsistency. It is a puzzle game. The app's public description mentions "photos" but it does not suggest the photo is being taken or shot by cameras. At the interface level, the combination of the word "photos" and the phrase "Take a Picture" provides strong evidence to the camera use. In this case, users may be confused since the app's description does not clearly describe the camera feature. Consequently, they may avoid installing the app to explore the UI.

Animals Game for Kids

User Interface:
Balloon Animals for Kids Menu button to resume NEW GAME RESUME G ... (71 words)

Code:
org/anddev/andengine/engine/camera/Camera;->onUpdate
org/anddev/andengine/engine/camera/Camera;-><init>

Figure 2: Example of Inconsistency between Interface and Code

Inconsistency between Interface (U) and Code (C): An app is inconsistent between its user interface and code if the label of an interface component (e.g., "New") does not match the code this component invokes (e.g., takePicture). Animals Game for Kids (Figure 2) is an example we discovered that exhibits this type of inconsistency. It is a game for kids. There is no indication on the user interface that the camera is used. But at the code level, camera API calls are found. There is no logical connection between any interface component and these API calls. In this case, users would be confused because it is not clear what UI component triggers the camera function.

Inconsistency between Public (P) and Code (C): An app is inconsistent between the public and the code levels when there is a mismatch between its public description and its actual behavior as revealed by code. Figure 3 shows an actual example of this type of inconsistency we discovered. This app appears to be a glossary app. The app's code contains calls to take pictures. A user would be very surprised when a picture is taken while he/she is using the app to look up math terms.

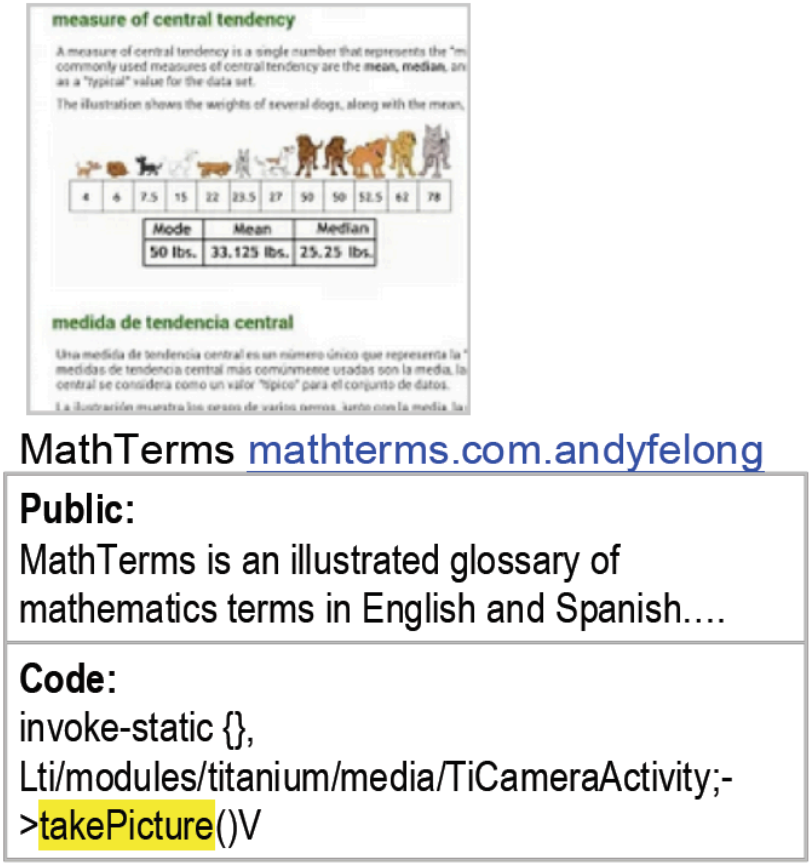


Figure 3: Example of Inconsistency between Public and Code

4.1.2 Similarity Inconsistencies

A classic example of similarity inconsistency is when two apps appear to be different at the public level but have very similar user interfaces. We further analyze our dataset to identify this type of inconsistency. We used the classic term frequency-inverse document frequency (tf-idf) model to calculate the similarity among apps. We used 95% as the threshold. If two apps are more than 95% similar, they are considered as near identical. We treat them as inconsistent apps because most apps have no near identical apps. At the public level, we analyzed 11,880 apps and focused on just the app's description. We found 630 apps (5.3%) with near identical apps in terms of their description. Let's denote the set of these apps as P. At the interface level, we analyzed a sample of 72,993 apps and found 1,339 apps (1.83%) with near identical apps. Let's denote this set as I. Having found P and I, we can compute the difference between the two, which will tell us which apps are similar at one level but not at the other.

As an example, NetCounter and Network Traffic Monitor are a pair of apps we discovered that exhibit similarity not at the public level and only at the user interface level. By reading the public information, users would see them described differently, made by separate creators, and filed under distinct categories. But our UI similarity analysis reveals that the interfaces of the two apps are almost identical, suggesting that one app may be a knockoff of the other. An unsuspecting user would have no way to tell until after installing the app. A more cautious user may compare the two apps and notice that one requires as many as 28 permissions while the other requires only five. Also, one has more than 20K user ratings with an average of 4.5 star while the other has only 177 ratings with an average of 2.5 stars. Based on this comparison, one may deduce that the one that receives weaker ratings and requires more permissions is probably a knockoff. Unfortunately, Google Play's similarity calculation does not take into account the user interface or the code. These two apps are not listed as similar apps to allow users to do such comparison.

4.2 Findings of App Artifact Synthesis

In our evaluation of app synthesis, we ask three research questions that aim to find out what types of information are required to piece together an app's true specification. Our first question looks at the descriptive artifacts provided by the developer. The second question examines artifacts that provide indirect information. Our last research question evaluates information provided by stakeholders besides the developer. The questions are:
RQ1: Which artifacts provide descriptive information of an app's behavior and are these consistent across artifacts?
RQ2: What information can be inferred indirectly from other artifacts?
RQ3: What type of specification information and app behaviors are not controlled by the developer?

4.2.1 Selecting the Apps

We performed in-depth manual analysis on 14 free apps selected from the Google Play store. We chose apps that span categories and size (as determined by apk size). Table 1 shows our sample in more detail. The apps' sizes range from 806KB to 49MB and are distributed across 10 different categories. We only selected apps which were expected to be of average to high quality. Such app's we assumed would be better documented and contain more consistent artifacts. Both of these properties should facilitate our goal of spec construction. The ratings range from 3.6 to 4.6 on a 5.0 rating scale and include several developers with Top Developer badges.

4.2.2 Preparing the Apps for Evaluation

To prepare the apps for manual evaluation, we began by collecting information from each app's Google Play store page as well as downloading the respective .apk file. We then performed the following:
(1) From each app's Google Play page, we scraped the page for developer rating, user rating, user comments, number of installs, screenshots, video (if uploaded), permission request text,

Table 1: Apps in Case Study

App Name	Category	Developer Rating	APK Size listed	actual
HealthTap (HT)	Health/Fitness	Top	47M	49.5M
Android Auto (AA)	Transportation	Top	27M	27.9M
Nest (NST)	Lifestyle	Standard	24M	25.3M
Kids Puzzle: Vehicles (KPV)	Educational	Standard	21M	20.7M
Peel Smart Remote (PSR)	Entertainment	Standard	17M	17.5M
Diabetes Logbook by mySugr (DL)	Medical	Standard	14M	14.9M
Timely Alarm Clock (TAC)	Lifestyle	Top	9.4M	9.9M
HiFont - Cool Font Text Free (HF)	Personalization	Standard	6.1M	6.4M
Paint for Kids (PFK)	Casual	Standard	6.8M	6.3M
Water Your Body (WB)	Health/Fitness	Standard	5.7M	6M
Spell Checker - Spelling boost (SP)	Books/Reference	Standard	2.9M	3M
IDO Calculators Plus Free (ICP)	Tools	Standard	2.7M	2.7M
Pedometer (PED)	Health/Fitness	Standard	1.9M	2M
Barcode Scanner (BS)	Shopping	Standard	Varies	806K

listed interactive elements text, whether the app contained in-app purchases, and the text in the app's description.

(2) From each app's .apk file, we obtained source code files by disassembling the .apk file into *smali* byte code files with apktool [5]. We also decompiled the .apk file with dex2jar [6] to obtain the app's Java source code files.

This resulted in a collection of quantitative, categorical, text, and image based information on the app that guided the evaluations discussed in Section 4.2.8.

4.2.3 Evaluation Methods

Additional preparation was required for each app's text, video and image-based artifacts. This included parsing the app's Manifest file for permission and library usage, parsing code files for API calls and Intents, as well as manual analysis. Manual analysis required the evaluator to watch a video, read text or view a collection of images. After viewing, reading or watching, the evaluator then produced a set of behaviors that the non-code related video, text, or images described. Behaviors were defined as a higher-level action the app could perform that required permission. To address ambiguity within the artifacts and implementation, we listed all possible related behaviors. For example, a description may list "Share with friends" as a feature. This would result in messaging and social media behaviors. Images often contain many behaviors in a single image. For example, suppose a screenshot shows the device with a map and consequently shows the UI which contains a download and sync button. Such a screenshot would result in maps, navigation, download, sync, location behaviors. The reason for including these clusters was to ensure that the inconsistencies found were less a matter of evaluator opinion.

To determine incompleteness of descriptions against permissions, we took each descrip-

tion's set of behaviors and converted them into the permissions they require. This conversion was done by referencing Android permission documentation as well as PScout's permissions mappings [13]. We then went through the app's requested permissions. First, we looked to see if the permission was accounted for in the desc's set. If there was a permission unaccounted for in the description that was requested by the app then the descriptions was marked incomplete. If all permissions were accounted for by the description, we then evaluated the app's behavior by comparing its higher level features with its screenshots, videos and our own use of the app (i.e. we ran the app on a physical device). If the app's behaviors contained in the description were in the app when we used it, we called this app description complete. If there were features in the description not in the app, it was labeled contradictory. The only app we could not run (due to compatibility issues) was *Android Auto*, but the app was already inconsistent and did not require such analysis.

In order to determine each app's use of advertisement (ad) and analytic libraries, we evaluated the app's source code files. Specifically, an app was said to use ads and analytics if their disassembled code contained both of the following: (1) the app contained a key for the package in its Android Manifest file and (2) the library's code was included in the app's disassembled byte code. We then obtained each of the library's use of sensitive methods by string searching through the app's code for API calls found by PScout [13], a tool that extracts the permission specification from Android source code using static analysis, for each app's Android build and parsing the results for the identified library-related class paths.

To evaluate an app's use of non-permission protected Intents, we consulted Android documentation. Android lists common Intents and describes several which do not require permissions. While there are many Intents, we focused our studies on those related to otherwise permission-sensitive behaviors—specifically those related to SMS, contacts, internet, phone, calendar and camera/video—in order to see where permissions were lacking information about program behavior. Note we did not account for all permission-sensitive behaviors, such as those that open a user's settings, although the addition of these could only keep or enhance the findings we present.

In order to evaluate user comments, we obtained the first 20 comments on the app's Play Store page ranked by newest to oldest and filtered for all versions. This is in contrast to filtering the comments for those only on the latest versions – another option provided by the Play Store. The resulting comments were then combined with the first 20 returned comments ranked from most helpful to least helpful on any version of the app. Helpfulness is determined by fellow users who can rate the comments on the page. Duplicate comments were removed and the resulting set consisted of both the most recent ratings as well as those that have been voted as providing the most useful information about the app's functionality. When evaluating the comments, a comment was considered negative if it indicated either a functionality problem or an indication that the app should undergo changes. This included comments reporting crashes, bugs, freezing, hanging, missing features, and ads.

4.2.4 Threats to Validity

This study has some potential threats to validity that we outline here.

Threats to External Validity. First with respect to external validity, we used only 14 apps and we do not span all categories (we cover 10 of 26). We also do not evaluate only

the highest rated apps such as those with 4.6 or above, but instead chose to use a variety of ratings. Given that this is meant as a case study and we are not necessarily generalizing, we believe that this is valid.

Threats to Internal Validity. With respect to internal validity, the behaviors we extract may be subjective and require extensive background knowledge of Android's API and framework. Consistency and incompleteness suffer from the same problem. We erred on the side of caution and always gave the benefit of doubt for consistency when a case was too close to determine. This means that our results should be conservative.

Another internal threat is that the manual analysis was performed by us. The manual work might also be subject to mistakes, but we double checked our results to improve our confidence in their correctness.

The use of the apps and their documentation were limited to the free portion of the application (features or descriptions that are provided with the paid version are missed). Since we compared artifacts all within the free domain, we believe our results are at least consistent. Additionally, some apps described pro and paid versions on these pages as well as free in their descriptions. Further, each artifact is subject to threats to validity associated with human actions (e.g, any errors the user may have made when filling out the forms). These may contribute to the study's observed contradictions and inconsistencies. Finally, one app, Android Auto, could not be used manually due to compatibility issues, therefore for that app we were restricted to static analysis.

Threats to Construct Validity. With respect to construct validity, we could have chosen different artifacts or measures, but we based our analysis on other types of studies that have been performed before on Android.

We now look at the results for each of our three research questions. We then discuss some observations in more detail and summarize what we have learned from this study.

4.2.5 RQ1: Descriptive Artifacts

To answer RQ1 we examined the data that developers explicitly provide on the Google Play store. When a developer publishes their app on the Play store they are required to give a description with a 4000 character limit. This description then appears on the app's Play store page, which in turn can be seen by users to help them determine whether or not to install the app.

As we read each of the app's descriptions, we noticed the apps' developers were using this space in different ways. Not only were there varying lengths ranging from as few as 57 words to 622 words, but there were differing uses of the descriptions. Some developers were explicitly stating the app's use of permissions (or providing a link to a site which did so), many others included a bulleted list of the app's features, while others focused less on its complete behavior, and instead focused on its benefits, how it could be used (in a generic way), or just quoted positive reviews.

Table 2 divides our apps' descriptions by the type and amount of information present in the description. The first column (len) shows the variation in user description length. The second (Exp) shows whether the permissions were explicit or contained in a link, and the third (List) whether a list of features was provided. In addition, Table 2 shows whether the description provides extra (non-behavioral) information (Other) such as the benefits of

downloading the app. This text does not provide specifications and may actually hurt an NLP analysis. Finally, we show whether descriptions offered incomplete or contradictory information with respect to the app's behavior as determined by the evaluation methods discussed earlier.

As can be seen from Table 2, 7 of the 14 apps had incomplete descriptions and one contained a contradictory statement. The contradictory statement in *IDO Calculator Plus*'s description said that the app did not contain ads when in fact it did. Interestingly, the apps which listed features and stated the permissions explicitly weren't necessarily immune from being incomplete.

During our evaluations, we noticed that other artifacts on the app's Play store page disclosed behaviors that the descriptions had excluded, and as a result, realized that looking at the description alone may be too limited of a scope. Specifically, *developers were using assets such as videos, screenshots and the* What's New *section to provide relevant spec information not present in the description.* The most blatant example of this was in Google's *Android Auto*. The description contains only 57 words and states the app "brings familiar apps and services to a car". However, it provides a video which demos the app's use, and screenshots which shows the app's UI and core capabilities. This shows the user the UI as well as functionalities like the use of maps and navigation, SMS, phones calls, voice commands and scheduling reminders and events. Further, in the app's *What's New* section, it includes 74 words explaining the features (16 more words than its description). The new features listed included all of the behaviors shown in the video and screenshots, which were missing from its description (navigation, SMS sending, scheduling & notifications, phone calls, and voice commands). Therefore, while *Android Auto*'s description provided little information on the app's behavior, other accessible information on the app's page could impact their expectations. One thing that we noticed is that some of the videos and other artifacts were in languages other than English, which will complicate an analysis.

The apps we studied used on average 9.9 screenshots and nearly half of them included a video that was on average 2 minutes long. The use of the *What's New* section varied in our apps. Some developers simply listed "fixed bugs", while others offered long explanations of new features like *Android Auto*. Most importantly, we identified approximately 8 behaviors that were not included in the apps' descriptions that were contained in the videos and screenshots. In summary, nearly half of our apps' descriptions did not exhaustively and explicitly explain their app's behavior, and we identified alternative artifacts within an app's Play store page where developers disclose spec related information on the app's behavior.
Summary of RQ1: *Play store app descriptions are used in varying ways. Most of the apps studied had incomplete descriptions and one had contradictory information. Some developers opt to use alternative assets like images, videos, or the What's New field to reveal additional behavior.*

4.2.6 RQ2: What Types of Inferred Information Exists

To answer this research question we examine types of information that are provided by alternative artifacts (not explicitly provided). For this we use the AndroidManifest.xml file and the user interface (UI) itself. We begin with the manifest file. Since prior studies show that the manifest file often over requests permissions, we were surprised to find several

Table 2: Description Lengths & Usage. "I" are incomplete descriptions. "C" are contradictory descriptions.

App	Len	Exp.	List	Other	I	C
AA	57			✓	✓	
BS	184	✓	✓	✓		
DL	243		✓	✓	✓	
HF	197		✓	✓	✓	
HT	622	✓	✓	✓	✓	
ICP	184		✓	✓		✓
KPV	184		✓	✓	✓	
NST	104	✓		✓		
PED	251			✓		
PFK	105		✓	✓	✓	
PSR	317		✓	✓		
SC	197			✓		
TAC	257		✓	✓		
WB	198		✓	✓	✓	
Totals		3(20%)	9(64%)	14(100%)	7(50%)	1(7.14%)

Table 3: Permission Requests, Intents and API calls.

App App	# of Non-permission protected Intents	App App	# of Non-permission protected Intents
BS	3	PFK	1
AA	1	SP	1
DL	1	WB	1
HF	1	ICP	0
HT	1	PED	0
KPV	1	PSR	0
NST	1	TAC	0

cases where the opposite was true, and the permissions lacked important information on the app's behavior. Specifically, we observed apps that were using Intents to perform significant actions without the need to request permission. These Intents are not those defined in the manifest file, but could be found by searching the app's decompiled java files. We provide some examples here.

Paint for Kids. *Paint for Kids* is a kid's app that provides pages for children to color. It has a user rating of 3.6 and approximately 1,000,000-5,000,000 installations. Its description says it can color, save and share pictures as well as take pictures which can be turned into coloring pages for use in the app. The Play store page contains a video which demos the app and shows the app taking a picture and then turning it into a coloring page. Screenshots show painted pictures as well as buttons with icons for saving, sharing, painting and taking pictures. Despite all indications of camera usage, however, when a user asks to install the app, no camera permission is requested and no permission request appears in the app's manifest file. When the app is used, the camera is indeed opened by the camera button, and everything else proceeds as demoed in the developer's video.

What allows *Paint for Kids* to use the camera without a user's permission is that the app is invoking the camera via a non-permission protected Intent. This Intent allows the app to employ another app on the device to take and save the picture. *Paint for Kids* then has access to the saved image and can do what it likes with the file. Intents exist for other typically sensitive operations such as creating and writing SMS texts, editing contacts, dialing phone numbers and capturing video [1]. An app (possibly malicious) may circumvent the permissions by accessing these sensitive operations via Intents.

Table 3 shows the number of such Intents not accounted for by an app's permission list, along with the number of permissions the app does request. 10 of the 14 apps used this mechanism to access protected resources. Surprisingly, *Paint for Kids* was not the only app aimed at children with such behavior. *Kid Puzzle-Vehicles* included an ad-related class that dialed phone numbers. Although the phone itself is not invoked, the dial pad will pop up with a number, and a child can then easily press the call button to execute the call.

The second artifact we used for inference was the UI since it offers additional information about what an application does and impacts the user's expectation. For example, a user might see an icon of an envelope and will expect the app to be associated with some sort of messaging or email behavior. Several studies have identified the UI as a source of valuable information for program understanding [39], but most evaluate the app's strings statically and leave the icons as future work. As a result we were interested in evaluating how an app's strings and icons interact with respect to what unique information they each provide. In short, do our app's icons provide information that is not available in the app's strings?

To answer this question, we manually evaluated all of the app's strings defined in values and values-en as well as all included .png files in drawables, drawables-nopi and drawables-hdpi that were icons (i.e., not images or graphics used for content or background purposes) and listed the associated behaviors for each string and icon. Because app's often include the same icon for different states (such as pressed and active), the unique images within the drawables was typically much less then the number of included .png files. During our analysis we failed to find a icon-related .png that did not contain a defined string that defined it. That said, the set of defined strings in our apps greatly outnumbered the defined icons so this may be a superset of information.

In summary, the collective set of icon-related .pngs in drawables, drawables-nopi, and drawables-hddpi was contained in the combined set of strings in values and values-en. What remains unclear and may be a subject for future work is either set's relationship to our app's actual UI and behavior– whether the icons are a more concise representation of the UI or whether the strings define too many false values not used by the app.

Summary of RQ2. *We conclude additional (and possibly differing) information for an app's spec is indirectly available through the permission set, non-permission protected Intents, and/or the app's UI as represented by sets of icons and strings. Our research suggests that the icons an app defines have significant overlap with its strings.*

4.2.7 RQ3: What Externally Controlled Information is Available?

To answer the last research question, we turn to three pieces of information that are not explicitly provided by the developer – advertisements, user ratings and user comments.

Advertisements. Little information on advertisements (ads), analytics and social media libraries was contained on the app's Play store page, yet these libraries present obstacles in specification construction at the code level. First, they introduce extra code in the source code files, icons, and strings which may potentially cause a specification to include behaviors an app doesn't actually perform. Second, they often employ reflection causing an app's API calls to be incomplete [34], and as confirmed in our case studies, little information about them can be obtained from the Play store – apps tend not to be upfront about their usage. Last, they can be dynamically added and removed (e.g., the app might cycle through or change their advertisers on a regular basis, and the ads themselves, may evolve on a cycle that differs from the app evolution). In fact, some other studies have chosen to remove ads from their analyses [34, 56]. Since our case studies were all free apps, the use of ads, analytics and social media service libraries may not be surprising. We observed on average 2 such libraries in each app with one app containing as many as 5 and another not containing any. It is relatively easy to determine if an app includes a library based on the source code, but it is difficult to determine what aspects it uses due to the presence of dead code and use of reflection. Two cases that we present next help us show some behaviors found in our app's code that appeared related to the libraries.

Kids Puzzle - Vehicles. *Kids Puzzle - Vehicles* is a children's game app that also teaches the user different languages. Its description describes the different levels and characters of the game as well as the benefits of learning a new language. Its Play store page confirms these behaviors and includes screenshots of the different games children can play as well as different languages to use while playing. It has over 50,000 downloads and a user rating of 3.9. Five of the 28 comments evaluated reported the app hanging, crashing, or freezing and one of these comments reported crashing because of ads. This comment was the only mention of ads on the page. The app requested 12 permissions and included several which were not described by its description, UI or comments. These included READ_LOGS and MOUNT_UNMOUNT_FILESYSTEM. While evaluating the byte code further, we found that the app included the `Umeng` ad library, which has been reported as harmful adware [4]. No mention of `Umeng` were in the description and the app did not include an interactive elements section. `Umeng` could be detected when inspecting the resource files for the app. Execution of the app also revealed ads, which were not mentioned in the app description.

Table 4: Negative Comments, User Ratings & Quality. "TD" indicates whether the app was developed by a Top Developer. "DR" is whether the number of times the developer responded to negative comments. "CR" is the number of comments that indicated crashing or freezing. "LH" is whether the app had the `<:largeHeap>` attribute set to true. This determines whether the app's processes are created with a large Dalvik heap and is discouraged by Android.

App	Installs	TD	User Rating	Size	Neg. Comments	Repeating Issues	DR	CR	Perm	Lib	LH
AA	>10K	✓	3.7	27M	28:38 (73.6%)	Compatibility, Night Mode Needed, Car Display Requirements, Limited App Support	0	1	16	0	
PSR	>10M		4.2	17.5M	20:29 (68.9%)	Crashes, Issues w/ Update, no guide available, button functionality issues	1	3	13	0	
NST	>500K		4.4	25.3M	25:38 (65.7%)	Offline, Auto-Sched. & Geofencing Doesn't Work	0	1	12	2	
TAC	>5M	✓	4.4	9.9M	25:40 (62.5%)	Can't Dismiss Alarm, Alarm Doesn't Go Off When Phone is Silenced, Dislike of Notifications for Upcoming Alarms, Requests for Skip Next Features, Wrong Time	0	1	14	1	
BS	>100M		4.1	806K	21:40 (52.5%)	Slow to Scan, Won't Scan	2	0	9	0	
SC	>1M		3.9	3M	16:32 (50%)	Ads, Crashes, Trouble w/ Foreign Languages, Doesn't Work, Char. Limit	1	3	4	4	
HT	>1M	✓	4.4	47M	12:29 (41.3%)	Wrongly Charged For Services, Freezing	8	2	22	2	✓
PFK	>1M		3.6	6.4M	11:27 (40.7%)	Ads, Slow to Download	0	1	4	1	✓
DL	>1K		4.6	14.9M	10:38 (26.3%)	Include Carb Search, Too Expensive to Upgrade	1	0	14	2	✓
PED	>1M		4.2	2M	9:38 (23.6%)	Inaccurate Step Counting, Trouble Resetting Session	0	0	4	2	
KPV	>100K		3.9	20.7M	5:28 (17.8%)		0	3	12	2	
HF	>10M		4.2	6.4M	7:40 (17.5%)	Trouble Installing Fonts	0	0	19	5	
ICP	>500K		4.1	2.7M	4:29 (13.7%)	Permanent Ads in Notification Bar	0	0	10	2	
WB	>5M		4.5	6M	4:37 (10.8%)	Ads		0	11	1	

HealthTap. *HealthTap* is a health related app that allows a user to search and consult with doctors via text, video and phone as well as send them medical documents and images. It's developed by a Top Developer, has 1,000,0000-5,000,000 installs and a rating of 4.4. It requests 22 permissions including those related to location, camera, recording audio, accounts, contacts, and internet. While their app requires a lot of permissions, each of these could be reasoned from the description, which is over 600 words long. In the description, videos and screenshots, no mention of ads or social media services appears, however they were included in the app's code and resources as well as the Android Manifest file where their keys and meta data were included. When viewing the page, a viewer may infer the app used location to personalize feeds as well as facilitate other features like searching for doctors and prescription delivery, which were features of the app. The byte code indicated other undisclosed uses such as tracking the user to send localized marketing via push notifications.

In summary, we saw that apps only occasionally mention ad libraries and analytics to users in the app's descriptions and never showed ads or analytics in screenshots or videos. Furthermore, when ads were mentioned they were typically inferred when the description mentioned that the paid version contained no ads or in user comments complaining about them. The store's new interactive elements category [2] (described in Table 5) may help with this, although it is still new and not all developer's have updated their pages to include this information. For the purposes of constructing the app's specification, this category will not solve all the presented issues. These two examples help demonstrate that while the use of ad and analytics libraries may be expected in free apps, their specific behaviors are important to include.

User Provided Information. Every app's Google Play store page contains several user-controlled categories that display information submitted or defined by the app's users. These categories include developer ratings, number of installs, user comments and user ratings. We included these artifacts in our evaluation to assess what information they could provide and to determine whether they could be used as indicators of the app's general quality.

Table 4 presents these artifacts and other measures of quality for each app in our study. We were surprised to see the apps that had reports of major functionality issues– such as reported crashes, freezing, and buttons not working– continued to be downloaded by many users and receive high user ratings. Two examples are discussed next.

Peel Smart Remote. *Peel Smart Remote* is a remote control which enables a user's Android device to control his or her television, Apple TV, etc. The app's description explains that *Peel* can personalize viewing, set reminders and change and customize channels. When evaluated against the app's permissions, the artifacts appeared relatively consistent. Despite its overall consistency, user comments suggest the app may functionally be sub-optimal. At the time of the study, 3 out of its then 20 latest reviews report crashing and 4 additional comments reported freezing and dysfunctional button usage. Despite these complaints, the app holds a 4.4 rating on a 5.0 scale.

Timely Alarm Clock. *Timely Alarm Clock* is an app that allows you to synchronize alarms across multiple devices. Its description describes the app's ability to sync devices and customizable interface, and its UI is straightforward with clock-related icons and strings. Its video highlights the app's customizable UI and alarms as well as its ability to sync with devices, work as a timer, and adjust audio with different user gestures. Like *Peel*, *Timely Alarm Clock* received many complaints. Seven of the last 20 comments at the time of the

study reported functionality issues such as not being able to turn off the alarm, the set alarm not going off or going off late, and its time lagging behind the system clock. In the combined set of helpful and new user comments spanning all versions, 25 out of 40 were negative. Nonetheless, *Timely* maintains a very high user rating of 4.4 as well as a Top Developer badge.

Such contradictions were witnessed in several other apps as well. Table 4 lists the apps by decreasing percentage of negative comments received. Apps with 5 million downloads and user ratings of 4.4 received a high percentage of negative comments. Furthermore, as indicated by the location of the check marks in the table, developers with Top Developer badges appeared to receive just as many negative comments as non-top developers, with some receiving as many as 60-70% negative comments. Such results suggests that either user ratings and developer ratings were not indicators of quality in our study or perhaps the sample of user comments were biased towards negative reviews.

We did identify one potential reason for this discrepancy, which might help explain *Peel Smart Remote*. User ratings are cumulative and include ratings from older versions of the application, while user comments can be sorted by decreasing variables (such as date, helpfulness and rating) and filtered by the latest or by all versions. When evaluating the different filters, the returned results tended to favor more recently submitted comments even when sorted by helpfulness. As a result, for some apps, this inconsistency between ratings and comments may be explained by a high user rating reflecting a history of high-quality and negative comments which may reflect a poor release or update. In essence we are comparing data from different points in time and this could be problematic for a user.

In conclusion, we saw inconsistencies between user approval indicators such as user ratings and installs and app quality indicators such as developer ratings and user comments. We also saw inconsistency between developer ratings and user comments. These may be explained by the user approval being unrelated to quality, or the user comments containing a bias in either how they are ranked and displayed by the Play store, or how users use them (such as dissatisfied users are more likely to submit a comment then satisfied users). Either way, the app's within our study with high user ratings or a Top Developer badge did not appear to indicate that the latest version available on the store was not buggy.

Summary of RQ3. *We see that there is dynamically added behavior (in the form of ads) that should be accounted for if we want to understand an app's true behavior. This is usually not discussed by the developer and may not be known. We also see that user approval and quality measures such as number of downloads, user & developer ratings and user comments can be utilized, but they were contradictory. Such contradictions may be a result of versioning or suggest that comments and ratings can not be compared.*

4.2.8 Summary Discussion

We now synthesize what we have learned from this study. Table 5 shows a summary of the artifacts we have studied. For each it shows whether they are located on the Play store or in an apk file, their visibility to the user, what information they contain, whether they provide behavioral information, and an overview of their potential source of inaccuracy when assembling an app's specification. Together these pieces should be combined to understand true specifications. As can be seen, many new analyses are needed to extract and merge this

Table 5: Synthesized Android Artifacts Used in our Study. In the column "Creator", "D" is for developer, "U" users, and "M" the Google Play store market. In the column "Type", "N" is for numerical, "C" categorical, "T" textual, and "P" pictorial.

	Artifact	Creator	Type	Potential Information	Behavioral	Source of Inaccuracy	Previous Work
Google Play store (visible to user)	Description	D	T	App features and behaviors.	✓	May be incomplete. Developers could use other elements of the page instead.	[35, 36, 39, 40]
	What's New	D	T	Features added to the latest updates.	✓	May be incomplete. Developers could use other elements of the page instead. May only include changes, not overall app behavior.	
	Screenshots	D	P	Visualization of app behavior and design.	✓	May be for an older version. May be absent from the page.	
	Videos	D	P	Real-time behaviors and design.	✓	May be for an older version. May be absent from the page.	
	Interactive Elements	D	C	Whether the app does the following: shares personal information with third-parties, shares location, allows purchases, contains user-provided uncensored content and/or has unrestricted internet access.	✓	May not be included by developer.	
	User Comments	U	T	Functionality, user approval, and behaviors	✓	May include previous versions. May be device specific.	[25, 26, 27, 45, 46, 53]
	User Ratings	U	N	User approval and quality.		May not reflect the latest version.	[34, 42, 46]
	Developer Rating	M	C	An indicator of developer trust and app quality.		May not reflect the Developer's latest version or project.	[34]
	Installs	U	N	The number of times the app has been downloaded by users.		Does not reflect number of uninstalls after install.	[34]
	Permissions (as listed on the Play Store)	D	C	The sensitive operations an app can perform.	✓	Displays certain categories of permissions only, so list may not be an exhaustive. May include permissions not used by the app. May miss behaviors performed via non-protected intents.	[14, 29, 31, 33, 35, 36, 37, 39, 40, 43, 44, 59]
.apk file	Strings (in strings.xml)	D	T	Text that may appear in an app's UI.	✓	May not be used at runtime. May may be an incomplete set because of hard-coding.	[33,36]
	Images (in /res/drawables)	D	P	Images that may appear in the app's UI.	✓	May not be used at runtime.	
	Permissions (in Android Manifest file)	D	C	The sensitive operations an app can perform.	✓	May include permissions not used by the app. May miss behaviors performed via non-protected intents.	[14, 29, 31, 33, 35, 36, 37, 39, 40, 43, 44, 59]
	Bytecode API calls	D	T	Android library calls.	✓	May contain dead code, include ads which use reflection, obfuscation, excludes method arguments values that are obtained at runtime.	[35, 36, 37, 38, 42, 43]
	Source Code API calls	D	T	Android library calls.	✓	May contain dead code, include ads which use reflection, obfuscation, excludes method arguments values that are obtained at runtime.	
	`<meta-data>` & `<uses-library>` elements	D	T	Libraries included.	✓	May be dead code.	[38]

information. Our key findings are:
(1) The apps' developers use descriptions in varying ways and other artifacts on the webpage, such as screenshots and videos, are also being used for specification purposes. We saw that some may be presented in differing languages (e.g. the app *Paint for Kids* has its app description in English but the demonstrational video language is Portuguese). Such results suggest that an app's description alone is not an accurate representation of either program behavior or user expectation of a program's behavior, and that alternative analyses that can extract information from artifacts such as video or screenshots is needed. To get a full specification, these analyses should ultimately be merged.
(2) While it is known that apps over-request permissions and therefore permissions may contain behavioral capabilities not actually utilized by the app, we saw several apps whose permission sets lacked information regarding important behaviors the apps perform. This offers yet another reason why permissions alone cannot be used as an app's specification.
(3) The majority of apps in our study contain ads and analytic libraries. Under the current developer practices such libraries present obstacles in program understanding. Specifically, they complicate the use of API calls through their use of reflection and injection of large amounts of dead code. Little information about their behavior is available on an app's Play store page other than the mention of their absence in paid versions and user comments mentioning related crashes, freezing and general annoyance. No mention is ever made about analytics.
(4) While the number of downloads, user ratings and user comments may seem like similar measures of user approval and even program quality, we observed contradictory results in several apps where users reported many bugs but users continued to download the app and rate it highly. Within our study, such negative comments were independent of whether the developer had received a Top Developer badge.

4.3 Evaluating Static Analysis

In this section, we present some preliminary results of our static analysis that generates permission paths. Thus far, we have applied our tool for generating permission paths to 48 of the challenges apps produced as part of the APAC program. Because the challenge apps include source code, there are no concerns about problematic decompilation. In Table 6, we list these preliminary results. For each app, we list the number of permission-sensitive methods found (Sensitive). Then, we classify the permission paths found into those that begin with a GUI handler (e.g., `onClick`), lifecycle handlers (e.g., `onCreate`), and background handlers. The final column list the remaining handlers that we have not yet classified.

For GUI handlers, we examine how many of those entry points can we discover a view refinement (see Section 3.3.5). For lifecycle handlers, we show how many of those entry points can we discover an Activity transition (see Section 3.3.6).

Table 6: Preliminary empirical results for finding permission paths. On each row, we list the Android application (App), the total number of sensitive methods found (Sensitive), the number of permission paths that begin from a GUI handler (GUI Handlers-Tot), the number of GUI handler permission paths where view refinements were found (GUI Handlers-VR), the effectiveness of view refinement (GUI Handlers-Eff), the number of permission paths from a lifecycle handler (Lifecycle Handlers-Tot), the number lifecycle handler permission paths with an associated Activity transition (Lifecycle Handlers-AT), the effectiveness of Activity transitions (Lifecycle Handlers-Eff), the number of permission paths that end in background entry points (Bg), and the remaining permission paths whose entry point have unknown classification (Unk).

App	Sensitive	GUI Handlers			Lifecycle Handlers			Bg	Unk
		Tot	VR	Eff	Tot	AT	Eff		
AndroidGame	1	0	0	-	0	0	-	0	1
AudioAssure	1	1	1	100%	0	0	-	0	0
AudioSidekick	1	1	1	100%	0	0	-	0	0
AutoQuiet	1	0	0	-	1	0	0%	0	0
MorseCode	1	1	0	0%	0	0	-	0	0
PasswordVault	1	1	1	100%	0	0	-	0	0
ShyGuyCamera	1	1	1	100%	0	0	-	0	0
TitleScreenActivity	1	1	1	100%	0	0	-	0	0
Expenses	2	0	0	-	2	2	100%	0	0
MeediaFun	2	0	0	-	2	0	0%	0	0
Meetloaf	2	0	0	-	0	0	-	2	0
Memotis	2	0	0	-	0	0	-	0	2
Orienterring	2	0	0	-	2	0	0%	0	0
Pondl	2	0	0	-	0	0	-	0	2
reveal	2	0	0	-	0	0	-	2	0
tomdroid	2	1	0	0%	0	0	-	1	0
TopicReel	2	0	0	-	2	0	0%	0	0
InstantMessage	3	3	3	100%	0	0	-	0	0
InstantReplay	3	2	2	100%	1	1	100%	0	0
RunningMap	4	0	0	-	0	0	-	4	0
SuperNote	4	1	0	0%	3	3	100%	0	0
ColorMatcher	5	1	1	100%	0	0	-	0	4
FlappyWidget	6	0	0	-	6	6	100%	0	0
SysWatcher	6	0	0	-	2	0	0%	4	0
KnectChat	7	7	7	100%	0	0	-	0	0
omnidroid	7	1	1	100%	6	0	0%	0	0
AgentSmith	8	0	0	-	0	0	-	8	0
apg-e	8	0	0	-	0	0	-	8	0

apg-m	8	0	0	-	0	0	-	8	0
Bites	8	0	0	-	0	0	-	8	0
MainRedditActivity	8	6	6	100%	0	0	-	2	0
Reveal	8	0	0	-	0	0	-	8	0
SMSBot	8	0	0	-	0	0	-	8	0
AndroidMap	9	9	9	100%	0	0	-	0	0
DvorakKeyboard	9	9	0	0%	0	0	-	0	0
Aperture	10	0	0	-	0	0	-	0	10
AWeather	10	0	0	-	0	0	-	10	0
com.deepSoft.fullcontrol	10	9	9	100%	1	0	0%	0	0
SysMon	10	4	0	0%	6	6	100%	0	0
Arabicize	11	0	0	-	0	0	-	11	0
NewsCollator	11	0	0	-	0	0	-	11	0
Entomologist	12	0	0	-	0	0	-	12	0
Sentinental	13	0	0	-	0	0	-	13	0
SmartCameraWebCam	13	2	0	0%	0	0	-	9	2
TextSecure	15	0	0	-	0	0	-	15	0
Sanity	18	1	0	0%	1	1	100%	2	14
OpenGPSTracker	23	0	0	-	0	0	-	0	23
Vermillion	29	0	0	-	12	0	0%	17	0
Total	330	62	43	69%	47	19	40%	163	58

5 CONCLUSIONS

The primary contribution of this project is to show that the user interface part of an Android app may be used to define what it means for an app to be malicious. An app that has a user interface that is inconsistent with its behavior is more likely to be malicious than another that declares everything in its textual description as well as user interface.

More specifically, we developed a new approach to identifying inconsistencies in Android apps. Our results on using a descriptor-based approach showed the promise of applying comprehensive and automated analysis techniques on Android apps.

In our broader work on synthsis of app behavior, we identified the puzzle pieces that are required to determine an app's true behavior and security. We call this a puzzle because, as we have shown, the behavior (or app's specification) is never fully explicitly revealed to an end user in a document. Instead, the behavior is revealed "in pieces" across the multiple artifacts that typically accompany an app. These pieces need to be put together to understand true behavior for security. This is important if we are concerned about evaluating the app's quality or testing it for functional correctness or suspicious behavior.

In our study of Android apps, we have learned that developers do not use descriptions in a consistent manner. In order to fully understand a complete description, one must add visual (screenshot and video) analysis. We also find that artifacts that provided inferred behavior (e.g., AndroidManifest.xml file's permissions section is thought to provide complete knowledge of the app's phone usage such as dialer, camera, and microphone) may not be complete as there are dynamic behaviors from advertisements that are almost completely ignored by developers and poorly documented. Finally, we have learned that user ratings

follow an app through its history, therefore newly introduced faulty behavior will not be reflected in ratings.

6 Bibliography

[1] Google: Android documentation, . URL `http://developer.android.com`.

[2] Google play store listings start showing the new content ratings and interactive elements on the web, . URL `https://lockerdome.com/androidpolice/7718397613978644`.

[3] Google play store. URL `http://play.google.com/store/apps/`.

[4] symantec: Android.umeng. URL `http://www.symantec.com/security_response/writeup.jsp?docid=2014-040307-5749-99`.

[5] apktool, 2015. `http://ibotpeaches.github.io/Apktool/`.

[6] dex2jar, 2015. `https://github.com/pxb1988/dex2jar`.

[7] Khalid Alharbi, Sam Blackshear, Emily Kowalczyk, Atif M. Memon, Bor-Yuh Evan Chang, and Tom Yeh. Android apps consistency scrutinized. In *CHI '14 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '14, pages 2347–2352, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2474-8.

[8] D. Amalfitano, A.R. Fasolino, and P. Tramontana. Using dynamic analysis for generating end user documentation for web 2.0 applications. In *Web Systems Evolution (WSE), 2011 13th IEEE International Symposium on*, pages 11–20, Sept 2011.

[9] D. Amalfitano, A. Fasolino, P. Tramontana, B. Ta, and A. Memon. MobiGUITAR – a tool for automated model-based testing of mobile apps. *Software, IEEE*, PP(99):1–1, 2014.

[10] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. Using GUI ripping for automated testing of android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 258–261, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1204-2.

[11] J. Annuzzi, L. Darcey, and S. Conder. *Introduction to Android Application Development: Android Essentials*. Developer's Library. Addison Wesley, 2014. ISBN 9780321940261.

[12] Gary Anthes. Invasion of the mobile apps. *Commun. ACM*, 54(9):16–18, September 2011. ISSN 0001-0782.

[13] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. Pscout: Analyzing the android permission specification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 217–228, 2012. ISBN 978-1-4503-1651-4.

[14] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. PScout: analyzing the Android permission specification. In *Computer and Communications Security (CCS)*, pages 217–228, 2012.

[15] Tanzirul Azim and Iulian Neamtiu. Targeted and depth-first exploration for systematic testing of Android apps. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 641–660, 2013.

[16] David Barrera, H Güneş Kayacik, Paul C van Oorschot, and Anil Somayaji. A methodology for empirical analysis of permission-based security models and its application to android. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 73–84. ACM, 2010.

[17] Sam Blackshear, Bor-Yuh Evan Chang, and Manu Sridharan. Thresher: Precise refutations for heap reachability. In *Programming Language Design and Implementation (PLDI)*, pages 275–286, 2013.

[18] Sam Blackshear, Bor-Yuh Evan Chang, and Manu Sridharan. Selective control-flow abstraction via jumping. 2015. In submission.

[19] Sam Blackshear, Alexandra Gendreau, and Bor-Yuh Evan Chang. Droidel: a general approach to Android framework modeling. In *State of the Art in Program Analysis (SOAP)*, pages 19–25, 2015.

[20] Kevin J. Boudreau. Let a thousand flowers bloom? an early look at large numbers of software app developers and patterns of innovation. *Organization Science*, 23(5): 1409–1427, 2012.

[21] Pern Hui Chia, Yusuke Yamamoto, and N Asokan. Is this app safe?: a large scale study on application permissions and risk signals. In *Proceedings of the 21st international conference on World Wide Web*, pages 311–320. ACM, 2012.

[22] E.J. Chikofsky and II Cross, J.H. Reverse engineering and design recovery: a taxonomy. *Software, IEEE*, 7(1):13–17, Jan 1990.

[23] Myra B. Cohen, Si Huang, and Atif M. Memon. Autoinspec: Using missing test coverage to improve specifications in GUIs. In *Proceedings of the 2012 IEEE 23rd International Symposium on Software Reliability Engineering*, ISSRE '12, pages 251–260, 2012. ISBN 978-0-7695-4888-3.

[24] Dimitrios Damopoulos, Georgios Kambourakis, Stefanos Gritzalis, and SangOh Park. Exposing mobile malware from the inside (or what is your mobile app really doing?). *Peer-to-Peer Networking and Applications*, 7(4):687–697, 2014. ISSN 1936-6442.

[25] Francesco Di Cerbo, Michele Bezzi, SamuelPaul Kaluvuri, Antonino Sabetta, Slim Trabelsi, and Volkmar Lotz. Towards a trustworthy service marketplace for the future internet. In Federico lvarez, Frances Cleary, Petros Daras, John Domingue, Alex Galis, Ana Garcia, Anastasius Gavras, Stamatis Karnourskos, Srdjan Krco, Man-Sze

Li, Volkmar Lotz, Henning Mller, Elio Salvadori, Anne-Marie Sassen, Hans Schaffers, Burkhard Stiller, Georgios Tselentis, Petra Turkama, and Theodore Zahariadis, editors, *The Future Internet*, volume 7281 of *Lecture Notes in Computer Science*, pages 105–116. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-30240-4.

[26] Gianluca Dini, Fabio Martinelli, Ilaria Matteucci, Marinella Petrocchi, Andrea Saracino, and Daniele Sgandurra. A multi-criteria-based evaluation of android applications. In *Trusted Systems*, pages 67–82. Springer, 2012.

[27] 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*, CCS '11, pages 627–638, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0948-6.

[28] 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*, SOUPS '12, pages 3:1–3:14, 2012. ISBN 978-1-4503-1532-6.

[29] Bin Fu, Jialiu Lin, Lei Li, Christos Faloutsos, Jason Hong, and Norman Sadeh. Why people hate your app: Making sense of user feedback in a mobile app store. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1276–1284. ACM, 2013.

[30] Adam P Fuchs, Avik Chaudhuri, and Jeffrey S Foster. Scandroid: Automated security certification of android applications. *Manuscript, Univ. of Maryland, http://www. cs. umd. edu/avik/projects/scandroidascaa*, 2(3), 2009.

[31] Laura V Galvis Carreño and Kristina Winbladh. Analysis of user comments: an approach for software requirements evolution. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 582–591, 2013.

[32] N. Gold, A. Mohan, C. Knight, and M. Munro. Understanding service-oriented software. *Software, IEEE*, 21(2):71–77, March 2004.

[33] 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.

[34] Michael C. Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WISEC '12, pages 101–112, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1265-3.

[35] A. Hammershoj, A. Sapuppo, and R. Tadayoni. Challenges for mobile application development. In *Intelligence in Next Generation Networks (ICIN), 2010 14th International Conference on*, pages 1–8, Oct 2010.

[36] S. Hess, F. Kiefer, and R. Carbon. Quality by construction through mconcappt: Towards using ui-construction as driver for high quality mobile app engineering. In *Quality of Information and Communications Technology (QUATIC), 2012 Eighth International Conference on the*, pages 313–318, Sept 2012.

[37] Leonard Hoon, Rajesh Vasa, Jean-Guy Schneider, and Kon Mouzakis. A preliminary analysis of vocabulary in mobile app user reviews. In *Proceedings of the 24th Australian Computer-Human Interaction Conference*, OzCHI '12, pages 245–248, 2012. ISBN 978-1-4503-1438-1.

[38] Minqing Hu and Bing Liu. Mining and summarizing customer reviews. In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '04, pages 168–177, 2004. ISBN 1-58113-888-1.

[39] Jianjun Huang, Xiangyu Zhang, Lin Tan, Peng Wang, and Bin Liang. Asdroid: Detecting stealthy behaviors in android applications by user interface and program behavior contradiction. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1036–1046. ACM, 2014.

[40] IBM Research. T.J. Watson Libraries for Analysis (WALA). `http://wala.sf.net`.

[41] A. Jain. Apps marketplaces and the telecom value chain. *Wireless Communications, IEEE*, 18(4):4–5, August 2011.

[42] Slinger Jansen and Ewoud Bloemendal. Defining app stores: The role of curated marketplaces in software ecosystems. In Georg Herzwurm and Tiziana Margaria, editors, *Software Business. From Physical Products to Software Services and Solutions*, volume 150 of *Lecture Notes in Business Information Processing*, pages 195–206. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-39335-8.

[43] Yiming Jing, Gail-Joon Ahn, Ziming Zhao, and Hongxin Hu. Riskmon: Continuous and automated risk assessment of mobile applications. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, CODASPY '14, pages 99–110, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2278-2.

[44] M.E. Joorabchi, A. Mesbah, and P. Kruchten. Real challenges in mobile app development. In *Empirical Software Engineering and Measurement, 2013 ACM / IEEE International Symposium on*, pages 15–24, Oct 2013.

[45] Henry Lee and Eugene Chuvyrov. Packaging, publishing, and managing applications. In *Beginning Windows Phone 7 Development*, pages 121–138. Apress, 2010. ISBN 978-1-4302-3216-2.

[46] Rory Lewis and Laurence Moroney. Deploying your app to the app store. In *iPhone and iPad Apps for Absolute Beginners*, pages 273–303. Apress, 2013. ISBN 978-1-4302-6361-6.

[47] Jesse Liberty and Jeff Blankenburg. Get money: Profiting from your applications. In *Migrating to Windows Phone*, pages 217–242. Apress, 2011. ISBN 978-1-4302-3816-4.

[48] Steffen Liebergeld and Matthias Lange. Android security, pitfalls and lessons learned. In Erol Gelenbe and Ricardo Lent, editors, *Information Sciences and Systems 2013*, volume 264 of *Lecture Notes in Electrical Engineering*, pages 409–417. Springer International Publishing, 2013. ISBN 978-3-319-01603-0.

[49] Jialiu Lin, Shahriyar Amini, Jason I. Hong, Norman Sadeh, Janne Lindqvist, and Joy Zhang. Expectation and purpose: Understanding users' mental models of mobile app privacy through crowdsourcing. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, UbiComp '12, pages 501–510, 2012. ISBN 978-1-4503-1224-0.

[50] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. Api change and fault proneness: a threat to the success of android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 477–487. ACM, 2013.

[51] Eric Malmi. Quality matters: Usage-based app popularity prediction. In *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct Publication*, UbiComp '14 Adjunct, pages 391–396, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3047-3.

[52] Hausi A. Müller, Scott R. Tilley, and Kenny Wong. Understanding software systems using reverse engineering technology perspectives from the rigi project. In *Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research: Software Engineering - Volume 1*, CASCON '93, pages 217–226, 1993.

[53] Roland M. Müller, Björn Kijl, and Josef K. J. Martens. A comparison of inter-organizational business models of mobile app stores: There is more than open vs. closed. *J. Theor. Appl. Electron. Commer. Res.*, 6(2):63–76, August 2011. ISSN 0718-1876.

[54] Damien Octeau, Somesh Jha, and Patrick McDaniel. Retargeting android applications to Java bytecode. In *Foundations of Software Engineering (FSE)*, page 6, 2012.

[55] Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. Whyper: Towards automating risk assessment of mobile applications. In *USENIX Security*, volume 13, 2013.

[56] Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David Wagner. Addroid: Privilege separation for applications and advertisers in android. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '12, pages 71–72, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1648-4.

[57] Hao Peng, Chris Gates, Bhaskar Sarma, Ninghui Li, Yuan Qi, Rahul Potharaju, Cristina Nita-Rotaru, and Ian Molloy. Using probabilistic generative models for ranking risks of android apps. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 241–252. ACM, 2012.

[58] Thanasis Petsas, Antonis Papadogiannakis, Michalis Polychronakis, Evangelos P. Markatos, and Thomas Karagiannis. Rise of the planet of the apps: A systematic

study of the mobile app ecosystem. In *Proceedings of the 2013 Conference on Internet Measurement Conference*, IMC '13, pages 277–290, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1953-9.

[59] Taylor Pierce and Dave Wooldridge. Keys to the kingdom: The app store submission process. In *The Business of iOS App Development*, pages 333–376. Apress, 2014. ISBN 978-1-4302-6238-1.

[60] Elisabeth Platzer and Otto Petrovic. Learning mobile app design from user review analysis. *iJIM*, 5(3):43–50, 2011.

[61] Paul POCATILU. Android Applications Security. *Informatica Economica*, 15(3):163–171, 2011.

[62] Michael Privat and Robert Warner. Submitting to the mac app store. In *Beginning OS X Lion Apps Development*, pages 323–363. Apress, 2011. ISBN 978-1-4302-3720-4.

[63] Zhengyang Qu, Vaibhav Rastogi, Xinyi Zhang, Yan Chen, Tiantian Zhu, and Zhong Chen. Autocog: Measuring the description-to-permission fidelity in android applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1354–1365. ACM, 2014.

[64] Roy Sandberg and Mark Rollins. Making sure your app will succeed. In *The Business of Android Apps Development*, pages 15–30. Apress, 2013. ISBN 978-1-4302-5007-4.

[65] Irina Shklovski, Scott D. Mainwaring, Halla Hrund Skúladóttir, and Höskuldur Borgthorsson. Leakiness and creepiness in app space: Perceptions of privacy and mobile app use. In *Proceedings of the 32Nd Annual ACM Conference on Human Factors in Computing Systems*, CHI '14, pages 2347–2356, 2014. ISBN 978-1-4503-2473-1.

[66] Falafel Software. Taking your app to market. In *Pro Windows Phone App Development*, pages 517–528. Apress, 2013. ISBN 978-1-4302-4782-1.

[67] Rajesh Vasa, Leonard Hoon, Kon Mouzakis, and Akihiro Noguchi. A preliminary analysis of mobile app user reviews. In *Proceedings of the 24th Australian Computer-Human Interaction Conference*, OzCHI '12, pages 241–244, 2012. ISBN 978-1-4503-1438-1.

[68] A. von Mayrhauser and A.M. Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, Aug 1995.

[69] J. Ziegler, M. Graube, J. Pfeffer, and L. Urbas. Beyond app-chaining: Mobile app orchestration for efficient model driven software generation. In *Emerging Technologies Factory Automation (ETFA), 2012 IEEE 17th Conference on*, pages 1–8, Sept 2012.