ABSTRACT

Title of dissertation: INTERACTION-BASED SECURITY
FOR MOBILE APPS

Kristopher Micinski, Doctor of Philosophy, 2017

Dissertation directed by: Professor Jeffrey S. Foster,

Department of Computer Science

Mobile operating systems pervade our modern lives. Security and privacy is of particular concern on these systems, as they have access to a wide range of sensitive resources. Apps access these sensitive resources to help users perform tasks. However, apps may use these sensitive resources in a way that the user does not expect. For example, an app may look up reviews of restaurants nearby, but also leak the user's location to an ad service every hour.

I claim that interaction serves as a valuable component of security decisions, because the user's interaction with the app's user interface (UI) deeply informs their mental model of how apps access sensitive data. I introduce the notion of *interaction-based security*, wherein security decisions are driven by this interaction.

To help understand and enforce interaction-based security, I present four pieces of work. The first is Redexer, which performs binary instrumentation of off-the-shelf Android binaries. Binary instrumentation is a useful tool for enforcing and studying

security properties. I demonstrate one example of how Redexer can be used to study location privacy in apps.

Android permissions constrain how data enters apps, but do not constrain how the information is used or where it goes. Information-flow allows us to formally define what it means for data to leak from applications, but it is unclear how to use information-flow policies for Android apps, because apps frequently declassify information. I define interaction-based declassification policies, and show how they can be used to define policies for several example apps. I then implement a symbolic executor which checks Android apps to ensure they respect these policies.

Next, I test the hypothesis that the app's UI influences security decisions. I outline an app study that measures when apps use sensitive resources with respect to their UI. I then conduct a user study to measure how an app's UI influences their expectation that a sensitive resource will be accessed. I find that interactivity plays a large role in determining user expectation of sensitive resource use, and that apps largely access sensitive resources interactively. I also find that users may not always understand background uses of these sensitive resources and using them expectation requires special care in some circumstances.

Last, I present a tool which can help a security auditor quickly understand how apps use resources. My tool uses a novel combination of app logging, symbolic execution, and abstract interpretation to infer a formula that holds on each permission use. I evaluate my tool on several moderately-sized apps and show that it infers the same formulas we laboriously found by hand.

INTERACTION-BASED SECURITY FOR MOBILE APPS

by

Kristopher Micinski

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2017

Advisory Committee:
Professor Jeffrey S. Foster, Chair/Advisor
Assistant Professor Dana Dachman-Soled
Professor Jennifer Golbeck
Assistant Professor Michelle L. Mazurek
Assistant Professor David Van Horn

Dedication

For all of the family and friends who supported me over the past six years.

Acknowledgements

I would not have completed this dissertation without the help of many people who offered thoughtful and challenging advice and perspective.

My advisor, Jeff Foster, was instrumental in helping develop an appreciation for good science. I was initially surprised at Jeff's outlook on research. Instead of being focused on a particular collection of techniques or research areas, Jeff was someone who cared foremost about problems. He inspired a sense that there will always be exciting research problems to work on—especially if we're willing to step outside of the area with which we're comfortable.

Michelle Mazurek also provided deeply helpful direction. Michelle taught me an appreciation for techniques from Human-Computer Interaction, and was crucial in helping form the parts of my thesis related to usability. Although not a programming language researcher, Michelle quite capably understood and thoughtfully commented on my work in that area.

I was also guided by several other professors at Maryland and elsewhere. Michael Hicks offered lots of useful advice, and provided a different (but equally thoughtful) perspective from Jeff. He has consistently challenged me to acheive more. David Van Horn was helpful in providing some interesting discussions on static analysis and his methods always inspired my thought. Last, Michael Clarkson played a formative role in my thinking about formalizations of security. My

thesis was deeply informed by our work together on temporal logics for hyperproperties. He inspired me as someone who was both a great teacher and researcher, and I hope to teach as well as he someday.

The other members of the PLUM lab were also great friends and provided useful feedback and direction. Jinseong Jeon was incredibly kind and intelligent, and mentored me during my first years. Jinseong could write enormous amounts of code acting as if it was no big deal. He answered many hasty emails near deadlines. David Darais and I have had many interesting conversations about both programming languages and coffee. He has urged me to pursue ideas I might have given up on, and I look forward to his continued input. Tom Gilray shares many of my goals on visualization and static analysis, and we come at programming languages from a similar perspective. He has helped me understand the nuts and bolts of abstract interpretation in a way that distilled it simply. He has offered helpful critique on my ideas.

Daniel Votipka began his PhD during my last year, and performed at the level of a very advanced graduate student quickly. I was impressed and thankful at how much he was able to get done while having relatively little experience in the areas we were working in. Rock Stevens was a great aid and a master scripter. He had a laser focus on achieving goals, and I have no idea how he was able to download a few hundred apps from Google Play in a few hours, circumventing the security controls which I assumed would stop us. I was fortunate enough to work with a number of talented undergrads, including Philip Phelps and Rebecca Norton.

Many, many people reviewed my papers over the years and offered useful and constructive feedback. Among these are Sascha Fahl, Yasemin Acar, and members of HCIL at Maryland.

This research was supported in part by NSF CNS-1064997 and by a research award from Google. The work was also supported by the partnership between UMIACS and the Laboratory for Telecommunication Sciences.

Last, I am greatful to my family and friends. My parents have been incredibly supportive of all of my goals, in every way I might imagine. I owe them more than I can give. They gave me a lot of life advice that helped me succeed in grad school, and nurtured an appreciation for honesty and self-accountability.

Contents

List of Figures

List of Abbreviations

ACG	Access Control Gadget
AIC	Akaike Information Criterion
API	Application Programming Interface
APK	Android application Package
GPS	Global Positioning System
GUI	Graphical User Interface
IBNI	Interaction-Based Non-Interference
LTL	Linear Temporal Logic
MTurk	Amazon Mechanical Turk
OS	Operating System
UI	User Interface

Chapter 1: Introduction

Mobile devices pervade our daily lives. Surveys from the Pew Research Center show that smartphone ownership among US adults grew from just 35% in 2011 to 77% in 2016 [1]. With these devices ever-present in our daily lives, it is crucial that users be able to understand the devices' security mechanisms.

Mobile devices have access to a wide range of sensitive resources: the user's camera, microphone, text messages, and more. Apps can use these sensitive resources to help perform tasks and create a immersive user-experience. For example, an app may read the user's text messages to eliminate spam or take a picture after some amount of time. But the ability for apps to access these sensitive resources leaves open the possibility of misuse.

To combat this, mobile operating systems implement security mechanisms. iOS and Android (together comprising over 99% of the smartphone market [2]) both implement a *permission* system that apps must honor. Before an app uses a sensitive resource, the app must have been granted access to the corresponding permission (e.g., CAMERA).

Permission systems are intended to keep information from being accessed without authorization, however it is unclear if their implementation achieves this. All permissions systems balance transparency with habituation. On one hand, the system may ask users for permission every time a resource is used by an app. This may achieve more transparency, but with the risk that users may begin to ignore notifications [3]. On the other hand, the system may ask for permission only *once*, but this risks the user being unaware of certain accesses. At the time of writing, both Android and iOS ask for permission only once for each app.

It has recently been suggested [4] that security decisions are largely influenced by context. Specific to mobile devices, there is a large amount of work [5–8] that suggests app context deeply informs user security decisions. Context is naturally established in apps by the user's interactions with the UI. For example, the user will likely expect that the app may take a picture after clicking on a specific "take photo" icon, but not when the phone is sitting on their desk.

This motivates the notion of *interaction-based* security. I define interaction-based security as security and privacy policies which are informed by the user's interaction with the user interface of an application.

Thus, I provide the following thesis:

Interaction-based security can be supported in mobile operating systems without any changes to the underlying system. Further, formal notions of security (such as information flow) can be naturally modified to accommodate interaction-based security. With advances to program understanding tools, we can aid auditors in quickly discovering interaction-based security policies.

To substantiate this thesis, I present four main pieces of work that illustrate how program analysis and understanding can be used to support interaction-based security. The first is Redexer, a bytecode rewriter for Android. Redexer allows a variety of transformations to be performed on Android apps. For example, it allows retrofitting apps with finer grained permissions so that users may control the granularity of location information given to an app. I introduce interaction-based information flow policies. These policies formally relate the information released by an app to sequences of UI interactions made by the user. I implement ClickRelease, which analyzes Android apps to check that they satisfy interaction-based policies.

Next, I directly study users to understand how user interactions relate to intuitions about when resources are used. I do this by performing both an app study and a user study, to understand how top apps access resources and understand how this relates to user expectations.

Last, I use a novel combination of execution logging, symbolic execution, and abstract interpretation to design a tool which allows a security auditor to quickly understand the different circumstances under which permissions may be used by an app. I conclude by commenting on how we can apply the lessons learned in this work to enhance app security, and highlight future directions of work in this area.

Note that within this dissertation, I focus on the Android operating system. I speculate that interaction-based security could offer insight about other mobile operating systems (such as iOS).

1.1 Binary Rewriting for Enhanced Security on Android

I begin by describing Redexer [9,10], a tool I (along with my colleagues) built to perform static binary rewriting on Android apps. Rewriting Dalvik bytecode is challenging because of the many subtle invariants to which the bytecode structure must adhere. For example, instructions include register ranges which they are allowed to address. Redexer tackles this by giving the programmer an API that allows inserting bytecode inside of a method without maintaining these invariants. Redexer includes passes that clean up rewritten bytecode to maintain these invariants, which greatly simplifies writing binary transformations. I describe these in Chapter 2.

Binary rewriting is useful for a variety of transformations to Android apps, and I rely upon Redexer for subsequent chapters of this dissertation. In Chapter 3, I outline one specific use of Redexer to replace the default Android location API with an API that offers coarser access to the user's location. I show that most apps can have their inputs truncated by one kilometer or more without appreciably degrading the results presented.

1.2 Interaction-Based Declassification Policies

Permissions protect the ways in which data enters apps. However, once apps read data, permissions do not guard what apps do with that data. *Information flow* [11] allows formally reasoning about how sensitive data is released by applications to (potentially untrusted) observers. One standard information-flow property is noninterference: a public observer can learn nothing about the private inputs to the program. In practice, information flow is challenging to apply because many realistic applications leak information by design (e.g., telling the user they entered the wrong password leaks the fact that the password was not what they guessed).

There has been a large amount of work [12–17] on incorporating *declassification* into noninterference. While there is some work [18, 19] that addresses information flow for interactive systems, no prior work of which I am aware allows incorporating the user interface into information flow properties.

In chapter 4, I present *interaction-based declassification* to allow incorporating an app's user interface into information flow policies [20]. This allows us to state policies such as the following:

$$\mathsf{ph!} * \wedge (\mathcal{F}(\mathsf{sendBtn!unit} \wedge \mathit{last}(\mathsf{phBox}, \mathsf{true}))) \rhd \mathit{Low}$$

The formalisms speak about channels, time-varying streams of data along which events occur. Here, the $\mathsf{ph!*}$ notation is used to represent a write of any value on the channel ph, which is used here to represent the phone number. The policy can be read as follows: "At time i, if the input at time i is on the ph channel, and at some *future* (indicated by the stylized $\mathcal{F}$) point j, the send button is pressed ($\mathsf{sendBtn!unit}$) *while* the last setting of the phBox was true, the input at time i may be declassified to security level *Low*."

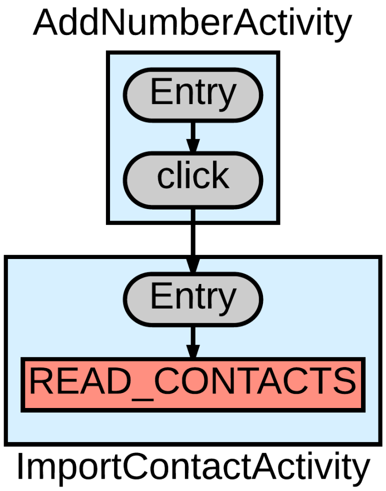
Because these policies rely on precise temporal orderings of program states, I used *symbolic execution*, a program analysis that executes the program with symbolic variables to build up sets of states along with formulas that must hold within those states. In a previous collaboration I worked on SymDroid [21], a symbolic executor for Dalvik bytecode. I extended SymDroid to create ClickRelease. ClickRelease checks these interaction-based policies by collecting pairs of program paths and forming equations over them. These equations encode interaction-based declassification to guarantee that apps only leak data in accordance with the policy. I applied ClickRelease to four synthetic apps including two malicious variants of those apps. ClickRelease was able to find information flow leaks in the malicious variants and generate counterexamples.

1.3 User Interactions and Permission Use on Android

After exploring interaction-based declassification, I ran two studies to understand to what extent such policies match user perceptions of security and the extent to which apps access resources interactively [22]. The first measures 150 top apps to determine the interaction patterns apps used to access permissions, and the second one measures users to test how closely those patterns align with user expectation.

I first implement AppTracer, a tool that helps a skilled auditor understand the relation between an app's GUI and how it accesses sensitive resources [23]. AppTracer inserts instrumentation into an app that logs its control flow. I can then run the rewritten app on a device either manually or via automated exploration and capture an execution log. Next, AppTracer transforms this log of program behavior into a graph that helps an auditor understand which GUI events caused which permission uses.

I used the graph AppTracer produced and a human-curated codebook to assign a set of codes to each permission use within each app. For example, in the graph below (a subgraph of one produced by AppTracer), the app read the contacts immediately after a button click, which is coded as a *Click* use of that permission.



Next, I conducted a user study. This study evaluated when users expected permission use to occur with respect to the app's GUI. It is comprised of a walkthrough of an app interspersed with questions about what resources the user expects will be used. For example, in one scenario the user first sees an app's home screen, then presses the "voice order coffee" button. To mirror the current Android permissions system, they are then prompted to allow access to the microphone. In other scenarios this dialog occurs at the beginning of the app or not at all. After seeing the dialog, the user is asked a series of Likert-style questions assessing their expectation a given resource will be used at that point in the app. The user is then shown another interaction with the app, in this case going back to the phone's home screen. They are again asked expectation questions to understand how their perceptions of permission use changes as context changes.

Combining the results of the two studies, I discovered that apps mostly matched user expectations, but noted some shortcomings with the permission system. Users seemed to expect the most invasive permissions (e.g., camera and microphone) to be used only after a related click. Our app study confirmed this is almost exclusively the case. We recommend that the policy of allowing sensitive resource use only after relevant clicks be made mandatory with rare exceptions. Even when users understand resources will be accessed after these clicks, Android still asks users for explicit permission via a dialog screen. This suggests Android may be too invasive, and that these interactions alone are sufficient to authorize the permission use, as the dialog was not providing any additional information. Also, we found that when users were prompted for authorization immediately after a click, they assumed that the associated resource was used *only* after that interaction. For example, if an app waits until a map screen to ask users for permission to use location, users will be much less likely to expect the location to be used later in the app (e.g., while the app is off the screen). We observed infrequent but occasional uses of this in our app study.

1.4 Permission-Use Provenance in Android Using Sparse Dynamic Analysis

AppTracer helps skilled auditors understand why apps use permissions, but it works based on traces of app events. Because of this, AppTracer associates permission uses with UI elements that are temporally close to permission uses. This works well for cases where permissions are used immediately after a button click or navigation to a new screen. However, this heuristic makes it challenging to understand the root causes of permission uses that happen in the background. Our user study suggested that these background uses may be tricky for users to understand, so I decided to pursue tools that would help understand these permission uses.

In Chapter 6, I introduce Hogarth, a system which uses a combination of app logging, symbolic execution, and abstract interpretation to explain permission uses within apps. The goal of Hogarth is to summarize why a permission use occurs in terms of the app's UI. Android apps are callback-oriented, which makes it challenging to define what it means to take a path through an app. Hogarth tackles this problem by building an inter-handler callback graph, based on a model of which callbacks may run as a result of certain registration methods. It then presents a graph of app behavior, where nodes in this graph are callbacks (some containing permission uses) and edges connect callbacks when one callback registers another.

Connecting callbacks allows an auditor to understand roughly where a permission is used within an app's source code. However, it does not allow a user to understand why that permission is used. To explain permission use, Hogarth also reconstructs a *path condition* that holds along every path to the permission use observed in the set of logs. I applied Hogarth to five moderately-sized Android apps from the F-Droid app store [24] and the contagio malware dataset [25]. I found that—while Hogarth is still a prototype—it was effectively able to recover path conditions that were found by otherwise time-consuming reverse-engineering efforts.

Chapter 2: Binary Rewriting for Android Apps

This chapter introduces Redexer, a bytecode rewriting tool that allows static instrumentation of Android binaries. Bytecode rewriting enables a wide range of dynamic analyses. Several chapters in this dissertations rely on Redexer.

Rewriting Dalvik bytecode requires significant care. For example, one common pattern in bytecode instrumentation is to add sequences of instructions to methods in key locations (e.g., before method calls or field lookups). To maintain the original semantics of the method, we must be careful to ensure that the registers used by the inserted instructions do not overwrite registers used by the target method. Redexer provides an API for a programmer to traverse the app's bytecode and insert instructions without considering these invariants. It then performs various cleanup passes to ensure the app's bytecode is corrected.

I begin by describing Redexer in detail, and then I discuss one specific use of Redexer, where it is applied to study location privacy in Android apps. This demonstrates how Redexer can be applied to provide enhanced security policies on Android apps. Redexer has been utilized for both industrial and research applications.

2.1 Redexer

Redexer is a bytecode transformation framework for Android apps. Android apps are distributed in the APK file format. Given an input app, Redexer uses apktool [26] to unpack the APK into its constituent files and directories. It then parses each Dex file (those files within the APK which contain the bytecode) into an abstract syntax tree. This allows a programmer to make changes on a high-level representation of the bytecode, rather than a compact binary representation. Redexer next performs a variety of passes to perform bytecode manipulation. Finally, it dumps out the transformed app and repackages it as an output APK. In the following subsections, I detail the passes that Redexer provides to allow app transformation.

2.1.1 Parsing Android Bytecode

As a first stage, Redexer includes a parser for Dalvik bytecode. Dalvik bytecode files are structured as a series of indexed "pools" that contain, among others, strings, types, field signatures, method signatures, classes, field definitions, and method definitions. The various pools are tightly intertwined, with many pointers from one to another, e.g., a method signature contains a list of pointers to the type pool, and the type pool contains pointers to elements in the string pool (containing the actual type names). Dalvik bytecode instructions (which appear in method definitions) often refer to elements in various pools, e.g., method invocation instructions include a pointer to a method signature. Before the Dalvik Virtual Machine executes a bytecode file, it first *verifies* it to check that, among other things, the file is well-formed and the method bodies are type safe.

2.1.2 Linking

Redexer includes an API for statically linking Dex files together. This allows a programmer to instrument an app by inserting method calls to a library they write that performs the main work. For example, Redexer's logging library inserts calls to Logger.logMethodEntry, which subsequently performs more complex work. This makes writing instrumentation simpler by allowing common utilities to be written in Java rather than as bytecode sequences.

2.1.3 Modification

Redexer's modification API allows a programmer to manipulate a Dex file in various ways. For example, it allows adding and replacing methods, classes, strings, and other pieces of the Dex file. It also includes an API to allow inserting bytecode within a method at various points.

Inserting code within a method without altering the app's original semantics is particularly tricky for several reasons. First, whenever bytecode is inserted into a method at some point, we must be careful to do so hygenically: the bytecode may not overwrite live variables at that point. To handle this, Redexer has a utility that allows the programmer to shift the registers used by a method by some constant number. This allows the programmer to use the registers within some range without fear that they are disrupting the state of live variables at that point.

Unfortunately, register shifting is complicated by the fact that instructions include bounds on the registers that they can access. For example, the move instruction in Dalvik can only address the first 16 registers. After shifting registers, it is possible that the method's bytecode will be malformed. One solution might be to add registers in the upper range of the set of registers used for the method. Unfortunately, this does not work: the Android calling convention dictates that the n arguments to the method are put in the last n registers the method reserves.

To rectify this problem, Redexer includes a cleanup pass that inserts the appropriate transformation to move registers into the right place. For example, a move instruction will be replaced by a move-from16 instruction which can address 65k registers. This is slightly complicated in the case of comparisons, which can only address the first 16 registers. In this case, Redexer would insert instructions to move the register from its incorrect position (above 16) into the a register in the lower range. However, this requires knowing whether or not the register is an object or not, because—while the comparison bytecode is polymorphic—the move instructions are not. Using an incorrect move instruction will result in verification errors. To account for this, Redexer performs dataflow analysis to recover whether or not the register is an object.

Inserting instructions is also tricky because of the way exceptions are handled within Dalvik. Dalvik exceptions are implemented by specifying an exception table, which lists ranges of bytecode offsets to denote try blocks and associates those with exception handlers of a given type. However, within an exception handler the

Dalvik virtual machine will additionally perform an analysis to ascertain whether instruction may throw an exception. For example, Dalvik declares that the move instruction will not throw an exception, while an invoke form might throw an exception[1].

This presents a subtle complication for bytecode insertion: inserting invoke instructions (or other instructions which may throw exceptions according to the Dalvik VM) may alter control flow, which subsequently impacts method verification (in particular, extra live variables may be added which were not present in the original semantics). To mitigate this, Redexer allows splitting the exception into two separate handlers: one which ends right before the inserted instructions, and one which begins after the inserted instructions.

2.1.4 Visitor API and extensions

One common rewriting pattern is to traverse the app's code and perform some operation for each part of the app. To accomplish this, Redexer includes a visitor API. This allows the programmer to define some code that will be called for each class definition, method body, etc.present in the app.

To aid a programmer in constructing instrumentation, Redexer includes a few utility APIs to recover intraprocedural facts about a method body. For example, it includes an intraprocedural control-flow analysis, along with various dataflow analyses and a constant propagation analysis.

[1] the complete listing of bytecode instructions and their definitions is given in [27].

2.1.5 Logging

One particularly useful feature of Redexer is its logging API, which instruments the bytecode to log the execution of the app. The logging API is configurable to log a set of user-specified methods. It is used in several of the chapters later in this dissertation.

The logging API inserts bytecode at the beginning and end of each method in the app, making a call to logMethodEntry or logMethodExit respectively with the parameters or return value. This allows recording when specific methods are invoked. API calls are logged in a similar way: adding a call to logApiEntry and logApiExit before and after the call.

Note that there are a few complications in inserting these call instructions. First, the instrumented method's code must have its registers shifted so that the inserted bytecode sequence used to invoke logMethodEntry does not overwrite live variables used by the app. This is done using the modification APIs described in section 2.1.3. We also must perform intra-procedural control-flow analysis to recover where the end of the method is (if one exists).

Instrumenting apps with a large amount of logging can be challenging to do at scale. Apps containing many methods may slow down when logging is not implemented efficiently. logging is configurable via a set of regular expressions, so the user can choose which subset of methods are logged.

In some circumstances, we need to log all methods in a way which scales to arbitrarily large apps. We found that this can be achieved via using a combination of a worker thread and an efficient buffer to communicate between them. The logging code is optimized so that it assembles an array of objects to pass as a data structure to a worker thread. This is crucial, since the Android OS will kill an app if the main thread (the thread reacting to app UI requests) if it becomes unresponsive. The worker thread thus performs the relatively slow task of serializing the data structure to disk. The code being logged communicates with the worker thread using an efficient lock-free queue (a ConcurrentLinkedQueue described in [28]).

2.2 An Empirical Study of Location Truncation on Android

This section demonstrates one particular application of Redexer to study location privacy within Android apps. Many apps use location information. As implemented today, such apps typically send the device location over the Internet, and hence users may be understandably concerned about potential privacy violations. In response, researchers have developed a number of location privacy-enhancing mechanisms [29–34], using a range of analytical models to estimate the degree of privacy and anonymity achieved.

However, there has been much less work [35] on understanding how location privacy-enhancing technology affects the *utility* of apps—that is, does an app still work well enough even when location privacy is increased. I tackle this question by *directly* measuring how *location truncation*—quantizing the current location to a user-specified grid spacing—changes the output of a range of Android apps.

Location truncation is one of the more realistic approaches to location privacy, with several advantages: It is easy for users to understand and therefore trust. It can be implemented easily and efficiently. And it does not require foreknowledge of the set of locations to be visited. There are also some disadvantages: Location truncation primarily defeats localization attacks, rather than deanonymization or tracking [36]. Location truncation may also be overcome through a combination of prior knowledge (e.g., likely movement speeds, locations of road networks, known habits, etc) and repeated queries over a long period of time [37], [38]. Even so, we believe that for many everyday uses, these limitations are not severe for typical users.

While location truncation potentially increases privacy, it may not be ideal for all apps (e.g., it defeats turn-by-turn navigation), and its effects may be hard to measure (e.g., for apps that use location to decide which ads to show). We retrieved the top 750 apps across all categories of the Google Play store, and found that 275 of these use location. We ran each app to determine how and why they use location, and found six main categories: ad localization, listing nearby objects, fine-grained tasks (e.g., turn-by-turn navigation), geotagging content (such as pictures), finding the nearest city, and getting local weather information. (See Section 2.2.1.)

Based on this result, we chose to study the effect of location truncation on apps that produce lists of nearby objects. These apps are a good target for a variety of reasons: First, the effect of truncation is easy to determine, as we can simply look at the output list. Second, truncation is plausibly useful for these apps: on Android, most apps do not include internal maps, but rather use Android's Intent mechanism to ask the Google Maps app to do any necessary mapping (this was the case for all of the apps in our study). Thus, truncating location will help increase privacy of the user to the subject app, but will not affect the ultimate usage of the information, assuming that Google and Google Maps are trusted. Finally, location truncation gives users a very clear tradeoff, in which they can go a little out of their way in exchange for more privacy.

We implemented location truncation in CloakDroid, a tool built on top of Redexer. CloakDroid modifies a subject app to use a special location service that snaps reported locations to a latitude/longitude grid whose spacing is specified by the user. (See Section 2.2.2 for details of CloakDroid.)

To evaluate how CloakDroid affects app utility, we applied it to six Android apps and measured how their output changed under a range of conditions. We developed three different metrics to judge utility of output: The *edit distance* of the nominal list under truncation compared to the reference list without truncation; the *additional distance*, which computes how much farther away the first list item in the nominal list is compared to the reference list; and the *set intersection size*, which counts the elements common to the reference list and nominal list.

We ran each app against 10 randomly chosen locations spread across 6 regions of various sizes, ranging from New York, NY, population 8.2 million, to Decatur, TX, population 6,000. For each location–region pair, we varied the truncation grid spacing from 0.1 km to 50 km and measured the result. We found that the under the additional distance metric, there is typically no change in app utility up to 1 or 2 km truncation, and that if the user is willing to go up to 1 km out of their way, most apps can sustain truncation amounts of 5 km or more. Additionally, the most important determinant of acceptable truncation was the density of the objects listed by the app—the lower the density, the more location could be truncated before seeing an effect. Across all metrics, most of the subject apps were able to have their inputs truncated to between 5 km and 20 km without significantly degrading app utility.

To our knowledge, we are the first to directly and systematically study how location truncation affects app utility, and our results suggest location can be truncated to a significant degree without compromising app utility.

2.2.1 How Apps Use Location

We began our study by trying to understand how apps use location in practice. We downloaded the top 750 most popular free apps across all categories of the Google Play store as of April 2012, and found that 275 of these apps request location permission. We installed each app on a device or emulator and used the app, navigating through its screens until we could categorize how the app uses location information. We identified the following categories of location usage, summarized in Figure 2.1:

- Ads — Most ad networks deliver location-targeted ads to users, and ads are very common on Google Play apps. Thus, this was the most popular category of location usage. However, this category is not a good subject for our study, as it is unclear how to measure the effect of location truncation on ad selection. In Figure 2.1, the first row counts apps that use location only for ads. Often apps use location both for ads and another purpose; the count of these apps is called out separately in the remaining rows.

- Listing nearby objects — The second most common category of location usage is to find some set of objects that are near the user's current location. For example, an app might inform the user of nearby traffic accidents or construction to avoid. This is the category we selected for our study.

- Fine-grained targeted content — Many apps require very fine-grained location information to provide their service, e.g., an app that remembers where the user parked. Location truncation is not likely a good idea for these apps.

- Content tagging or check-in — Location information is often used for geotagging data (e.g., pictures or social network posts) or "checking in" at the current location. For these apps, location truncation is plausible, but it is difficult to measure the effect on utility.

- Nearest city — Several apps only need to know the current location to a very coarse degree, e.g., Living Social and Craigslist just need to know the nearest city to select the right data to display to the user. Truncating location for these apps will have essentially no effect, as they already use coarse data (though clearly it might be good from a security perspective).

- Weather — Weather apps use the current location to choose what weather data to report. We briefly explored the effect of truncation on these apps, but found two key problems: First, by nature, small location changes usually have

	w/ ads	Total
Only ads		58
Listing nearby objects	6	43
Fine-grained uses	7	34
Content tagging	4	28
Nearest city	1	22
Weather	2	21
Unsure		17
Unable to test		52

Figure 2.1: Location information usage in the top apps

no effect on weather; and second, weather data changes over time, and so it is hard to get consistent data sets because our experiments can take a few hours to run for each app.

- Unsure — There were 17 apps for which we could not determine why or how they use location information. These may be cases of over-permission, app functionality that we missed in our testing, or possibly malicious information gathering.

- Unable to test — Finally, we could not test 52 apps because they required either telephony-specific features not available on our test devices (which did not have phone or data plans) or subscriptions to paid services. For example, many vendors offer apps that work only on phones connected to their network.

Based on the results of this study, we decided to investigate the effect of location truncation on apps that list nearby objects, as this category is common and it is clearly meaningful to measure the result of truncation on these apps.

2.2.2 Implementation

Apps that use location information must request ACCESS_COARSE_LOCATION or ACCESS_FINE_LOCATION; the former allows access to a cell-tower–based location and the latter to a GPS location (if available). FINE_LOCATION access is provided at full device resolution, and COARSE_LOCATION access is truncated to 200 m or 2 km resolution, depending on the device parameters.[2] All the apps in our experiments use FINE_LOCATION.

We implemented location truncation in the form of CloakDroid, a tool that changes a subject app to receive modified location information. Location access on Android goes through a fairly narrow API, which makes it easy for CloakDroid to intercept. Apps first request an instance of class LocationManager, which includes, among others, methods to retrieve the last known location and to register a handler for location updates. CloakDroid controls the granularity of location information by modifying the subject app so that, instead of using Android's LocationManager, it uses a replacement class provided by CloakDroid. This leverages the modification APIs of Redexer described in section 2.1.3. The replacement for LocationManager delegates all calls to the system's LocationManager, but adds a user-specified amount of truncation before returning coordinates.

Once we can intercept calls that pass location information to the app, it is easy to modify the coordinates however we wish. For our experiments, we truncate location information, snapping to a user-specified grid, as illustrated in Figure 2.2.

[2] https://android.googlesource.com/platform/frameworks/base/+/refs/heads/master/location/java/android/location/Location.java

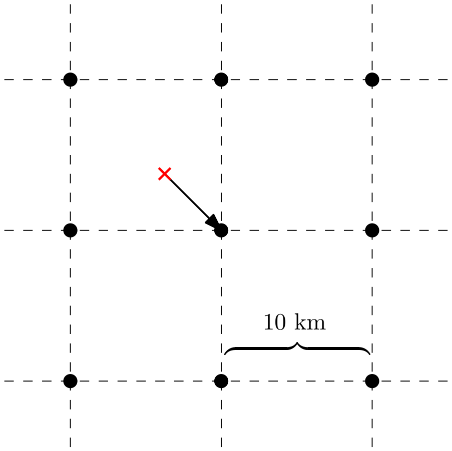


Figure 2.2: The user (represented by the red cross) has their location truncated to a grid of fixed points.

Formally, our implementation truncates to a grid with spacing s in kilometers using the formulae:

$$lat' = s_{lat}\lceil lat/s_{lat}\rceil \quad long' = s_{long}\lceil long/s_{long}\rceil$$

where lat and $long$ are the actual latitude and longitude, and their primed versions are the new coordinates. Here s_{lat} and s_{long} are the grid spacing translated into degrees of latitude and longitude, respectively. For latitude we use the WGS 84 approximation for North America [39]:

$$s_{lat} = \frac{s \text{ km}}{111.5 \text{ km/deg}}$$

For longitude we use a standard approximation [40]:

$$s_{long} = s \text{ km} \cdot \frac{180\sqrt{(1 - e^2 \sin^2(\phi))}}{\pi a \cos(\phi)}$$

where $e^2 = (a^2 - b^2)/b^2$ is the eccentricity of Earth, ϕ is the latitude, a is the radius to the equator, and b is the radius to the poles.

We verified that CloakDroid works correctly in two ways. First, we inserted logging code in CloakDroid to give the original position and position after location truncation. We then took the set of locations from our testcases, truncated them to varying amounts, and then verified the resulting positions were correct. Second, we confirmed that our subject apps' behaviors changed in sensible ways as we varied the location truncation amount.

The Android location manager API also can provide speed information, which our implementation truncates speed using a similar formula. However, none of our subject apps use speed information. Moreover, device-reported speed is often unreliable, so many apps ignore it, preferring instead to estimate speed using successive location fixes.

2.2.3 Experimental Design

We conducted a systematic study of the effect of location truncation on apps that list nearby objects, the largest category in Figure 2.1 for which location truncation's effect is clearly measurable. We selected a variety of such apps, listed in Figure 2.3a, from Google Play. We chose apps that rely on a variety of different data sets and run under our tool chain with minimal difficulty. We then modified each app using CloakDroid, as outlined in Section 2.2.2. Next, we randomly chose a set of locations and then captured the output list of each app as we varied the

Name	Objects in List
Gasbuddy	Gas stations
Restaurant Finder	Restaurants
Hospitals Near Me	Hospitals
WebMD	Pharmacies and clinics
Walmart	Stores
TD Bank	ATMs and branches

(a) Descriptions of each subject app.

City	Population	Radius (km)
New York, NY	8,200,000	30
Dallas, TX	1,200,000	20
New Haven, CT	130,000	10
Baltimore, MD	620,000	12
Redmond, WA	54,000	4
Decatur, TX	6,000	4

(b) Selected population centers.

0 km	0.1 km	0.2 km	0.5 km	1 km
2 km	5 km	10 km	20 km	50 km

(c) Truncation amounts tested.

Figure 2.3: Experimental parameters.

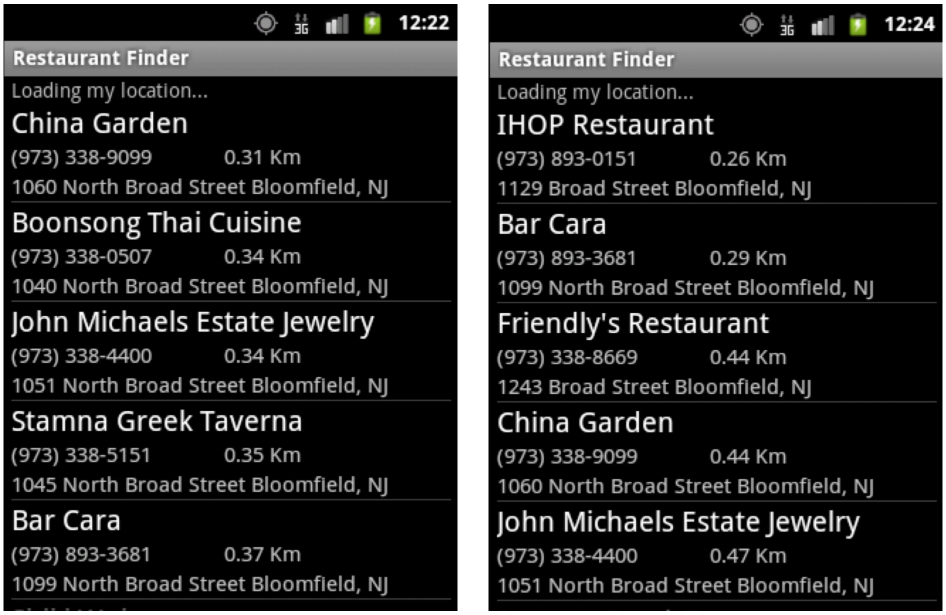
current location and the amount of truncation applied to it. Finally, we used several metrics to estimate how each truncation affected the app's utility.

2.2.3.1 Locations and truncations

In a pilot study, in which we examined the results of CloakDroid on a variety of apps, locations, and truncations in an ad hoc manner, we noticed that truncation has quite different effects depending on where apps are run. For example, there is a significantly higher density of restaurants in a big city than in a rural area, causing one of our subject apps, Restaurant Finder, to behave very differently under truncation in each locale. Thus, in selecting locations for our study, we chose them from a range of population centers of varying sizes, as shown in Figure 2.3b.

We modeled each city as a circle centered on the geographical city center with a radius estimated from looking at a map of the region. In our pilot study, we also found that many apps produce error screens if given a location that cannot be mapped to a real address, e.g., if the current location is in the ocean or a heavily forested area. Thus, we discarded any random points that caused problematic outputs from our apps, and randomly picked new points until we had a set of 10 points that produced correct output across all apps when the apps were run with no truncation.

For each random location, we ran each app with 10 different truncation amounts (including no truncation), listed in Figure 2.3c. We chose an upper limit of 50 km because many of our subject apps give results up to or less than 50 km, so the


12:22

Restaurant Finder

Loading my location...

China Garden
(973) 338-9099 0.31 Km
1060 North Broad Street Bloomfield, NJ

Boonsong Thai Cuisine
(973) 338-0507 0.34 Km
1040 North Broad Street Bloomfield, NJ

John Michaels Estate Jewelry
(973) 338-4400 0.34 Km
1051 North Broad Street Bloomfield, NJ

Stamna Greek Taverna
(973) 338-5151 0.35 Km
1045 North Broad Street Bloomfield, NJ

Bar Cara
(973) 893-3681 0.37 Km
1099 North Broad Street Bloomfield, NJ

12:24

Restaurant Finder

Loading my location...

IHOP Restaurant
(973) 893-0151 0.26 Km
1129 Broad Street Bloomfield, NJ

Bar Cara
(973) 893-3681 0.29 Km
1099 North Broad Street Bloomfield, NJ

Friendly's Restaurant
(973) 338-8669 0.44 Km
1243 Broad Street Bloomfield, NJ

China Garden
(973) 338-9099 0.44 Km
1060 North Broad Street Bloomfield, NJ

John Michaels Estate Jewelry
(973) 338-4400 0.47 Km
1051 North Broad Street Bloomfield, NJ

Figure 2.4: Restaurant Finder without (left) and with (right) 0.2 km of truncation.

metrics would be undefined beyond that level.

2.2.4 Metrics used for evaluation

Recall from Section 2.2.1 that we identified apps that list nearby objects as the best candidate for our study of location truncation. For example, Figure 2.4 gives two screenshots of the Restaurant Finder app being run on a location in New York, NY (close to Bloomfield, NJ) both without and with truncation. Here the lists we consider are the restaurants, in order from closest to farthest, along with their distances.

We explored three ways to measure location truncation's effect on the output list of such apps: edit distance of the changed list from the original, the size of the intersection of the original and changed lists, and the additional distance of the first (closest) list item on the changed list.

2.2.4.1 Edit distance

The first metric we investigated is *edit distance*, which is the number of edits (insertions, deletions, or swaps) needed to change one list into another [41]. For example, the edit distance between the lists in Figure 2.4 is five, because every item in the list has changed.

Like the other apps we studied, Restaurant Finder returns more than one screenful of objects. Thus, in performing our measurements, we need to decide how much of the app's output to use. We could pick only the first screen of data, but this restricts edit distance to just a few possible values. We could pick all the results, but that has the disadvantage that there could be many changes in the long tail of a list, yet few users would bother looking that far. Thus, as a compromise, we opted to compute edit distance over the first four screens of an app, which we think covers typical usage patterns while providing useful data. In each of the apps we tested, this corresponded to the first twenty list elements.

2.2.4.2 Set intersection size

Edit distance was the first metric we thought of, but when we tried measuring it, we found it to be problematic: Especially on the first screen of data, many locations are very close, e.g., on the left of Figure 2.4, distances range from 0.31 km to 0.37 km. Yet edit distance would count reorderings of those locations in the metric.

Thus, we also explored using set intersection size as a metric. We ignore ordering and compute how many objects are in both the original and changed list. For example, in Figure 2.4, the set intersection has size three. As with edit distance, we compute the intersection on the list displayed across the first four screens of output.

The set intersection metric is also more natural than edit distance in that it corresponds directly to a common user task of checking whether some particular object is nearby. For example, we might use Restaurant Finder to ask, is there any McDonalds nearby, or we might use GasBuddy to look for the closest Shell station.

2.2.4.3 Additional distance incurred

Ideally, we would like to measure how actual users' behavior is affected by location truncation, but this is difficult to do without a large number of human participants. Instead, we developed an *additional distance* metric that computes how much farther a user who always picks the first list item would go given the changed list compared to the original list.

For example, in Figure 2.4, the closest restaurant is actually China Garden, 0.31 km away. However, under truncation, the closest restaurant appears to be an IHOP. While the IHOP's reported distance is 0.26 km, that is the result from the truncated location—we need to look at the original output list to find out how far away the restaurant actually is. In this case, IHOP appears on the second screen of the original list, and it is 0.42 km away. For this example, then, the additional distance is $0.42 - 0.31 = 0.11$ km.

Restating this more generally, let $D(X)$ be the distance to object X on the original list. Then the additional distance of a changed list is $\Delta = D(\textit{First original}) - D(\textit{First changed})$, where *First original* is the first item of the original list and *First changed* is the first item of the changed list. Notice that this metric may be undefined if *First changed* does not appear on the original list. To reduce the chances of this, we consider all output screens of an app in computing the metric, and not just the first four screens. However, in our experiments the metric is still sometimes undefined for some apps under large truncation amounts.

2.2.4.4 Testing infrastructure

To compute the results of our experiments, we needed to run each subject app at 60 locations and 10 different truncation amounts. This is clearly far too many configurations to run by hand. Instead, we developed testing infrastructure to let us run each configuration automatically, scrape the resulting output lists, and then run our metrics on the result.

The core technology we used is Troyd [42], an open source, black box testing framework that can execute apps (e.g., launch them, click buttons, enter text boxes, scroll the screen, etc.) and gather text from the GUI. Troyd allows us to write a high level test case (e.g., to click a sequence of buttons and gather some text) that can be reused for various test configurations. We developed a server that takes an app modified by CloakDroid; resigns it with a shared user ID so that it can be controlled by instrumentation; and then runs testcases against the app using Troyd.

We ran apps on both Google Nexus S devices and on device emulators; our infrastructure works the same in both cases, and performing some runs on actual devices acted as a sanity check. In all cases, the Android platform was Gingerbread 2.3.3 with the Google APIs, such as maps, installed. These APIs are special dynamic libraries licensed by Google and not included on plain Android distributions; these APIs are needed for the subject programs to run.

We found that, while Troyd mostly did what we wanted, we needed to extend it to handle additional GUI elements. For example, while Troyd can capture the currently visible GUI elements, many Views (such as those in ListView elements) are reused, and their contents only appear when the screen is scrolled. Thus, we extended Troyd with special functionality to navigate to screens that may have unrendered information. We also found that, as is usual in large-scale testing, experimental tests sometimes failed unpredictably, e.g., an emulator or device would crash for no apparent reason. These are likely due to subtle bugs lurking in the OS or emulator implementation that were tickled by the experiment's atypical usage pattern. Thus, our testing infrastructure catches any failed runs and reschedules those experiments to be run again. Finally, our testing infrastructure allows multiple experiments to be done in parallel; the majority of the experiments were done on a machine with six Intel Xeon processors, each with four cores, and 48 GB of RAM. Each individual test took on the order of a few minutes to run, depending on the app. Our tests ran up to fifteen emulators at once.

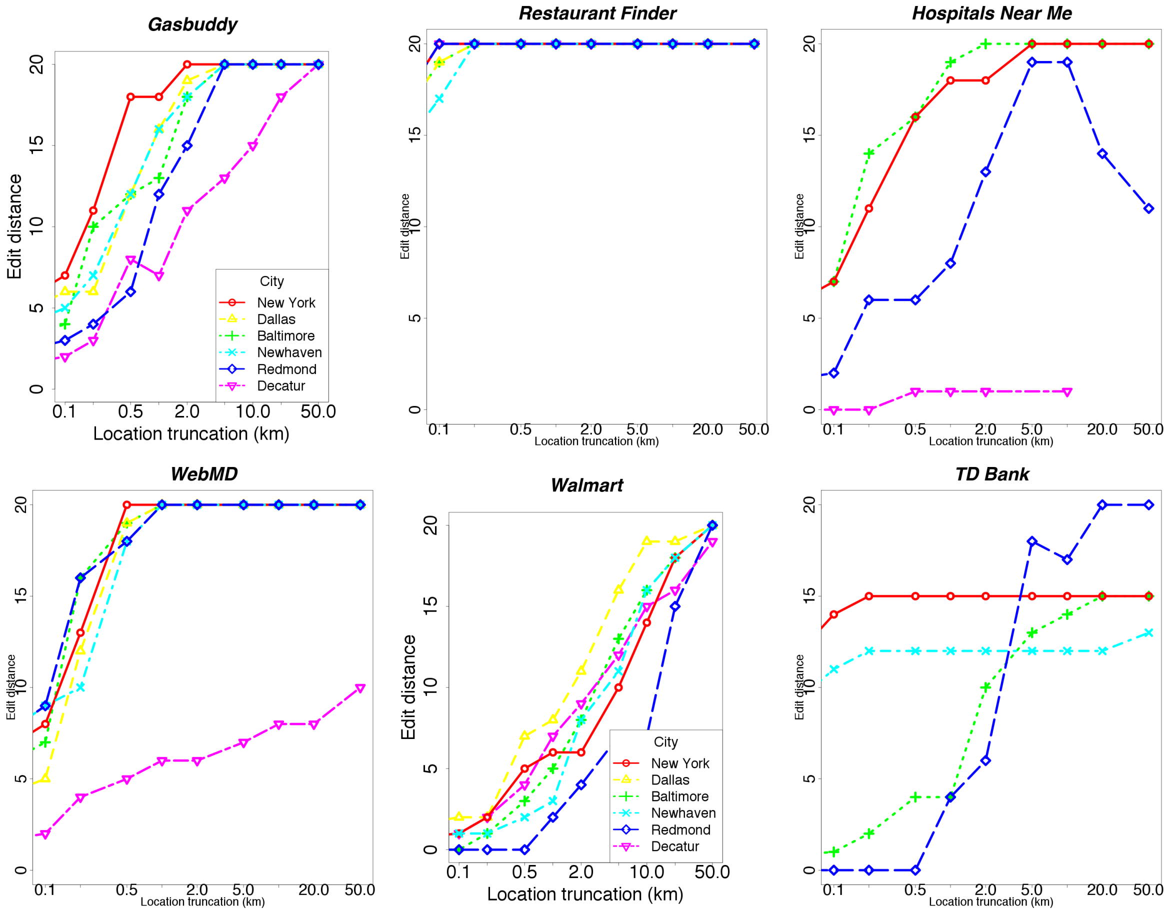


Figure 2.5: Graphs of median edit distance versus truncation amount. Higher edit distance implies lower utility.

2.2.5 Results

Finally, we present the results of our study. At a high level, we found that across our subject apps and locations, location can be significantly truncated without significantly harming utility: up to 20 km when an app is used in a lower population area, and usually (for five of six apps) at least 5 km in more populous areas. We also found that apps have relatively little change in utility up to a certain amount of truncation, typically about 5 km, after which the app's output becomes much less usable. Finally, we found that the factor that most determines the ability to truncate location was the density of the set of locations the app computes over.

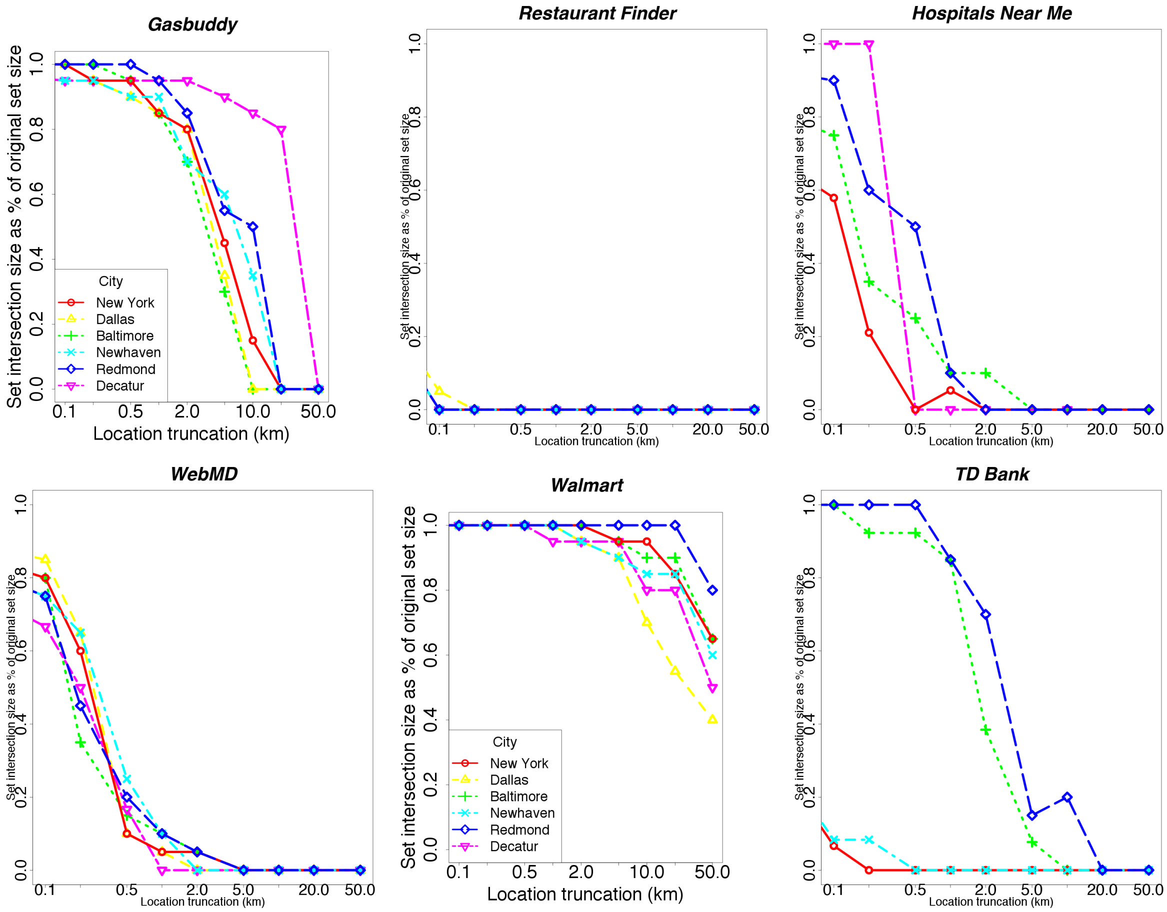
In the discussion that follows, we include several plots. In each plot, the x-axis is the truncation amount; note that because of our choice of truncations, this axis is essentially log-scale. The y-axis is the median of one of our metrics on the 10 randomly chosen points for each population center.

We next consider each of our three metrics in turn.

2.2.5.1 Edit Distance

Figure 2.5 shows how edit distance varies on our location set. Note that Hospitals Near Me and TD Bank do not have data for cities for which the app did not generate enough output to calculate a significant result. For example, there are no TD banks in Dallas. In some cases, TD Bank and Hospitals Near Me do generate data, but they generate a very small set of results (Hospitals Near Me only returns hospitals within 50 miles, for example). Thus, edit distance appears to go down at large truncation values because in less populous locations they output smaller sets of locations.

The plots show that in most cases, the output lists change to some degree even at the smallest truncation level. Moreover, in two cases (Gasbuddy and Restaurant Finder), the edit distance quickly reaches its maximum value (all items in the list change) in several locales. WebMD is similar, though it plateaus somehwat later. The remaining three apps, in contrast, show a generally steady progression of edit distance versus truncation amount.



(a) Graphs of median set intersection size versus truncation amount. Lower set intersection size indicates lower utility.

	New York	Dallas	Baltimore	New Haven	Redmond	Decatur
Gasbuddy	5	5	2	2	5	50
Restaurant Finder	1	0.5	0.1	0.1	0.1	0.2
Walmart	50	10	50	50	·	50
TD Bank	10	·	50	20	·	·
WebMD	10	5	2	10	10	50
Hospitals Near Me	5	·	20	·	20	·

(b) Largest truncation amount (km) before which the median set intersection size is lower than 80%.

Figure 2.6: Set intersection size results.

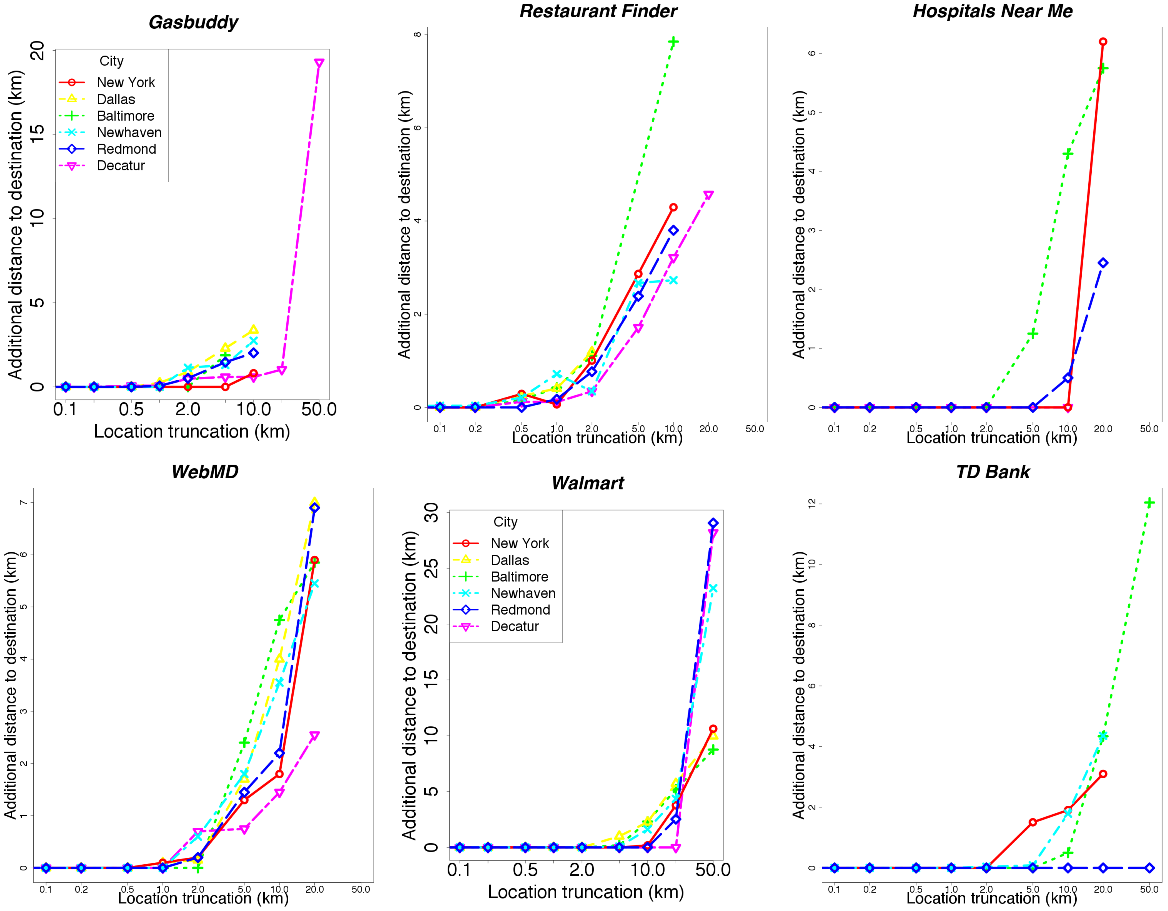
The reason for these trends is density of objects–Gasbuddy, Restaurant Finder, and WebMD all return lists of items that can commonly be found everywhere, whereas there may be only a few hospitals, Walmarts, or TD Bank locations. Indeed, looking at Gasbuddy, we see that edit distance plateaus quickest for New York, the most populous locale, and slowest for Decatur, the least populous locale.

One problem with the edit distance metric is that even insignificant reordering of the output list adds to the distance—yet users likely will not care about the exact ordering as long as relevant results appear within the first few items of the nominal list. Additionally, edit distance does not have a clear physical interpretation, nor does it seem to correspond to any typical tasks a user might want to perform. Thus, while it provides insight into app behavior under truncation, we think that edit distance is not the best metric of utility.

2.2.5.2 Set Intersection

Figure 2.6a shows the results for the set intersection size metric, with the y-axis the percentage of the set intersection size compared to the original set size. Figure 2.6b shows the truncation amount after which the set intersection percentage drops below 80% of the original value. More precisely, let $I(s)$ be the median intersection set size under truncation s, defined as $|L \cap L'|/|L|$ where L is the list without truncation and L' is the list with truncation s. Then the chart lists the s_i such that $I(s_i) \leq 0.8$ and $I(s_{i+1}) > 0.8$, where s_j is the sequence of location truncation amounts.

We observe that apps measuring less densely distributed locations can be truncated more aggressively than those measuring more densely distributed locations. For example, we see that Gasbuddy and Restaurant Finder, which both measure



(a) Graph of median additional distance versus truncation amount. Higher additional distance implies lower utility.

	New York	Dallas	Baltimore	New Haven	Redmond	Decatur
Gasbuddy	·	5	5	2	5	20
Restaurant Finder	2	2	2	5	5	5
Walmart	20	5	10	10	20	50
TD Bank	5	·	20	10	·	·
WebMD	5	5	5	5	5	10
Hospitals Near Me	20	·	5	·	20	·

(b) Largest truncation amount (km) before median additional distance exceeds 1 km.

Figure 2.7: Additional distance results.

densely distributed objects, can sustain some truncation, but their utility drops off much faster than the other apps such as Hospitals Near Me. In the table, we see this again in that Gasbuddy and Restaurant Finder drop below a set intersection size of 80% at a lower truncation amount compared to Hospitals Near Me.

One slightly surprising trend in the table is that for Restaurant Finder, location can be truncated more in New York than in other location, whereas we would expect the opposite due to its high density of restaurants. Looking into the results in detail, we discovered this occurred because of the large radius (20 km) we chose for New York: Restaurants are typically clustered in the city center and we chose locations randomly within the radius, so a large radius reduced restaurant density.

2.2.5.3 Additional Distance

Figure 2.7a shows the results for the additional distance metric applied to our subject apps. Figure 2.7b shows, for each city and each app, the last truncation amount before median additional distance degrades beyond 1 km. In other words, if we let $\Delta(s)$ be the median additional distance under truncation s, the chart lists the s_i such that $\Delta(s_i) \leq 1\ km$ and $\Delta(s_{i+1}) > 1\ km$.

Note that the additional distance is undefined if a location truncation produces a new closest location that was not present in the original output list, as in this case we cannot compute how far the new closest location is from the current location. In the charts, undefined additional distances are omitted, as thus some lines end before reaching the maximum truncation. This also limits the number of points on which we can compute the cutoff. For example, the Gasbuddy app never causes the user to go more than 1 km out of the way in New York, so it is listed as “·” in Figure 2.7b.

From these plots, we see that under the additional distance metric, there is typically no change in app utility up to 1 or 2 km truncation. Notice that this is quite different behavior than under the edit distance metric, where the edit distance is nonzero even with small amounts of location truncation.

In most cases, the lack of change in additional distance is meaningful, but in a few cases it only reveals the low density of located objects. In particular, in Redmond, the TD Bank app can sustain location truncation up to 50 km without any change, but this occurs because the first result on the list is over 100 km away. This also happens in the Hospitals Near Me app, as there are relatively few hospitals in any area.

Looking at Figure 2.7b, we see that if a user is willing to go 1 km out of their way, then in most locations apps can sustain truncation amounts of 5 km or more. However, as with edit distance the effect is highly dependent on object density, e.g., Gasbuddy and Restaurant Finder are limited to 5 km truncation (except in Decatur, which has fewer gas stations), whereas other apps can be truncated to higher levels.

2.2.5.4 Discussion

Each of the three metrics applies in a different situations: edit distance for when an exact order is desired, set intersection for when a user wants to find any number of locations close by, and additional distance for when a user wants to find a nearest location to visit. The general trend for all three are similar: the metrics are much more forgiving in less dense areas, which makes intuitive sense. In terms of absolute values of truncation that can be sustained, we see that edit distance is the least permissive, as very small changes have large effects. (We did not include a table like Figure 2.7b or 2.6b for edit distance because we have no intuition of what a reasonable cutoff would be.) The next most restrictive metric is set intersection size, and the most permissive metric is additional distance.

2.3 Related Work

Several other researchers have proposed mechanisms to refine or reduce permissions in Android. MockDroid allows users to replace an app's view of certain private data with fake information [43]. Apex is similar, and also lets the user enforce simple constraints such as the number of times per day a resource may be accessed [44]. TISSA gives users detailed control over an app's access to selected private data (phone identity, location, contacts, and the call log), letting the user decide whether the app can see the true data, empty data, anonymized data, or mock data [45]. AppFence similarly lets users provide mock data to apps requesting private information, and can also ensure private data that is released to apps does not leave the device [46]. A limitation of all of these approaches is that they require modifications to the Android platform, and hence to be used in practice must either be adopted by Google or device providers, or must be run on rooted phones. In contrast, Dr. Android and Mr. Hide run on stock, unmodified Android systems available today.

Researchers have also developed other ways to enhance Android's overall security framework. Kirin employs a set of user-defined security rules to flag potential malware at install time [47]. Saint enriches permissions on Android to support a variety of installation constraints, e.g., a permission can include a whitelist of apps that may request it [48]. These approaches are complementary to our system, as they take the platform permissions as is and do not refine them.

There have been several studies of Android's permissions, sensitive APIs, and the use of permissions across apps. Barrera et al. [49] analyze the way permissions are used in Android apps, and observe that only a small number of Android permissions are widely used but that some of these, in particular Internet permissions, are overly broad (as we have also found). Vidas et al. [50] describe a tool that, using documentation-derived information, can statically analyze an app's source code to find a minimum set of permissions it needs. Stowaway [51] performs a static analysis on the Android API itself to discover which APIs require what permissions, something they found is not always well documented. (We used Stowaway's data set in several cases to help determine what adapters we needed to implement in Mr. Hide.)

Finally, several tools have been developed that look for security issues in Android apps. TaintDroid tracks the flow of sensitive information [52]. Ded [53], a Dalvik-to-Java decompiler, has been used to discover previously undisclosed device identifier leaks. ComDroid [54] finds vulnerabilities related to Intent handling. Felt et al. [55] study the problem of permission redelegation, in which an app is tricked into providing sensitive capabilities to another app. Woodpecker [56] uses dataflow analysis to find capability leaks on Android phones. All of these tools focus on improper use of the current set of Android's permissions. Dr. Android and Mr. Hide take a complimentary approach, replacing existing permissions with finer-grained ones to reduce or eliminate consequences of security issues.

Related Work for Location Privacy There is a large body of work on increasing location privacy, though we are aware of no other work that directly measures the resulting utility of mobile Android apps.

The Caché system [57] caches and prefetches content from a server, obfuscating the user's location at the cost of potentially stale content and higher bandwidth. Additionally, application writers must specify rules for caching up front. The system caches data for a set of regions, quantizing the user's location as in our approach. The authors note that app utility will be impacted by their technique, but they do not measure it explicitly.

Hornyack et al. [46] study apps that use sensitive information (such as location), and present AppFence, a system that can fuzz inputs to apps. While the AppFence authors note the effects of fuzzing app inputs on usability, they do not study it systematically using any particular metric.

Shokri et al. [30] present a systematic approach to quantifying Location Privacy Preserving Mechanisms (LPPMs). They also describe an meter on the user's devise that continuously informs the user of their privacy. In our work, we implemented a simple truncation-based LPPM (corresponding to a technique they call *precision reduction*), and use this to study utility as a function of the degree of truncation. As future work, we may consider evaluating the utility of other policies from their system.

In follow up work, Shokri et al. [31] present an optimal strategy to prevent localization attacks. Their analysis formulates location privacy as a Bayesian Stackelberg game, where a user chooses a location cloaking strategy without knowing their potential adversary. While their analysis considers service quality, they use metrics that do not clearly map to utility, such as Euclidean distance from the true location (rather than the actual effect of the change on the service).

Our study focuses on privacy for a single user at a stationary point. Allowing the app to collect traces of data reveals much more information [37], [58]. Much of the existing work on location privacy [29], [32], [33], [34] focuses on k-anonymity: if a user is making a query to a location based service, they can only be identified to be within a set of k potential users. One popular technique is to use mix zones [29]: once a user enters a designated area their location information becomes "mixed" with others in that same area. This technique requires a trusted middleware layer (to properly mix location data) and requires these mix zones to be defined. In contrast, location truncation can be done locally on a mobile device.

None of these previous approaches study the impact of location privacy on app utility. The work that comes closest is by Shokri et al. [31], but while that framework considers utility as an element of their models, it is not directly measured on apps. Our work is complementary: we focus not on optimal obfuscation techniques, but rather we fix an obfuscation strategy and study how utility changes under that technique. As future work, we intend to couple our empirical utility functions with the theoretical models presented by Shokri et al. Doing so would allow us to determine bounds for k and allow us to study how the optimal strategies presented by Shokri et al. are affected.

2.4 Conclusion

We presented an empirical study of how location truncation affects mobile app utility. We began by examining how Android apps use location. Across the apps we looked at, we found that the second most common pattern is using the current location to list various sorts of nearby objects. The most common pattern, location-targeted ads, is not amenable to evaluating utility. We then used Dr. Android and Mr. Hide to implement CloakDroid, a tool that modifies existing apps' bytecode to use truncated location information. We designed an experiment in which we measured the effect of a range of location truncations (from 0 km to 50 km) on 60 points randomly chosen from various locales (from population 6,000 Decatur, TX to population 8.2 million New York, NY). We identified three metrics that approximate app utility: edit distance, set intersection size, and additional distance. We found that, under these metrics, the factor that most determines the utility–truncation tradeoff is the density of objects being returned by the app, and that in many cases, location can be truncated significantly without losing much utility. To our knowledge, our work provides the first end-to-end evaluation of how location truncation affects Android app utility. As subject of future work, we plan to apply CloakDroid to a larger number of apps and test the metrics using user studies.

Chapter 3: Checking Interaction-Based Declassification Policies for Android Using Symbolic Execution

The last chapter introduced Redexer, a tool that allows binary rewriting of Android apps. Redexer allows us to change the dynamic behavior of an Android app, e.g., to change the way it uses permissions. However, manipulating the way an app accesses information does not allow us to understand what is done with the information once it enters the app. In this chapter, I describe how to check *information flow* properties using a static analysis. In doing so, I detail a novel policy framework that allows reasoning about how UI interactions can be used to declassify information.

3.1 Introduction

The Android platform includes a *permission* system that aims to prevent apps from abusing access to sensitive information, such as contacts and location. Unfortunately, once an app is installed, it has *carte blanche* to use any of its permissions in arbitrary ways at run time. For example, an app with location and Internet access could continuously broadcast the device's location, even if such behavior is not expected by the user.

To address this limitation, I present a new framework for Android app security based on *information flow control* [11] and user interactions. The key idea behind our framework is that users naturally express their intentions about information release as they interact with an app. For example, clicking a button may permit an app to release a phone number over the Internet. Or, as another example, toggling a radio button from "coarse" to "fine" and back to "coarse" may temporarily permit an app to use fine-grained GPS location rather than a coarse-grained approximation.

To model these kinds of scenarios, we introduce *interaction-based declassification policies*, which extensionally specify what information flows may occur after which sequences of *events*. Events are GUI interactions (e.g., clicking a button), inputs (e.g., reading the phone number), or outputs (e.g., sending over the Internet). A policy is a set of *declassification conditions*, written $\phi \rhd S$, where ϕ is a linear-time temporal logic (LTL) [59] formula over events, and S is a sensitivity level. If ϕ holds at the time an input occurs, then that input is declassified to level S. We formalize a semantic security condition, *interaction-based noninterference* (IBNI), over sets of event *traces* generated by an app. Intuitively, IBNI holds of an app and policy if observational determinism [60] holds after all inputs have been declassified according to the policy. (Section 5.2 describes policies further, and Section 3.3 presents our formal definitions.)

We introduce ClickRelease, a static analysis tool to check whether an Android app and its declassification policy satisfy IBNI. ClickRelease generates event traces using SymDroid [21], a Dalvik bytecode symbolic executor. ClickRelease works by simulating user interactions with the app and recording the resulting execution traces. In practice, it is not feasible to enumerate all program traces, so ClickRelease generates traces up to some *input depth* of n GUI events. ClickRelease then synthesizes a set of logical formulae that hold if and only if IBNI holds, and uses Z3 [61] to check their satisfiability. (Section 5.4 describes ClickRelease in detail.)

To validate ClickRelease, we used it to analyze four Android apps, including both secure and insecure variants of those apps. We ran each app variant under a range of input depths, and confirmed that, as expected, ClickRelease scales exponentially. However, we manually examined each app and its policy, and found that an input depth of at most 5 is sufficient to guarantee detection of a security policy violation (if any) for these cases. We ran ClickRelease at these minimum input depths and found that it correctly passes and fails the secure and insecure app variants, respectively. Moreover, at these depths, ClickRelease takes just a few seconds to run. (Section 3.5 describes our experiments.)

In summary, we believe that ClickRelease takes an important step forward in providing powerful new security mechanisms for mobile devices. We expect that our approach can also be used in other GUI-based, security-sensitive systems.

3.2 Example Apps and Policies

We begin with two example apps that show interesting aspects of interaction-based declassification policies.

Bump app. The boxed portion of Fig. 3.1 gives (simplified) source code for an Android app that releases a device's unique ID and/or phone number. This app is inspired by the Bump app, which let users tap phones to share selected information with each other. We have interspersed an insecure variant of the app in the red code on lines 14 and 16, which we will discuss in Section 3.3.1.

Each screen of an Android app is implemented using a class that extends Activity. When an app is launched, Android invokes the onCreate method for a designated main activity. (This is part of the *activity lifecycle* [62], which includes several methods called in a certain order. For this simple app, and the other apps used in this chapter, we only need a single activity with this one lifecycle method.) That method retrieves (lines 3–5) the GUI IDs of a button (marked "send") and two checkboxes (marked "ID" and "phone"). The onCreate method next gets an instance of the TelephonyManager, uses it to retrieve the device's unique ID and phone number information, and unchecks the two checkboxes as a default. Then it creates a new callback (line 11) to be invoked when the "send" button is clicked. When called, that callback releases the user's ID and/or phone number, depending on the checkboxes.

This app is written to work with ClickRelease, a symbolic execution tool we built to check whether apps satisfy interaction-based declassification policies. As we discuss further in Section 5.4, ClickRelease uses an executable model of Android that abstracts away some details that are unimportant with respect to security. While a real app would release information by sending it to a web server, here we instead call a method Internet.sendInt. Additionally, while real apps include an XML file specifying the screen layout of buttons, checkboxes, and so on, ClickRelease creates those GUI elements on demand at calls to findViewById (since their screen locations

```
public class BumpApp extends Activity {
  protected void onCreate(...) {
    Button sendBtn = (Button) findViewById(...);
    CheckBox idBox = (CheckBox) findViewById(...);
    CheckBox phBox = (CheckBox) findViewById(...);
    TelephonyManager manager = TelephonyManager.getTelephonyManager();
    final int id = manager.getDeviceId();
    final int ph = manager.getPhoneNumber();
    idBox.setChecked(false); phBox.setChecked(false);
    sendBtn.setOnClickListener(
    new OnClickListener() {
      public void onClick(View v) {
        if (idBox.isChecked())
        Internet.sendInt(id); //Internet.sendInt(ph);
        if (phBox.isChecked())
        Internet.sendInt(ph); //Internet.sendInt(id);
    }})}}
```

$$\mathsf{id}! * \wedge(\mathcal{F}(\mathsf{sendBtn!unit} \wedge \mathit{last}(\mathsf{idBox}, \mathsf{true}))) \rhd \mathit{Low},$$
$$\mathsf{ph}! * \wedge(\mathcal{F}(\mathsf{sendBtn!unit} \wedge \mathit{last}(\mathsf{phBox}, \mathsf{true}))) \rhd \mathit{Low}$$

Figure 3.1: "Bump" app and policy.

are unimportant). Finally, we model the ID and phone number as integers to keep the analysis simpler.

ClickRelease symbolically executes paths through subject apps, recording a *trace* of *events* that correspond to certain method calls. For example, one path through this app generates a trace

$$\mathsf{id!42}, \mathsf{ph!43}, \mathsf{idBox!true}, \mathsf{sendBtn!unit}, \mathsf{netout!42}$$

Each event has a *name* and a *value*. Here we have used names id and ph for secret inputs, idBox and sendBtn for GUI inputs, and netout for the network send. In particular, the trace above indicates 42 is read as the ID, 43 is read as the phone number, the ID checkbox is selected, the send button is clicked (carrying no value, indicated by unit), and then 42 is sent on the network. In ClickRelease, events are generated by calling certain methods that are specially recognized. For example, ClickRelease implements the manager.getDeviceId call as both returning a value and emitting an event.

Notice here that in the trace, callbacks to methods such as idBox and sendBtn correspond to user interactions. The key idea behind our framework is that these actions convey the user's intent as to which information should be released. Moreover, traces also contain actions relevant to information release—here the reads of the ID and phone number, and the network send. Thus, putting both user interactions and security-sensitive operations together in a single trace allows our policies to enforce the user's intent.

The policy for this example app is shown at the bottom of Fig. 3.1. Policies are comprised of a set of *declassification conditions* of the form $\phi \triangleright S$, where ϕ is an LTL formula describing event traces and S is a security level. Such a condition is read, "At any input event, if ϕ holds at that position of the event trace, then that input is declassified at level S." For this app there are two declassification conditions. The top condition declassifies (to *Low*) an input that is a read of the ID at any value (indicated by $*$), if sometime in the future (indicated by the $\mathcal{F}$ modality) the send button is clicked and, when that button is clicked, the last value of the ID checkbox was true. (Note that *last* is not primitive, but is a macro that can be expanded into regular LTL.) The second declassification condition does the analogous thing for the phone number.

To check such a policy, ClickRelease symbolic executes the program, generating per-path traces; determines the classification level of every input; and checks that every pair of traces satisfies noninterference. Note that using LTL provides a very general and expressive way to describe the sequences of events that imply declassification. For example, here we precisely capture that only the last value of the checkbox matters for declassification. For example, if a user selects the ID checkbox but then unselects it and clicks send, the ID may not be released.

Although this example relies on a direct flow, ClickRelease can also detect implicit flows. Section 3.3.2 defines an appropriate version of noninterference, and the experiments in Section 3.5 include a subject program with an implicit flow.

Notice this policy depends on the app reading the ID and phone number when the app starts. If the app instead waited until after the send button were clicked, it would violate this policy. We could address this by replacing the $\mathcal{F}$ modality by $\mathcal{P}$ (past) in the policy, and we could form a disjunction of the two policies if we wanted to allow either implementation. More generally, we designed our framework to be sensitive to such choices to support reasoning about secret values that change over time. We will see an example next.

Location resolution toggle app. Fig. 3.2 gives code for an app that shares location information, either at full or truncated resolution depending on a radio button setting. The app's onCreate method displays a radio button (code not shown) and then creates and registers a new instance of RadioManager to be called each time the radio button is changed. That class maintains field mFine as true when the radio button is set to full resolution and false when set to truncated resolution.

```
public class ToggleRes extends Activity { ...
  LocSharer mLocSharer = new LocSharer();
  RadioManager mRadio = new RadioManager();
  protected void onCreate(...) { ...}
  private class LocSharer implements LocationListener { ...
    public LocSharer(RadioManager rm) {
      lm = (LocationManager) getSystemService(LOCATION_SERVICE);
      lm.requestLocationUpdates(mCurrentProvider, SHARE_INTERVAL, distance, this);
    }
    public void onLocationChanged(Location l) {
      if (mRadio.mFine) {
        Internet.sendInt(l.mLatitude);
        Internet.sendInt(l.mLongitude);
      } else {
        Internet.sendInt(l.mLatitude & 0xffffff00);
        Internet.sendInt(l.mLongitude & 0xffffff00);
  } } }
  private class RadioManager
  implements OnClickListener {
    public boolean mFine = false;
    public void onClick(View v) { mFine = !mFine; }
} }
```

$$\begin{array}{l}\mathsf{longitude!} * \wedge \mathit{last}(\mathsf{mRadio}, \mathsf{true}) \rhd \mathit{Low},\\ \mathsf{longitude!} * \wedge \mathit{last}(\mathsf{mRadio}, \mathsf{false}) \rhd \mathit{MaskLower8}\end{array}$$

Figure 3.2: Location sharing app and policy.

Separately, onCreate registers LocSharer to be called periodically with the current location. It requests location updates by registering a callback with the LocationManager system service. When called, LocSharer releases the location, either at full resolution or with the lower 8 bits masked, depending on mFine.

The declassification policy for longitude appears below the code; the policy for latitude is analogous. This policy allows the precise longitude to be released when mRadio is set to fine, but only the lower eight bits to be released if mRadio is set to coarse. Here ClickRelease knows that at the *MaskLower8* level, it should consider outputs to be equivalent up to differences in the lower 8 bits.

Finally, notice that this policy does not use the future modality. This is deliberate, because location may be read multiple times during the execution, at multiple values, and the security level of those locations should depend on the state of the radio button at that time. For example, consider a trace

$$\mathsf{mRadio!false}, \mathsf{longitude!}v_1, \mathsf{mRadio!true}, \mathsf{longitude!}v_2$$

The second declassification condition ($\mathsf{longitude!} * \wedge \mathit{last}(\mathsf{mRadio}, \mathsf{false})$) will match the event with v_1, since the last value of mRadio was false, and thus v_1 may be declassified only to *MaskLower8*. Whereas the first declassification condition will match the event with v_2, hence it may be declassified to *Low*.

3.3 Program Traces and Security Definition

Next, we formally define when a set of program traces satisfies an interaction-based declassification policy.

3.3.1 Program Traces

Fig. 3.3(a) gives the formal syntax of events and traces. *Primitives* p are terms that can be carried by events, e.g., values for GUI events, secret inputs, or network sends. In our formalism, primitives are integers, booleans, and terms constructed from primitives using uninterpreted constructors f. As programs execute, they produce a *trace* t of *events* η, where each event $\mathit{name}!p$ pairs an event name *name* with a primitive p. We assume event names are partitioned into those corresponding

$$\begin{array}{lcll}
\text{Primitives} & p & ::= & n \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{unit} \mid f(p_1, \ldots, p_i) \\
\text{Events} & \eta & ::= & \mathit{name}!p \\
\text{Traces} & t & ::= & \eta\ \mathit{list}
\end{array}$$

(a) Event and Trace Definitions.

$$\begin{array}{lcll}
\text{Policies} & P & ::= & C_1, C_2, \ldots \\
\text{Conditions} & C & ::= & \phi \rhd S \\
\text{Security Levels} & S & ::= & \mathit{High} \mid \mathit{Low} \mid \mathit{MaskLower8} \mid \ldots \\
\text{Atoms} & A & ::= & \mathit{name}!s \mid s \oplus s \\
\text{Messages} & s & ::= & x \mid p \mid * \\
\text{Formulae} & \phi & ::= & A \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \rightarrow \phi \mid \exists x.\phi \mid \forall x.\phi \\
 & & \mid & \mathcal{X}\phi \mid \phi\ \mathcal{U}\ \phi \mid \mathcal{G}\phi \mid \mathcal{F}\phi \mid \phi\ \mathcal{S}\ \phi \mid \mathcal{P}\phi
\end{array}$$

(b) Interaction-based Declassification Policy Language.

Figure 3.3: Formal definitions.

to inputs and those corresponding to outputs. For all the examples in this chapter, all names are inputs except netout, which is an output.

Due to space limitations, we omit details of how traces are generated. These details, along with definition of our LTL formulas, can be found in a companion tech report [63]. Instead, we simply assume there exists some set $\mathcal{T}$ containing all possible traces a given program may generate. For example, consider the insecure variant bump app in Fig. 3.1, which replaces the black code with the red code on lines lines 14 and 16. This app sends the phone number when the email box is checked and vice-versa. Thus, its set $\mathcal{T}$ contains, among others, the following two traces:

$$\mathsf{id!0, ph!0, idBox!true, sendBtn!unit, netout!0} \quad (1)$$

$$\mathsf{id!0, ph!1, idBox!true, sendBtn!unit, netout!1} \quad (2)$$

In the first trace, ID and phone number are read as 0, the ID checkbox is selected, the button is clicked, and 0 is sent. The second trace is similar, except the phone number and sent value are 1. Below, we use these traces to show this program violates its security policy.

3.3.2 Interaction-based Declassification Policies

We now define our policy language precisely. Fig. 3.3(b) gives the formal syntax of declassification policies. A policy P is a set of *declassification conditions* C_i of the form $\phi_i \rhd S_i$, where ϕ_i is an LTL formula describing when an input is declassified, and S_i is a *security level* at which the value in that event is declassified.

As is standard, security levels S form a lattice. For our framework, we require that this lattice be finite. We include *High* and *Low* security levels, and we can generalize to arbitrary lattices in a straightforward way. Here we include the *MaskLower8* level from Fig. 3.2 as an example, where $\mathit{Low} \sqsubseteq \mathit{MaskLower8} \sqsubseteq \mathit{High}$. Note that although we include *High* in the language, in practice there is no reason to declassify something to level *High*, since then it remains secret.

The *atomic predicates* A of LTL formulae match events, e.g., atomic predicate $\mathit{name}!p$ matches exactly that event. We include $*$ for matches to arbitrary primitives. We allow event values to be variables that are bound in an enclosing quantifier. The atomic predicates also include atomic arithmetic statements; here $\oplus$ ranges over standard operations such as $+$, $<$, etc. The combination of these lets us describe complex events. For example, we could write $\exists x.\mathit{spinner}!x \wedge x > 2$ to indicate the *spinner* was selected with a value greater than 2.

Atomic predicates are combined with the usual boolean connectives (¬, ∧, ∨, →) and existential and universal quantification. Formulae include standard LTL modalities $\mathcal{X}$ (next), $\mathcal{U}$ (until), $\mathcal{G}$ (always), $\mathcal{F}$ (future), $\phi\ \mathcal{S}\ \psi$ (since), and $\mathcal{P}\phi$ (past). We include a wide range of modalities, rather than a minimal set, to make policies easier to write. Formulae also include $\mathit{last}(\mathit{name}, p)$, which is syntactic sugar for $\neg(\mathit{name}!*)\ \mathcal{S}\ \mathit{name}!p$. We assume a standard interpretation of LTL formulae over traces [64]. We write $t, i \models \phi$ if trace t is a model of ϕ at position i in the trace.

Next consider a trace $t \in \mathcal{T}$ for an arbitrary program. We write $\mathit{level}(t, P, i)$ for the security level that policy P assigns to the event $t[i]$:

$$\mathit{level}(t, P, i) = \begin{cases} \prod_{\phi_j \triangleright S_j \in P} \{S_j \mid t, i \models \phi_j\} & t[i] = \mathit{name}!p \\ \mathit{Low} & t[i] = \mathsf{netout}!p \end{cases}$$

In other words, for inputs, we take the greatest lower bound (the most declassified) of the levels from all declassification conditions that apply. We always consider network outputs to be declassified. Notice that if no policy applies, the level is H by definition of greatest lower bound.

For example, consider trace (1) above with respect to the policy in Fig. 3.1. At position 0, the LTL formula holds because the ID box is eventually checked and then the send button is clicked, so $\mathit{level}((1), P, 0) = \mathit{Low}$. However, $\mathit{level}((1), P, 1) = \mathit{High}$ because no declassification condition applies for ph (phBox is never checked). And $\mathit{level}((1), P, 4) = \mathit{Low}$, because that position is a network send.

Next consider applying this definition to the GUI inputs. As written, we have

$\mathit{level}((1), P, 2) = \mathit{level}((1), P, 3) = \mathit{High}$. However, our app is designed to leak these inputs. For example, an adversary will learn the state of idBox if they receive a message with an ID. Thus, for all the subject apps in this chapter, we also declassify all GUI inputs as *Low*. For the example in Fig. 3.1, this means adding the conditions $\mathsf{idBox}! * \rhd \mathit{Low}$, $\mathsf{phBox}! * \rhd \mathit{Low}$, and $\mathsf{sendBtn}! * \rhd \mathit{Low}$. In general, the security policy designer should decide the security level of GUI inputs.

Next, we can apply *level* pointwise across a trace and discard any trace elements that are below a given level S. We define

$$\mathit{level}(t, P)^S[i] = \begin{cases} t[i] & \mathit{level}(t, P, i) \sqsubseteq S \\ \\ \tau & \text{otherwise} \end{cases}$$

We write $\mathit{level}(t, P)^{S,in}$ for the same filtering, except output events (i.e., network sends) are removed as well. Considering the traces (1) and (2) again, we have

$$\begin{aligned} \mathit{level}((1), P)^{\mathit{Low}} &= \mathsf{id!}0, \mathsf{idBox!true}, \mathsf{sendBtn!unit}, \mathsf{netout!}0 \\ \mathit{level}((2), P)^{\mathit{Low}} &= \mathsf{id!}0, \mathsf{idBox!true}, \mathsf{sendBtn!unit}, \mathsf{netout!}1 \\ \mathit{level}((1), P)^{\mathit{Low},in} &= \mathsf{id!}0, \mathsf{idBox!true}, \mathsf{sendBtn!unit} \\ \mathit{level}((2), P)^{\mathit{Low},in} &= \mathsf{id!}0, \mathsf{idBox!true}, \mathsf{sendBtn!unit} \end{aligned}$$

Finally, we can define a program to satisfy noninterference if, for every pair of traces such that the inputs at level S are the same, the outputs at level S are also the same. To account for generalized lattice levels such as *MaskLower8*, we also need to treat events that are equivalent at a certain level as the same. For example, at *MaskLower8*, outputs `0xffffffff` and `0xffffff00` are the same, since they do not differ in the upper 24 bits. Thus, we assume for each security level S there is a appropriate equivalence relation $=_S$, e.g., for *MaskLower8*, it compares elements ignoring their lower 8 bits. Note that $x =_{Low} y$ is simply $x = y$ and $x =_{High} y$ is always true.

Definition 1 (Interaction-based Noninterference (IBNI)) *A program satisfies security policy P, if for all S and for all $t_1, t_2 \in \mathcal{T}$ (the set of traces of the program) the following holds:*

$$\mathit{level}(t_1, P)^{S,in} =_S \mathit{level}(t_2, P)^{S,in} \implies \mathit{level}(t_1, P)^S =_S \mathit{level}(t_2, P)^S$$

Looking at traces for the insecure app, we see they violate non-interference, because $\mathit{level}((1), P)^{Low,in} = \mathit{level}((2), P)^{Low,in}$, but $\mathit{level}((1), P)^{Low} \neq \mathit{level}((2), P)^{Low}$ (they differ in the output). We note that our definition of noninterference makes it a 2-hypersafety property [65, 66].

3.4 Implementation

We built a prototype tool, ClickRelease, to check whether Android apps obey the interaction-based declassification policies described in Section 3.3. ClickRelease is based on SymDroid [21], a symbolic executor for Dalvik bytecode, which is the bytecode format to which Android apps are compiled. As is standard, SymDroid computes with *symbolic expressions* that may contain *symbolic variables* representing sets of values. At conditional branches that depend on symbolic variables, SymDroid invokes Z3 [61] to determine whether one or both branches are feasible. As it follows branches, SymDroid extends the current *path condition*, which tracks branches taken so far, and forks execution when multiple paths are possible. Cadar and Sen [67] describe symbolic execution in more detail.

SymDroid uses the features of symbolic execution to implement nondeterministic event inputs (such as button clicks or spinner selections), up to a certain bound. Since we have symbolic variables available, we also use them to represent arbitrary secret inputs, as discussed below in Sec. 3.4.2. There are several issues that arise in applying SymDroid to checking our policies, as we discuss next.

3.4.1 Driving App Execution

Android apps use the Android framework's API, which includes classes for responding to events via callbacks. We could try to account for these callbacks by symbolically execution Android framework code directly, but past experience suggests this is intractable: the framework is large, complicated, and includes native code. Instead, we created an *executable model*, written in Java, that mimics key portions of Android needed by our subject apps. Our Android model includes facilities for generating clicks and other GUI events (such as the View, Button, and CheckBox classes, among others). It also includes code for LocationManager, TelephonyManager, and other basic Android classes.

In addition to code modeling Android, the model also includes simplified versions of Java library classes such as StringBuffer and StringBuilder. Our versions of these APIs implement unoptimized versions of methods in Java and escape to internal SymDroid functions to handle operations that would be unduly complex to symbolically execute. For instance, SymDroid represents Java String objects with OCaml strings instead of Java arrays of characters. It thus models methods such as String.concat with internal calls to OCaml string manipulation functions. Likewise, reflective methods such as Class.getName are handled internally.

For each app, we created a driver that uses our Android model to simulate user input to the GUI. The driver is specific to the app since it depends on the app's GUI. The driver begins by calling the app's onCreate method. Next it invokes special methods in the Android model to inject GUI events. There is one such method for each type of GUI element, e.g., buttons, checkboxes, etc. For example, Trace.addClick(id) generates a click event for the given id and then calls the appropriate event handler. The trace entry contains the event name for that kind of element, and a value if necessary. Event handlers are those that the app registered through standard Android framework mechanisms, e.g., in onCreate.

Let m be the number of possible GUI events. To simulate one arbitrary GUI event, the driver uses a block that branches m ways on a fresh symbolic variable, with a different GUI action in each branch. Typical Android apps never exit unless the framework kills them, and thus we explore sequences of events only up to a user-specified *input depth* n. Thus, in total, the driver will execute at least m^n paths.

3.4.2 Symbolic Variables in Traces

In addition to GUI inputs, apps also use secret inputs. We could use SymDroid to generate concrete secret inputs, but instead we opt to use a fresh symbolic variable for each secret input. For example, the call to manager.getDeviceId in Fig. 3.1 returns a symbolic variable, and the same for the call to manager.getPhoneNumber. This choice makes checking policies using symbolic execution a bit more powerful, since, e.g., a symbolic integer variable represents an arbitrary 32-bit integer. Note that whenever ClickRelease generates a symbolic variable for a secret input, it also generates a trace event corresponding to the input.

Recall that secret inputs may appear in traces, and thus traces may now contain symbolic variables. For example, using α_i's as symbolic variables for the secret ID and phone number inputs, the traces (1) and (2) become

$$\mathsf{id!}\alpha_1, \mathsf{ph!}\alpha_2, \mathsf{idBox!true}, \mathsf{sendBtn!unit}, \mathsf{netout!}\alpha_2 \quad (1')$$

$$\mathsf{id!}\alpha_1, \mathsf{ph!}\alpha_2, \mathsf{idBox!true}, \mathsf{sendBtn!unit}, \mathsf{netout!}\alpha_2 \quad (2')$$

We must take care when symbolic variables are in traces. Recall *level* checks $t, i \models \phi$ and then assigns a security level to position i. If ϕ depends on symbolic variables in t, we may not be able to decide this. For example, if the third element in $(1')$ were $\mathsf{idBox!}\alpha_3$, then we would need to reason with conditional security levels such as $\mathit{level}(t, P, 0) = \textbf{if } \alpha_3 \textbf{ then } \mathit{Low} \textbf{ else } \mathit{High}$. We avoid the need for such reasoning by only using symbolic variables for secret inputs, and by ensuring the level assigned by a policy does not depend on the value of a secret input. We leave supporting more complex reasoning to future work.

3.4.3 Checking Policies with Z3

Each path explored by SymDroid yields a pair (t, Φ), where t is the trace and Φ is the path condition. ClickRelease uses Z3 to check whether a given set of such trace–path condition pairs satisfies a policy P. Recall that Definition 1 assumes for each S there is an $=_S$ relation on traces. We use the same relation below, encoding it as an SMT formula. For our example lattice, $=_{High}$ produces `true`, $=_{Low}$ produces a conjunction of equality tests among corresponding trace elements, and $=_{MaskLower8}$ produces the conjunction of equality tests of the bitwise-and of every element with `0xffffff00`.

Given a trace t, let t' be t with its symbolic variables primed, so that the symbolic variables of t and t' are disjoint. Given a path condition Φ, define Φ' similarly. Now we can give the algorithm for checking a security policy.

Algorithm 1 *To check a set $\mathcal{T}$ of trace–path condition pairs, do the following. Let P be the app's security policy. Apply* level *across each trace to obtain the level of each event. For each (t_1, Φ_1) and (t_2, Φ_2) in $\mathcal{T} \times \mathcal{T}$, and for each S, ask Z3 whether the following formula (the negation of Definition 1) is unsatisfiable:*

$$\mathit{level}(t_1, P)^{S,in} =_S \mathit{level}(t_2', P)^{S,in} \wedge \mathit{level}(t_1, P)^S \neq_S \mathit{level}(t_2', P)^S \wedge \Phi_1 \wedge \Phi_2'$$

If no such formula is unsatisfiable, then the program satisfies noninterference.

We include Φ_1 and Φ_2' to constrain the symbolic variables in the trace. More precisely, t_1 represents a *set* of concrete traces in which its symbolic variables are instantiated in all ways that satisfy Φ_1, and analogously for t_2'.

If the above algorithm finds an unsatisfiable formula, then Z3 returns a counterexample, which SymDroid uses in turn to generate a pair of concrete traces as a counterexample. For example, consider traces (1') and (2') above, and prime symbolic variables in (2'). Those traces have the trivial path condition true, since neither branches on a symbolic input. Thus, the formula passed to Z3 will be:

$$\alpha_1 = \alpha_1' \wedge \mathsf{true} = \mathsf{true} \wedge \mathsf{unit} = \mathsf{unit} \wedge \left(\alpha_1 \neq \alpha_1' \vee \mathsf{true} \neq \mathsf{true} \vee \mathsf{unit} \neq \mathsf{unit} \vee \alpha_2 \neq \alpha_2'\right)$$

Thus we can see a satisfying assignment with $\alpha_1 = \alpha_1'$ and $\alpha_2 \neq \alpha_2'$, hence noninterference is violated.

3.4.4 Minimizing Calls to Z3

A naive implementation of the noninterference check generates n^2 equations, where n is the number of traces produced by ClickRelease to be checked by Z3. However, we observed that many of these equations correspond to pairs of traces with different sequences of GUI events. Since GUI events are low inputs in all our policies, these pairs trivially satisfy noninterference (the left-hand side of the implication in Definition 1 is false). Thus, we need not send those equations to Z3 for an (expensive) noninterference check.

We exploit this observation by organizing SymDroid's output traces into a tree, where each node represents an event, with the initial state at the root. Traces with common prefixes share the same ancestor traces in the tree. We systematically traverse this tree using a cursor t_1, starting from the root. When t_1 reaches a new input event, we then traverse the tree using another cursor t_2, also starting from the root. As t_2 visits the tree, we do not invoke Z3 on any traces with fewer input events than t_1 (since they are not low-equivalent to t_1). We also skip any subtrees where input events differ.

3.5 Experiments

To evaluate ClickRelease, we ran it on four apps, including the two described in Section 5.2. We also ran ClickRelease on several insecure variants of each app, to ensure it can detect the policy violations. The apps and their variants are:

- *Bump.* The bump app and its policy appear in Fig. 3.1. The first insecure variant counts clicks to the send button sends the value of the ID after three clicks, regardless of the state of the ID checkbox. The second (indicated in the comments in the program text) swaps the released information—if the ID box is checked, it releases the phone number, and vice-versa.

- *Location toggle.* The location toggle app and its policy appear in Fig. 3.2. The first insecure variant always shares fine-grained location information, regardless of the radio button setting. The second checks if coarse-grain information is selected. If so, it stores the fine-grained location (but does not send it yet). If later the fine-grained radio button is selected, it sends the stored location.

Recall this is forbidden by the app's security policy, which allows the release only of locations received while the fine-grained option is set.

- *Contact picker.* We developed a contact picker app that asks the user to select a contact from a spinner and then click a send button to release the selected contact information over the network. The security policy for this app requires that no contact information leaks unless it is the last contact selected before the button click. (For example, if the user selects contact 1, selects contact 2, and then clicks the button, only contact 2 may be released.) Note that since an arbitrarily sized list of contacts would be difficult for symbolic execution (since then there would be an unbounded number of ways to select a contact), we limit the app to a fixed set of three contacts. The first insecure variant of this app scans the set of contacts for a specific one. If found, it sends a message revealing that contact exists before sending the actual selected contact. The second insecure variant sends a different contact than was selected.

- *WhereRU.* Lastly, we developed an app that takes push requests for the user's location and shares it depending on user-controlled settings. The app contains a radio group with three buttons, "Share Always," "Share Never," and "Share On Click." There is also a "Share Now" button that is enabled when the "Share On Click" radio button is selected. When a push request arrives, the security policy allows sharing if (1) the "Always" button is selected, or (2) the "On Click" button is selected and the user presses "Share Now." Note that, in the second case, the location may change between the time the request arrives and the time the user authorizes sharing; the location to be shared is the one in effect when the user authorized sharing, i.e., the one from the most recent location update before the button click. Also, rather than include the full Android push request API in our model, we simulated it using a basic callback. This app has two insecure variants. In the first one, when the user presses the "Share Now" button, the app begins continuously sharing (instead of simply sharing the single location captured on the button press). In the second, the app shares the location immediately in response to all requests.

Scalability. We ran our experiments on a 4-core i7 CPU @3.5GHz with 16GB RAM running Ubuntu 14. For each experiment we report the median of 10 runs.

In our first set of experiments, we measured how ClickRelease's performance varies with input depth. Figure 3.4 shows running time (log scale) versus input depth for all programs and variants. For each app, we ran to the highest input depth that completed in one hour.

For each app, we see that running time grows exponentially, as expected. The maximum input depth before timeout (i.e., where each curve ends) ranges from five to nine. The differences have to do with the number of possible events at each input point. For example, WhereRU has seven possible input events, so it has the largest possible "fan out" and times out with an input depth of five. In contrast, Bump and Location Toggle have just three input events and time out with an input depth of nine. Notice also the first insecure variant of Contact Picker times out after fewer events than the other variants. Investigating further, this occurs due to that app's

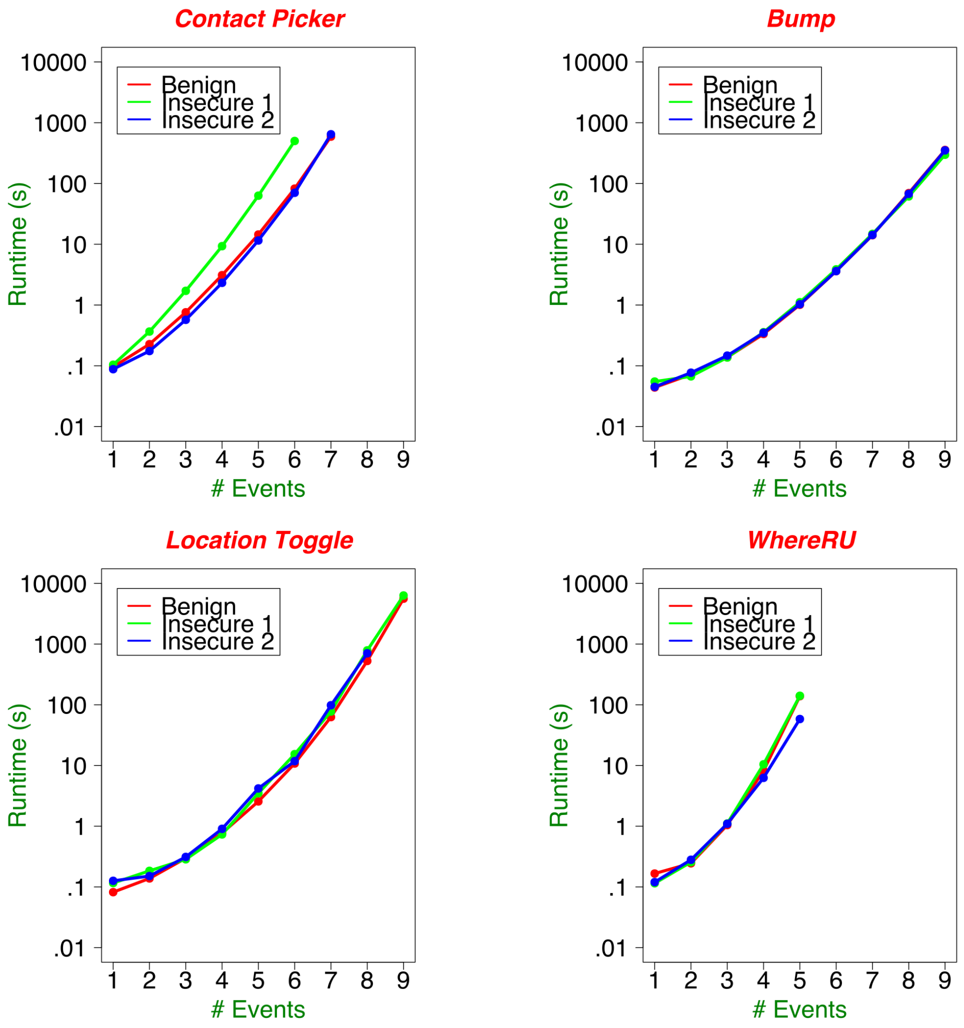


Figure 3.4: Runtime vs. number of events.

implicit flow (recall the app branches on the value of a secret input). Implicit flows cause symbolic execution to take additional branches depending on the (symbolic) secret value.

Minimum Input Depth. Next, for each variant, we manually calculated a *minimum* input depth guaranteed to find a policy violation. To do so, first we determined possible app GUI states. For example, in Bump (Fig. 3.1), there is a state with idBox and phBox both checked, a state with just idBox checked, etc. Then we

App	Input Depth	Time (ms)		
		Exploration	Analysis	Total
Bump	3	114	15	142
Bump (insecure 1)	5	2,100	1,577	3,690
Bump (insecure 2)	4	266	70	344
Location toggle	2	113	12	128
Location toggle (insecure 1)	2	143	12	163
Location toggle (insecure 2)	3	117	12	143
Contact picker	2	79	2	94
Contact picker (insecure 1)	2	325	27	361
Contact picker (insecure 2)	2	149	9	170
WhereRU	3	849	183	1,045
WhereRU (insecure 1)	3	860	234	1,108
WhereRU (insecure 2)	2	257	10	280

Figure 3.5: Results at minimum input depth.

examined the policy and recognized that certain input sequences lead to equivalent states modulo the policy. For example, input sequences that click idBox an even number of times and then click send are all equivalent. Full analysis reveals that an input depth of three (which allows the checkboxes to be set any possible way followed by a button click) is sufficient to reach all possible states for this policy. We performed similar analysis on other apps and variants.

Fig. 3.5 summarizes the results of running with the minimum input depth for each variant, with the depths listed in the second column. We confirmed that, when run with this input depth, ClickRelease correctly reports the benign app variants as secure and the other app variants as insecure. The remaining columns of Fig. 3.5 report ClickRelease's running time (in milliseconds), broken down by the exploration phase (where SymDroid generates the set of symbolic traces) and the analysis phase (where SymDroid forms equations about this set and checks them using Z3). Looking at the breakdown between exploration and analysis, we see that the former dominates the running time, i.e., most of the time is spent simply exploring program executions. We see the total running time is typically around a second or less, while for the first insecure variant of Bump it is closer to 4 seconds, since it uses the highest input depth.

Our results show that while ClickRelease indeed scales exponentially, to actually find security policy violations we need only run it with a low input depth, which takes only a small amount of time.

3.6 Limitations and Future Work

There are several limitations of ClickRelease we plan to address in future work.

Thus far we have applied ClickRelease to a set of small apps that we developed. There are two main engineering challenges in applying ClickRelease to other apps. First, our model of Android (Section 3.4.1) only includes part of the framework. To run on other apps, it will need to be expanded with more Android APIs. Second, we speculate that larger apps may require longer input depths to go from app launch to interfering outputs. In these cases, we may be able to start symbolic execution "in the middle" of an app (e.g., as in the work of Ma et al. [68]) to skip uninteresting prefixes of input events.

ClickRelease also has several limitations related to its policy language. First, ClickRelease policies are fairly low level. Complex policies—e.g., in which clicking a certain button releases multiple pieces of information—can be expressed, but are not very concise. We expect as we gain more experience writing ClickRelease policies, we will discover useful idioms that should be incorporated into the policy language. Similarly, situations where several methods in sequence operate on and send information should be supported. Second, currently ClickRelease assumes there is a single adversary who watches netout. It should be straightforward to generalize to multiple output channels and multiple observers, e.g., to model inter-app communication. Third, we do not consider deception by apps, e.g., we assume the policy writer knows whether the sendBtn is labeled appropriately as “send” rather than as “exit.” We leave looking for such deceptive practices to future work.

Finally, since ClickRelease explores a limited number of program paths it is not sound, i.e., it cannot guarantee the absence of policy violations in general. However, in our experiments we were able to manually analyze apps to show that exploration up to a certain input depth was sufficient for particular apps, and we plan to investigate generalizing this technique in future work.

3.7 Related Work

ClickRelease is the first system to enforce extensional declassification policies in Android apps. It builds on a rich history of research in usable security, information flow, and declassification.

One of the key ideas in ClickRelease is that GUI interactions indicate the security desires of users. Roesner et al. [69] similarly propose *access control gadgets* (ACGs), which are GUI elements that, when users interact with them, grant permissions. Thus, ACGs and ClickRelease both aim to better align security with usability [70]. ClickRelease addresses secure information flow, especially propagation of information after its release, whereas ACGs address only access control.

Android-based systems. TaintDroid [71] is a run-time information-flow tracking system for Android. It monitors the usage of sensitive information and detects when that information is sent over insecure channels. Unlike ClickRelease, TaintDroid does not detect implicit flows.

AppIntent [72] uses symbolic execution to derive the *context*, meaning inputs and GUI interactions, that causes sensitive information to be released in an Android app. A human analyst examines that context and makes an expert judgment as to whether the release is a security violation. ClickRelease instead uses human-written LTL formulae to specify whether declassifications are permitted. It is unclear from [72] whether AppIntent detects implicit flows.

Pegasus [73] combines static analysis, model checking, and run-time monitoring to check whether an app uses API calls and privileges consistently with users' expectations. Those expectations are expressed using LTL formulae, similarly to ClickRelease. Pegasus synthesizes a kind of automaton called a *permission event graph* from the app's bytecode then checks whether that automaton is a model for the formulae. Unlike ClickRelease, Pegasus does not address information flow.

Jia et al. [74] present a system, inspired by Flume [75], for run-time enforcement of information flow policies at the granularity of Android components and apps. Their system allows components and apps to perform trust declassification according to capabilities granted to them in security labels. In contrast, ClickRelease reasons about declassification in terms of user interactions.

Security type systems   Security type systems [76] statically disallow programs that would leak information. O'Neill et al. [18] and Clark and Hunt [77] define interactive variants of noninterference and present security type systems that are sound with respect to these definitions.

Integrating declassification with security type systems has been the focus of much research. Chong and Myers [14] introduce *declassification policies* that conditionally downgrade security labels. Their policies use classical propositional logic for the conditions. ClickRelease can be seen as providing a more expressive language for conditions by using LTL to express formulae over events. SIF (Servlet Information Flow) [78] is a framework for building Java servlets with information-flow control. Information managed by the servlet is annotated in the source code with security labels, and the compiler ensures that information propagates in ways that are consistent with those labels. The SIF compiler is based on Jif [79], an information-flow variant of Java.

All of these systems require adding type annotations to terms in the program code, e.g., method parameters, etc. In contrast, ClickRelease policies are described in terms of app inputs and outputs.

Event Based Models and Declassification   Vaughan and Chong [80] define expressive declassification policies that allow functions of secret information to be released after events occur, and extend the Jif compiler to infer events. ClickRelease instead ties events to user interactions.

Rafnsson et al. [81] investigate models, definitions, and enforcement techniques for secure information flow in interactive programs in a purely theoretical setting. Sabelfeld and Sands [12] survey approaches to secure declassification in a language-based setting. ClickRelease can be seen as addressing their “what” and “when” axes of declassification goals: users of Android apps interact with the GUI to control when information may be released, and the GUI is responsible for conveying to the user what information will be released.

3.8 Conclusion

We introduced interaction-based declassification policies, which describe *what* and *when* information can flow. Policies are defined using LTL formulae describing event traces, where events include GUI actions, secret inputs, and network sends. We formalized our policies using a trace-based model of apps based on security relevant events. Finally, we described ClickRelease, which uses symbolic execution to check interaction-based declassification policies on Android, and showed that ClickRelease correctly enforces policies on four apps, with one secure and two insecure variants each.

Chapter 4: User Interactions and Permission Use on Android

Our last chapter introduced interaction-based declassification policies. However, it is unclear whether this matches user intuition about how apps behave. In this chapter, I introduce a system for studying how permission uses relate to user interaction within apps. This system, named AppTracer, builds on the logging facilities present in Redexer. It leverages logging and visualization to help perform an app study over 150 top apps from the Google Play store. I then follow this up with a 961-participant user study to investigate how users understand permissions as they relate to the UI. I found that the app's UI does deeply inform a user's expectation that a permission will be used, but that the current implementation of the Android permission system may mislead users under certain circumstances.

4.1 Introduction

Android has a *permission system* that asks users for authorization before an app uses sensitive resources such as contacts or GPS location. A key challenge in such authorization systems is balancing user interruptions with making sensitive resource use transparent. We hypothesize that Android's existing authorization systems (install-time permission lists or run-time dialog boxes, depending on the version) could achieve a better balance by integrating with the app's user interface (UI), because the UI deeply informs the user's mental model of the app's behavior, including security-relevant behavior.

In particular, in this chapter we ask whether *user interactions*—button clicks, page changes, dialog boxes, etc.—can be taken as evidence of authorization to use certain sensitive resources. If so, this could reduce the need for separate authorization requests. Conversely, I ask whether sensitive resource use without an associated interaction suggests a need for additional authorization requests. Note that while our studies are heavily influenced by Android, I believe the results will generalize to related mobile OS's and similar settings like web apps.

To answer these questions, I conducted two related studies. First, I (and another grad studnet) reviewed 150 popular Android apps to determine whether sensitive resource uses are related to user interactions in existing apps. If so, an authorization mechanism integrated with the UI could work well with existing app designs. To carry out this study, we developed AppTracer, a dynamic analysis tool that instruments Android apps to log UI actions and resource uses, and then visualizes the logs as graphs that show temporal ordering of logged events. We used AppTracer to determine whether each observed resource use in each app was *interactive*, meaning either it was immediately preceded by a related UI event (e.g., accessing contacts after clicking a button marked "contacts"), or it was the main focus of the current screen (e.g., using location on a map screen).

We found that, across our subject apps, several resources (microphone, camera, external storage, and calendar) are used almost exclusively interactively; several others (including bluetooth and phone state) are used mostly non-interactively (which we call in the *background* even if the app itself is on screen); and several resources (most notably contacts and location) exhibit a mix of interactive and background uses. These results suggest interactive and background uses may call for different authorization mechanisms, and that these mechanisms cannot necessarily be divided strictly by resource.

These results informed the design of our second study, a 961-participant online survey investigating participants' expectations about interactive and background permission uses. Each participant viewed a slideshow of two usage scenarios for a mock mobile app, where each scenario shows a short interaction (e.g., launching the app, clicking a button, etc.) and then asks if the participant expects microphone, location, and/or contacts to be used after the interaction. We chose these resources to reflect a range of interactivity as measured in our app study. We aimed to gain insight into how different factors affect user expectations, and therefore which authorization mechanisms might be appropriate for different usage patterns.

As we anticipated, we found that users are much more likely to expect resources to be accessed after a related interaction than in the background. However, we also found that seeing one interactive use does not prime the user to expect a future background use, indicating a potential weakness in the Android M request-on-first-use authorization model. In contrast, our findings show that an authorization request at launch does increase expectations for both interactive and background accesses, perhaps because it better conveys the idea that the resource could be accessed at any time.

Drawing on the results of our studies, we make three design recommendations. First, resource uses should be made after associated interactions as much as possible. Given the current makeup of apps, this should be achievable for many commonly used resources without extensive effort. Second, separate authorization dialogs might be unnecessary for resources that are accessed mostly interactively (see, e.g., [82]). Finally, authorization for background resource uses should be distinct from authorization for interactive uses, and these background authorizations may be most effective when the app is launched.

4.2 Background and Related Work

In earlier versions of Android, users were presented with a list of permissions requested by an app at install time. The user could then either grant the app full use of those permissions or not install the app at all. This model had a number of problems: few users comprehended or even read the list of permissions [83], and many apps requested more permissions than they used [51]. Because of these issues, Android M [84] switched to a model where apps ask for a permission the first time it is needed; the permission is then granted indefinitely.

In our work, we ask whether authorization systems similar to Android's can be improved by taking the user interface into account. Note that our work is orthogonal to the question of whether permissions are at the right level of granularity [9,85] or protect the right resources [86].

Contextual Security on Mobile Devices The motivation for our work, that authorization can be better integrated with the UI, exemplifies *contextual security* [4], which suggests security decisions should take the context into account. Several researchers have studied contextual security on mobile devices. Almuhimedi et. al [87] showed users historical data about how apps accessed their locations. They found 95% of users reassessed the apps' need for location, with 58% of those users further restricting location access. King [5] found users are more likely to expect sensitive resource accesses when suggested by the context. Felt et. al [88] proposed a process for deciding the appropriate authorization mechanism for a permission based on the a permissions' use in context. Several researchers [6–8] found users are surprised by some sensitive resource accesses that occur when apps are in the background. Most closely related to this paper, in a field study Wijesekera et al. [8] found that context is an important factor in determining expectation of resource use. Our work builds on this finding by using a controlled experiment to distinguish how different contextual factors, including consecutive interactions, contribute to user expectations.

The works just mentioned mainly define context as whether the app is on or off the screen. In contrast, we use a much richer notion of context based on sequences of UI actions.

Enforcing Contextual Security Many systems have been proposed to enforce contextual security in apps. Chen et. al [73] present Pegasus, a static analysis system for analyzing apps and enforcing policies based on *permission event graphs* (PEGs). For example, Pegasus can check that contacts are only accessed after clicking a certain button. PEGs inspired the design of AppTracer. However, AppTracer uses dynamic (rather than static) analysis to reduce issues of false positives—every behavior AppTracer logs occurred in an actual run, whereas static analysis may report sensitive resource accesses that can never actually occur.

Yang et al. [72] presented AppIntent, which uses symbolic execution to determine sequences of UI events that lead to information leakage. Micinski et. al [63] use symbolic execution to enforce secure information-flow properties based on UI events. While both systems are promising, in practice symbolic execution is difficult to run at scale on Android apps due to the complexity of modeling the Android framework.

Stiegler et al. developed CapDesk [89] and later Polaris [90], two capability-based desktop system that utilize user interaction to drive access control. However, CapDesk and Polaris's focus is limited to file access. Roesner et. al [82] expand user-driven access control with *Access Control Gadgets* (ACGs), which tie resource accesses to certain UI elements, e.g., an ACG might allow location to be used only after a specific button is clicked. ACGs were later expanded to work on Android, with and later without modifying the operating system [91, 92]. The original ACG paper includes a user study measuring expectations related to interactive permission uses; our work expands on this idea to study a broader variety of factors and use cases. While the current paper does not directly implement or measure ACGs, our findings do support the use of ACGs.

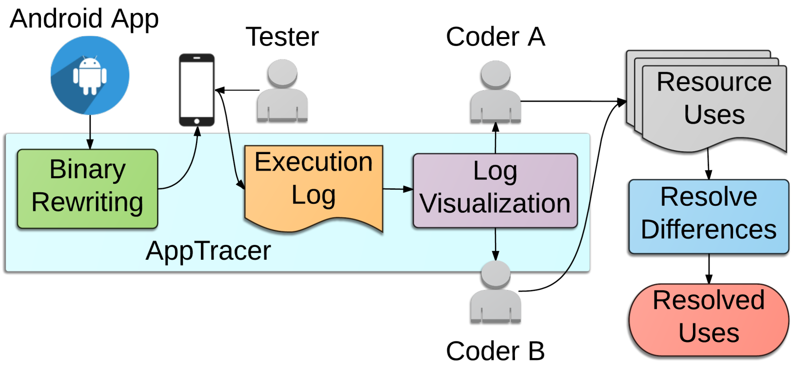


Figure 4.1: App Measurement Survey Procedure.

4.3 App Measurement Survey Methodology

In our first study, we reviewed a set of popular Android apps to determine how UI actions and resource uses are related. Figure 4.1 gives a high-level overview of our review process, which ultimately produces a set of *resource use codes* that indicate, for each resource use, what event (if any) caused the use. For example, we might determine that in some app, contacts were accessed just after a button click, or location was used immediately when the app launched.

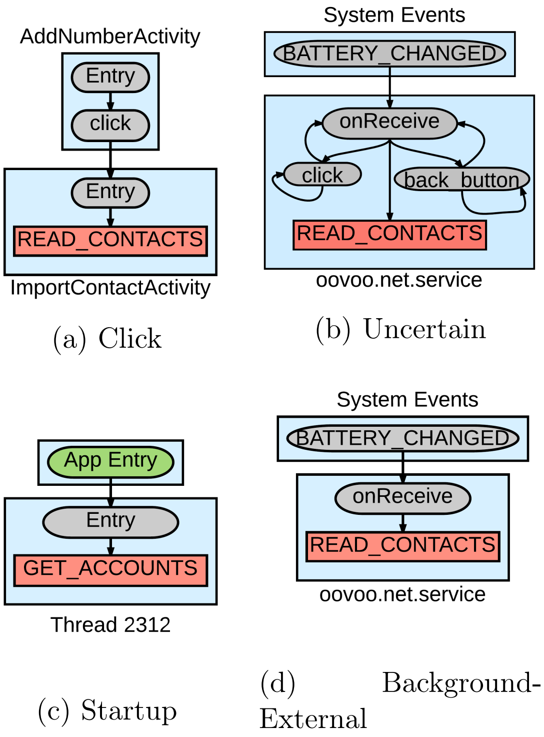
The next subsections walk through the review process in detail.

4.3.1 Binary Rewriting and Execution Logging

The first step of our process uses AppTracer, a dynamic analysis tool we developed. AppTracer instruments the subject app so that, when run, it writes a log of UI events and permission uses. AppTracer adds instrumentation using Redexer [9], a rewriting tool for Dalvik bytecode (the language to which Android apps are compiled). Generally speaking, AppTracer instruments code by appending a log method to the bytecode and inserting calls to log at the beginning of UI-related callbacks (e.g., onClick handlers for button events) and just before calls to permission-protected methods (e.g., getLastLocation).

We identified methods associated with the UI using the Android documentation. We identified permission-protected methods using the PScout dataset [93], which attempts to list every security-sensitive method and its associated permission. PScout also includes information about sensitive content providers, intents, etc. Note that while PScout is reasonably thorough, we found several cases it missed. For example, we observed several visual indicators of SD card use that had no corresponding log entry AppTracer. After investigating the apps' decompiled code, we found the SD card was used via Java IO methods omitted by PSCout. Whenever we found such cases, we added the missing methods to our copy of the PScout database and reran our evaluated apps.

Once an app has been instrumented, a tester runs it to generate a log. (Note that one log is usually sufficient because in most apps, any app state is reachable from any other app state.) We elide the details of the log, but at a high level, for each UI event, AppTracer records the type of event and the corresponding parameters (e.g., what button was clicked, which menu option was selected, etc.). For each permission use, AppTracer logs the name of the called method and its arguments.



(a) Click

(b) Uncertain

(c) Startup

(d) Background-External

Figure 4.2: AppTracer graphs and corresponding resource access patterns in our codebook.

4.3.2 Log Visualization

After the tester produces a log, the next step is log visualization. Figures 4.2a–4.2b show example portions of AppTracer's graph-based visualization, redrawn for compactness and to omit most package names. Here, light blue boxes represent *activities*, which are the "screens" of the app. Within each activity, gray ovals represent the beginning of that activity ("entry," which usually corresponds to the onStart handler), UI events (e.g., "click"), or system events (e.g., BATTERY_CHANGED). Red rectangles indicate calls to methods protected by the named permission. There is an edge from node A to node B if A occurs immediately before B in the log. For example, in Figure 4.2a, contacts were read immediately after the ImportContactActivity

activity was started.

Since logs can be quite lengthy, AppTracer heuristically merges nodes that arise from the same position in the bytecode. This sometimes results in ambiguity. For example, in Figure 4.2b, the single onReceive node actually represents calls to an Android broadcast receiver that AppTracer coalesced. We discuss this more below.

Finally, AppTracer also allows the user to directly view the log file entries corresponding to a node in the graph. This is useful to retrieve more detailed information about the node. For example, when reviewing Figure 4.2a, we looked in the log to determine that the clicked button had text associated with contacts. As another example, we used the logs to distinguish SD card accesses to user files from accesses to the app's own storage. We do not count the latter as a sensitive resource access because it accesses data owned by the app.

4.3.3 Resource Uses

The next step is to examine the AppTracer graph and record a set of codes that accurately categorize various resource accesses. More precisely, for each red node in the graph, the coder assigns a pair of the form $(resource, pattern)$, where *resource* indicates what is protected by the permission and *pattern* is one of six different UI patterns, discussed next.

To keep our results understandable, we grouped together resources according to Google's permission groups [94]. For example, the single SMS resource includes more fine-grained permissions such as READ_SMS and SEND_SMS.

We developed an initial codebook for UI patterns based on our knowledge of Android app development. We then iteratively applied our codebook to sets of five apps (not in our evaluation set) at a time and adjusted the codes as necessary. After evaluating a total of twenty apps, we felt we had reached a codebook with a minimal set of orthogonal patterns.

The six access patterns in our codebook are grouped into three categories. First are the *interactive* patterns *Click* and *Page*. The code *Click* indicates resource use preceded by a UI event (including non-clicks such as swipes). For example, in Figure 4.2a, we coded READ_CONTACTS as *Click* because the contacts were accessed on a straight-line path from the click of a button labeled as importing contacts. *Page* codes uses that are clearly associated with an activity but are not associated with exactly one click. For example, *Page* would code the use of location during an activity that shows a list of nearby stores.

Second, in the *background* patterns *Startup*, *Bg-App*, and *Bg-Ext*, the resource use has no obviously connected user action. *Startup* codes a resource accessed right after app launch but before any screen appears (and is thus disjoint from *Page*), e.g., the use of accounts in Figure 4.2c, which occurs in a thread created at app launch. *Bg-App* codes a resource used while the app is in the foreground, but with no clearly related user action. For example, a graph similar to Figure 4.2a would be coded as *Bg-App* if a button labeled for importing contacts was followed by the GET_ACCOUNTS permission rather than READ_CONTACTS. *Bg-Ext* denotes a resource used due to a system event, meaning it would be used whether or not the app is on screen. For example, in Figure 4.2d, accounts are read after the

BATTERY_CHANGED system event.

Finally, *Uncertain* denotes a resource access that did not fit any of the previous patterns. In Figure 4.2b, we see a path from a system event to onReceive to READ_CONTACTS, which should be coded as *Bg-Ext*. But, the onReceive node also has an incoming edge from the click handler and the back button. These uses would be coded as *Bg-App*. (The *Click* and *Page* codes do not apply because the associated buttons do not indicate contacts will be used.) Because there are multiple possibilities, we code this case as *Uncertain*.

If desired, an AppTracer user can subsequently review the logs to try to resolve the uncertainty. For example, here the log entry associated with BATTERY_CHANGED is followed by a call to onReceive and then READ_CONTACTS. Thus, the permission use is coded as *Bg-Ext*. We then separately looked at the log entries for click and back_button, and found they were followed by an onReceive that was *not* followed by READ_CONTACTS. Thus, examining the logs allowed us to distinguish paths that we merged in AppTracer.

Note that for each app we only count each $(\mathit{resource}, \mathit{pattern})$ pair once, no matter how often it occurs in the log. A stronger notion of frequency would be hard to interpret, since it would depend on how AppTracer heuristically coalesces graph nodes as well as to how the tester explored the app.

4.3.4 Coding Apps and Resolving Differences

Coding the resource uses of an app is inherently complex. Thus, two coders reviewed apps independently in sets of 15 and met after each set to resolve differences. Coders took approximately 10–20 minutes to code each app, and resolving differences for a set of 15 apps took approximately 30 minutes.

Most differences between coders were due to one coder overlooking a path or a resource in the AppTracer graph. In almost all such cases, when the other coder pointed out the omission, the first coder would quickly reach the same conclusion as the other coder. The remaining differences were caused by disagreements about whether the resource use was interactive. For example, one app read a user's accounts within an activity for filling a form. One coder recorded this as *Click*, since the accounts were read after a click. The other coder recorded this as *Bg-App*, because there was no observable use of the account data, such as pre-filling the form with data from the user's existing accounts. After encountering several such cases, the coders decided on a general principle of coding uses as *Bg-App* unless the UI explicitly mentioned that the resource could be used—hence, this example was resolved as *Bg-App*.

Inter-rater reliability between our coders for the non-visualization-error disagreements was Krippendorff's $\alpha = 0.897$, indicating close agreement [95].

4.3.5 App Selection

We drew our subject apps from a larger set comprising the 20 top downloaded free apps[1] from the 27 non-gaming categories on Google Play. We excluded gaming apps because they typically use native code that AppTracer cannot analyze. This yielded 503 apps (note there is overlap between categories).

We then randomly selected 150 apps to evaluate, subject to the constraint that no more than two apps were from the same developer. (We wanted to avoid bias due to overrepresentation of apps coded in the same style.). We excluded any apps that could not be run with AppTracer, replacing them with new randomly drawn apps to maintain an evaluation set of 150.

We excluded 48 apps because Redexer fails when rewriting them and 23 apps because either they refuse to run if modified (due to internal or system signature checks) or they are primarily implemented in native code. In most cases, we created accounts when signup was required to fully exercise an app, but we excluded 16 apps because they require accounts that are hard to set up online or require a fee (e.g., bank accounts).

4.3.6 Limitations

There are several potential limitations to this study. First, the tester may miss some app behavior, leading to a log that omits some possible resource uses and contexts. We tried to alleviate this concern by exercising as much of the app as

[1] as of June 11, 2016

possible. However, we did not use app features that required payment.

Second, AppTracer has limitations mentioned above: it may miss UI events or permission-protected methods, and it merges nodes that correspond to the same position in the bytecode. We tried to address the former issue by looking for cases where we expected an event to be in a log but it was not, and then adding the missing events or methods. We addressed the latter issue by manually disambiguating some of the uncertain cases.

Third, our study reviewed only popular Android apps. Observing that a resource is already accessed interactively in popular apps would indicate that changing the Android framework or system to implement interactivity-related protections for those accesses may be reasonable. These apps also represent the common case and likely help to set user expectations about apps and permissions. However, we note that the apps we examine likely differ from the long tail of other apps in important ways; popular apps are likely implemented to high standards and unlikely to be malicious. We leave similar measurements on a broader set of apps as future work.

4.4 App Measurement Survey Results

Figure 4.3 summarizes the resource use patterns we found. For example, across all apps, the camera was used in the *Click* pattern 57 times. Orange-shaded bars indicate interactive use patterns, and blue shades represent background uses. Resources are sorted by percent of interactive patterns.

We see that, to a rough approximation, the more sensitive the resource the

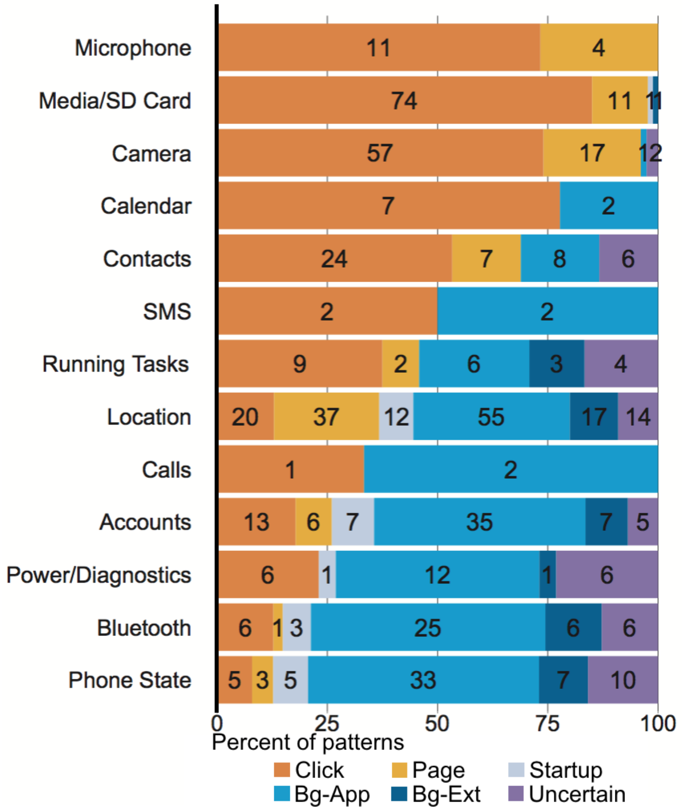


Figure 4.3: Percent of observed patterns of each type, per resource. Bar labels indicate how many apps we observed using each pattern, non-normalized.

more likely it is used interactively. Indeed, microphone, media, camera, and calendar were accessed almost exclusively interactively. We investigated the background and *Uncertain* uses for these resources. One calendar use and the two SD card uses were due to periodic background tasks (calendar syncing or scanning for files). One camera use took a picture after the passcode was entered incorrectly three times. One camera use and one calendar use did not seem to access sensitive resources—the camera use accessed configuration information, and the calendar use got the current date (which could be done without accessing the calendar). Finally, after disassembling and reading the bytecode, we found one *Uncertain* camera use could actually only happen interactively.

We next see that contacts, SMS, tasks, location, and calls are used in a mix of interactive and non-interactive ways. We investigated the background and *Uncertain* uses of these resources as well. Contacts were used in the background mainly to prefetch or sync the contact list. Two apps used SMS in the background to listen for a registration code after the user signed up for an account. Currently running tasks were polled in the background to monitor battery usage, scan for malicious apps, look for apps by the same developer (to communicate with them), or for analytics and tracking purposes. Call-related permissions were used in the background to block calls from a user-supplied blacklist. While there were too many background location uses to examine them all, those we did examine frequently had no obvious reason (as expected [7, 87]), even in apps that elsewhere used location for a clear purpose.

Finally, four resources—accounts, power, bluetooth, and phone state information—were mostly accessed in the background. We believe this is either because developers believe users care less about these accesses [86], the uses are hard to explain clearly to non-experts, or the uses are not naturally associated with an immediately preceding interaction.

Looking in more detail at the breakdown between *Click* and *Page*, we see that for most resources, *Click* is a clear majority of the interactive uses. The exception is location, which has more *Page* uses. This was mostly due to location use for map screens or lists of nearby places. Breaking down the background uses, we see the use of resources at *Startup* and in *Bg-Ext* becomes much more common lower in the chart.

Resource	**# Apps**	**Resource**	**# Apps**
Location	75	Microphone	14
Media/SD Card	69	Tasks	13
Camera	69	Power/Diag.	12
Phone State	43	Calendar	5
Accounts	39	SMS	4
Bluetooth	31	Calls	2
Contacts	30		

Table 4.1: Number of apps that used each resource.

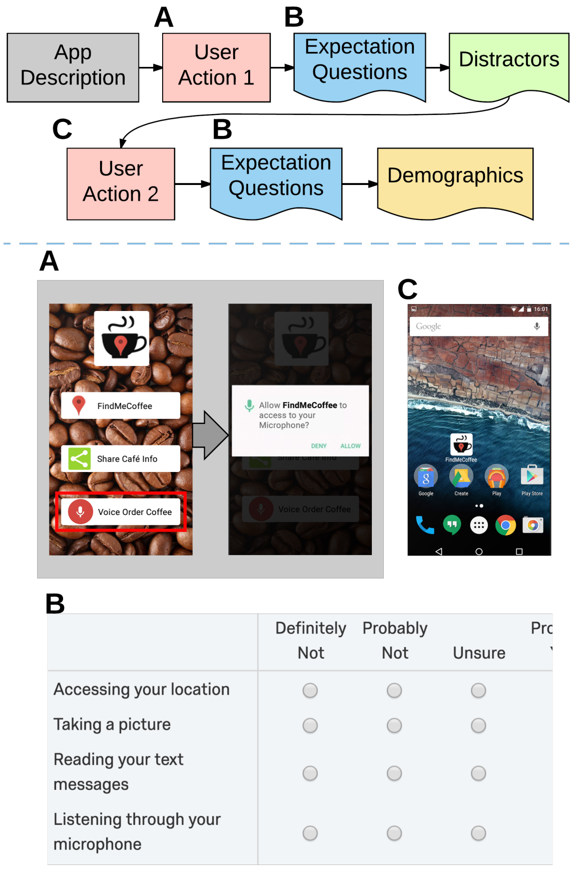
4.4.1 Resource Usage Across Apps

We also measured the number of apps that used each resource at least once, as a rough approximation for how familiar each resource is to users. Table 4.1 shows the results. We found a wide range across resources, with little correlation between frequency of appearance in apps and usage patterns. For example, location was used frequently, and Figure 4.3 shows that it was often used in the background, whereas media/SD card was also used frequently, but was rarely seen in the background.

4.4.2 Discussion

The results of our app survey suggest several possibilities. Given the large amount of interactive resource use overall, there seems to be a clear opportunity for better integrating authorization into the UI. However, the question remains to what extent interactions make resource use apparent to users. We try to answer this question in the next two sections.

Our app survey also shows that many resources are used in the background, and many of these uses seem reasonable, at least to the authors. Moreover, apps



	Definitely Not	Probably Not	Unsure	Pro
Accessing your location	○	○	○	
Taking a picture	○	○	○	
Reading your text messages	○	○	○	
Listening through your microphone	○	○	○	

Figure 4.4: User expectations study procedure. Lower portion shows partial examples of user actions and survey questions.

sometimes use the same resource both in the foreground and in the background, to different purposes. This suggests interactive and background uses should be authorized separately. Thus, in the next two sections, we also try to answer questions about how background uses should be authorized, and about whether users' expectations of interactive and non-interactive uses are related.

4.5 User Expectations Study Methodology

After our app survey, we conducted an online study to elicit users' expectations about resource use as different user actions are performed on a smartphone. Our goal was to understand to what extent user expectations align with the interactivity patterns we observed in our app study. Concretely, our user study examined the following three hypotheses:

H1. Users are more likely to expect resource accesses with an interactive use pattern than without.

H2. The more apps that use a resource (as measured in our app survey), the more likely users are to expect less-interactive uses of that resource.

H3. Users are more likely to expect resource accesses they have seen before.

Note that *H1* and *H3* have implications for the Android M permission system. Specifically, if *H1* is true, users already expect a resource to be accessed due to user action, so an explicit authorization request may be unnecessary. Additionally, if *H3* is false, then users granting authorization for a resource in one access pattern does not cause them to expect that resource to be used later in a different pattern. Thus, requesting authorization only on first use may be insufficient.

H2 is a proxy for a more general hypothesis: that users are more likely to expect accesses to resources with which they are familiar. Familiarity is likely affected by many factors including how many apps use the resource and how often they do so, whether passive notifications are present (as for location), coverage in the news media, etc. For simplicity, we use the frequencies in Table 4.1 as a metric of familiarity.

4.5.1 Study Overview

We recruited participants through Amazon's Mechanical Turk crowdsourcing service. All participants were at least 18 years old and located in the United States. Participants were paid $1.00 for completing the survey. The survey was approved by the University of Maryland IRB. Participants were instructed that we wanted their opinions about an app; privacy and sensitive resources were not explicitly mentioned.

Figure 4.4 gives a flowchart of the procedure followed by each study participant. First, the participant reads a short description of an app. We used two mock apps in our study: FindMeCoffee (*Coffee*) and HealthyFit Tracker (*Fitness*). FindMeCoffee locates nearby coffee shops, allows users to share the location of favorite coffee shops with friends, and supports ordering coffee via voice command. HealthyFit Tracker tracks workouts, allows sharing workouts with friends, and allows posting audio "smack talk" on the user's profile. Note while these apps demonstrate different categories, we do not attempt to fully study the effect of app type on user perceptions.

Next the participant views one sequence of user interactions with the app, shown as a slideshow of app screenshots. To avoid confusion with terminology in the next section, we refer to such a sequence as a *user action.* For example, in the user action in Figure 4.4A, the user clicks the bottom-most button (outlined in red), and the app then displays an authorization request dialog. In Figure 4.4C, which is only shown partially, the user exits the app and returns to the device home screen.

After viewing a user action, the participant answers five-point Likert questions such as those in Figure 4.4B, which ask whether a resource is “Definitely not” to “Definitely yes” accessed immediately after the user action. To avoid priming the user, the survey asks about camera, SMS, flashlight, “accessing credit card information,” and “looking up coffee shop reviews” (*Coffee*) or “reading your heart rate” (*Fitness*) as well as the three resources we study.

Next, the participant answers five distractor questions, views another user action, and answers the same Likert questions about the second user action. The distractors are designed to induce a cognitive break and ensure the two access patterns are treated as separate events. For example, one distractor asks users “Which button would you press if you wanted to find a new coffee shop?” We did not measure the effectiveness of distractors. The participant concludes the survey by answering demographic questions. Participants took 4 minutes and 45 seconds on average to complete the survey.

To understand the effect of resource access patterns on user expectation, we analyze responses about the first user action each participant viewed. We use responses about the second user action to examine how prior exposure affects expectation.

4.5.2 Conditions

Participants were assigned round-robin to one of 42 conditions, which varied across four variables: the *app*, the *resource* being accessed, the *authorization pattern*, and the pair of *interaction (int) patterns*.[2] Table 4.2 lists possible values for each variable, and conditions comprise one value from each column.

As mentioned earlier, our study used two mock apps, FindMeCoffee and HealthyFitTracker, and three resources, chosen to cover a range of int. patterns observed in our app survey: Microphone (*Mic*, only interactive accesses observed), Contacts (*Con*, mixed interactive and background uses observed), and Location (*Loc*, also mixed).

Our study considered three different authorization patterns. First use (*Fst*) mimics Android M, presenting an authorization dialog during the first user action but not the second. Launch (*Lch*) presents an authorization dialog immediately after the app's home page is shown, but before any further screenshots. We noticed anecdotally that a few apps in our study used this strategy. Never (*Nvr*) does not show any authorization dialog at run time. This mimics older versions of Android.

We examined three different int. patterns, also drawn from the app survey:

[2] Through this section we will use the abbreviation int. pattern to avoid confusion with the interactions in our regression analysis.

App	**Resource**[1]	**Authorization**[2]	**Int. Patterns**[3]
Coffee	*Mic*	*Fst*	*Clk-Clk* [4]
Fitness	*Con* [4]	*Lch*	*Clk-Bg*
	Loc	*Nvr*	*Bn-Bg*
			Bg-Bg [4,5]

[1] *Mic* - Microphone, *Con* - Contacts, *Loc* - Location
[2] *Fst* - First, *Lch* - Launch, *Nvr* - Never
[3] *Clk* - Click, *Bn* - Background w/Notif., *Bg* - Background Only
[4] Only used with *Coffee*
[5] Only used with *Lch*

Table 4.2: Possible values for each variable in tested conditions.

Button Click (*Clk*), Background with Notification (*Bn*, uses a resource without interaction, but displays an icon and short message in the notification drawer indicating some condition is met, such as being located near a coffee shop) and Background Only (*Bg*, uses a resource in a way not clearly shown in the UI). We can order these from most (*Clk*) to least (*Bg*) interactive. We always label the button clearly for its use (no deception). We do not test UI widgets besides buttons.

Regardless of a participant's assigned condition, the survey always asks about expectations for all resources and auxiliary actions. As a result, we implicitly collect data about users' expectations for the *Bg*-*Bg* int. pattern pair, with authorization pattern *Nvr*, for the two non-targeted resources in each condition. For example, a participant assigned to the *Coffee*-*Mic*-*Fst*-*Clk*-*Clk* condition also answers Likert questions about contacts and location that are analyzed within the *Coffee*-*Con*-*Nvr*-*Bg*-*Bg* and *Coffee*-*Loc*-*Nvr*-*Bg*-*Bg* conditions.

Testing the full-factorial combination of all variables was infeasible, so we eliminated conditions that were redundant or less relevant to our hypotheses, resulting in 42 final conditions. In more detail: We exclude conditions where the second int. pattern is more interactive than the first, as we assume the participant's second expectation would be dominated by the second int. pattern, rather than by the combination of patterns. We use *Bn* only in the first user action, because we are primarily interested in its effect on user expectations for the second user action. We assume expectations for *Bn* itself will depend entirely on whether the participant notices the passive cue, a topic that has been well studied [96, 97] but is somewhat orthogonal to our work. These two rules limit the int. pattern pairs we study to those in the last column of Table 4.2.

We exclude *Nvr-Bg-Bg* conditions because they provide no evidence of resource use, and are therefore identical to the implicit scenarios discussed above. We also exclude *Fst-Bg-Bg* because, in our experimental design, they are indistinguishable from *Lch-Bg-Bg*. In Table 4.2, *Bg-Bg* is highlighted in blue to indicate that it is only used with the *Lch* authorization pattern.

Because we do not comprehensively consider the effect of app type, we limit *Fitness* conditions to those we anticipated would have the largest variation in expectations. Specifically, we consider only *Loc* and *Mic* and only conditions where the two user actions exhibit different int. patterns. The *Fitness* scenarios therefore include all combinations of variables in Table 4.2 that not are not highlighted in blue or yellow.

4.5.3 Statistical Analysis

To test *H1* and *H2*, we primarily consider the expectations expressed by participants after the first user action. We use a logistic regression, appropriate for ordinal Likert data, to estimate the effect of the app, resource, authorization pattern, and int. pattern on participants' expectation. To test *H3*, which concerns the effect of the prior user action, we also use a logistic regression, with participants' expectation for the second user action as the outcome variable. We use the same input factors as before, this time including both int. patterns.

Each regression includes multiple observations of each participant: the explicit condition plus two implicit *Nvr-Bg-Bg* responses. As is standard, we include a mixed-model random effect that groups observations from the same participant [98].

Our initial model for each regression included the input variables plus all possible two-way interaction terms. To prevent overfitting, we tested all possible combinations of these inputs and selected the model with minimum Akaike Information Criterion (AIC), a standard measure of model quality [99]. We present only the final model for each regression.

4.5.4 Ecological Validity and Limitations

We use a controlled experiment with mock apps. While this limits ecological validity, it allows us to reason statistically about the effect of specific factors on participants' expectations, and to disregard factors such as participants' history with an app or reputation of the app's developer. In a study environment, participants may be less suspicious than if their real data were at risk. To partially account for this, we ask about expectation rather than comfort level [100].

By limiting our survey to two apps and restricting the int. patterns and resources tested, we likely miss factors, and particularly combinations of factors, that influence expectations. As one example, users likely expect SMS and Calls to have different usage patterns, and these expectations may vary with app type. However, based on the results of our app survey, we believe the variables chosen still provide useful insights.

Each participant sees two user actions in a relatively short time period. We do not study the effect of longer sequences of actions or long-term use (e.g., over days or weeks) of an app.

As is typical for online studies and self-reported data, some participants may not take the survey seriously, and some may try to complete the survey multiple times. We limit repeat participants using MTurk ID and browser cookies. While MTurk has generally been validated for high-quality data [101–104], U.S. MTurkers are somewhat younger and more male, tech-savvy, and privacy-sensitive than the general population, which may limit the generalizability of our results [105].

These limitations apply similarly across all conditions; we therefore consider comparisons among conditions to be valid.

4.6 User Expectations Study Results

We now present the results of our online user study. We found that *H1* holds: users were the most likely to expect a resource use when shown a more interactive int. pattern. In contrast, we observed that while resource type does affect user expectation, *H2* was not strongly supported. Finally, we found that *H3* does not hold. However, our results indicate that both background notification (*Bn*) and on-launch authorization requests (*Lch*) increase user expectation of future resource accesses.

For each of our hypotheses, we present key findings from our regression analysis. Summaries of our regressions are shown in Tables 4.3 and 4.4. Each table shows the included input variables and their values. Each variable includes a *base* *case* value (identified by dashes in the remaining columns). The odds ratio (OR) shows the observed effect of each value relative to the base case, measured as odds of expectation increasing one unit on our Likert scale. We also provide the 95% confidence interval (CI) and *p*-value for each measurement.

For example, the third row of Table 4.3 shows that switching from *Bg* to *Clk* in the first user action, all other variables held constant, is associated with a 106.3× likelihood of increasing one unit of expectation. The CI expresses 95% confidence that the "true" odds ratio is between 63.6 and 177.7. A *p*-value less than 0.05 is interpreted as statistically significant.

The second half of each table shows interactions between value pairs, given as the two values separated by a colon. These odds ratio indicate the change in likelihood when the two variables co-occur, relative to considering them independently. For example, the *Loc*:*Clk* odds ratio of 0.2 in Table 4.3 suggests the combined effect of these two values is *subadditive*: only 20% as strong as predicted by their individual effects.

4.6.1 Demographics

A total of 961 participants completed the study. Participants' ages ranged from 18 to 70+ years, with 37% age 18-29. Fifty-three percent reported being male and 47% female. ("Prefer not to answer" and "other" options for gender were provided.) Participants were required to be from the United States. Forty-five percent of participants reported holding a college degree, and 25% reported having

Variable	**Value**	**Odds Ratio**	**CI**	**_p_-value**
App	*Coffee*	–	–	–
	Fitness	1.3	[0.96, 1.78]	0.086
Int	*Bg*	–	–	–
	Clk	**106.3**	**[63.6, 177.7]**	**< 0.001***
	Bn	**4.1**	**[2.6, 6.7]**	**< 0.001***
Res	*Mic*	–	–	–
	Loc	**17.5**	**[13.4, 22.9]**	**< 0.001***
	Con	0.8	[0.6, 1.0]	0.056
Auth	*Nvr*	–	–	–
	Fst	**2.2**	**[1.2, 4.0]**	**0.008***
	Lch	**1.9**	**[1.2, 3.2]**	**0.008***
App:Res	*Coffee*:*Mic*	–	–	–
	Fitness*:*Loc	**0.4**	**[0.3, 0.6]**	**< 0.001***
	Fitness:*Con*	1.1	[0.8, 1.7]	0.546
Res:Auth	*Mic*:*Nvr*	–	–	–
	Con*:*Lch	**3.2**	**[1.5, 6.7]**	**0.002***
	Con:*Fst*	1.5	[0.6, 3.6]	0.41
	Loc:*Lch*	0.8	[0.4, 1.6]	0.487
	Loc:*Fst*	0.5	[0.2, 1.3]	0.166
Res:Int	*Mic*:*Bg*	–	–	–
	Loc*:*Bn	**2.4**	**[1.2, 5.0]**	**0.021***
	Con*:*Bn	**5.0**	**[2.3, 11.3]**	**< 0.001***
	Loc*:*Clk	**0.2**	**[0.1, 0.4]**	**< 0.001***
	Con*:*Clk	**0.2**	**[0.1, 0.4]**	**< 0.001***

*Significant effect – Base case (OR=1, definitionally)

Table 4.3: Summary of regression over participant expectations after the first user action.

“far above average” smartphone expertise. Each condition had at least 20 unique participants. Twenty people dropped out partway through the survey, distributed evenly across conditions.

Variable	**Value**	**Odds Ratio**	**CI**	***p*-value**
Int 2	*Bg*	–	–	–
	Clk	**810.4**	**[352.2, 1864.9]**	**$<$ 0.001***
Int 1	*Bg*	–	–	–
	Clk	1.0	[0.8, 1.4]	0.9
	Bn	**2.1**	**[1.5, 2.8]**	**$<$ 0.001***
Res	*Mic*	–	–	–
	Loc	**20.0**	**[15.5, 25.9]**	**$<$ 0.001***
	Con	1.0	[0.8, 1.3]	0.768
Auth	*Nvr*	–	–	–
	Fst	1.4	[0.9, 2.4]	0.174
	Lch	**1.7**	**[1.1, 2.7]**	**0.027***
Res:Auth	*Mic*:*Nvr*	–	–	–
	Con*:*Lch	**4.4**	**[2.2,8.5]**	**$<$ 0.001***
	Con:*Fst*	2.1	[1, 4.5]	0.056
	Loc:*Lch*	0.8	[0.4, 1.5]	0.445
	Loc*:*Fst	**0.5**	**[0.2, 0.9]**	**0.029***
Res:Int 2	*Mic*:*Bg*	–	–	–
	Loc*:*Clk	**0.1**	**[0.03, 0.3]**	**$<$ 0.001***
	Con*:*Clk	**0.03**	**[0.01, 0.1]**	**$<$ 0.001***

*Significant effect – Base case (OR=1, definitionally)

Table 4.4: Summary of regression over participant expectations after the second user action.

4.6.2 H1 – Interactivity v. Expectation

We found that *H1* holds: the more interactive the int. pattern, the more likely the user is to expect the resource access. In fact, interactivity (specifically *Clk*) is the strongest indicator of expectation we measured.

Table 4.3 shows that both *Clk* and *Bn* significantly increase the likelihood the user expects a resource access compared to *Bg*. The effect of *Clk* is particularly strong (OR 106.3, $p < 0.001$). Because the confidence intervals of *Clk* and *Bn* do not overlap, we can also conclude *Clk* generates significantly more expectation than *Bn*.

Table 4.4 confirms that the strong association between *Clk* and expectation holds for the second user action as well (OR 810.4, $p < 0.001$). Figure 4.5a illustrates this finding, showing that 90% of participants who saw a *Clk* int. pattern first definitely or probably expected the associated resource use, compared to 72% for *Bn* and 25% for *Bg*.

Explicit authorization, which is by definition interactive, is also associated with a significant increase in expectation. Compared to *Nvr*, both *Fst* and *Lch* have odds ratio effects of about 2× for the first user action (Table 4.3), both significant. This result makes intuitive sense, as the dialogs for both (*Fst* and *Lch*) occur very closely in time to the first int. pattern.

4.6.3 H2 – Real-World Frequency v. Expectation

We observed an inconsistent relationship between real-world frequency and expectation. Location was the most frequently seen resource in the app study (75 apps according to Table 4.1) and was also the most expected resource we tested. As shown in Tables 4.3 and 4.4, *Loc* was associated with 17-22× higher likelihood

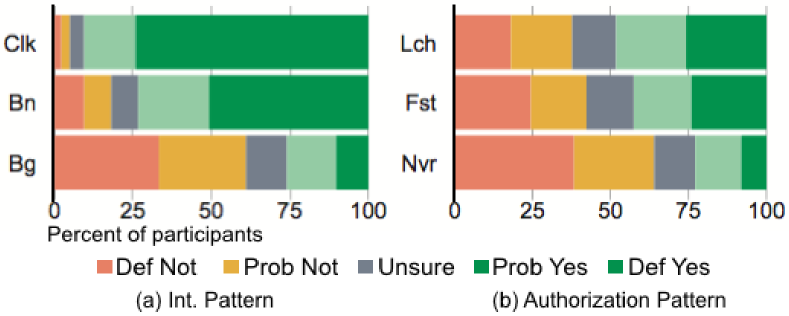


Figure 4.5: Likert-scale expectation responses for (a) the first user action, organized by int. pattern, and (b) the second user action, organized by authorization pattern.

of expectation compared to *Con* (seen in 30 apps) or *Mic* (seen in 14 apps). Non-overlapping CIs and p-values less than 0.05 indicate these effects are significant. This is consistent with *H2*; however, as our frequency measurement is only a very rough approximation of user familiarity, we note that this effect could relate to existing passive notifications for location, the higher volume of academic and media coverage of the location permission, or other factors.

However, *H2* does not hold when comparing *Con* and *Mic*. While *Con* was seen about twice as often in the app study, there was no significant difference in expectation between the two resources, at either the first or second user action. This could be because our app frequency metric does not sufficiently capture differences (or in this case, similarities) in users' overall exposure to each resource.

While there was no significant difference between the main effects of *Con* and *Mic*, we found evidence that users rarely expected *Mic* to be used with *Bg*. Looking at the *Res:Int* interaction in Table 4.3, *Loc*:*Bn* and *Con*:*Bn* both show significant superadditive results, indicating that these combinations are even more expected, relative to the baseline *Mic*:*Bg* combination, than those factors' main effects would predict. We hypothesize this difference is driven more by low expectation for background microphone access than high expectations for the other combinations. The interactions involving *Clk* are significantly subadditive, but we suspect this is a ceiling effect: expectations for *Clk*, which as shown in Figure 4.5a are very high, cannot increase beyond the end of our Likert scale. We see similar effects for the *Clk* interactions in Table 4.4.

4.6.4 H3 – Effect of Previously Seen Accesses

To examine the effects of prior accesses on participants' expectations, we focus on Table 4.4. Overall, we find that *H3* does not hold: participants are not more likely to expect resource accesses they have seen before. In particular, the int. pattern variable *Int1:Int2* does not appear in the final model, suggesting the combination of int. patterns does not meaningfully influence expectation at the second user action.

However, participants did appear more likely to expect a background access if they had previously seen an indication that background accesses might be used. Table 4.4 shows expectation at the second user action was significantly higher when the first int. pattern was *Bn* (OR 2.1, $p < 0.001$) or the *Lch* authorization request was shown (OR 1.7, $p = 0.027$), both of which indicate that background access may occur. Figure 4.5b illustrates this finding, showing that in the second user action, more participants definitely or probably expected resource usage in *Lch* conditions (47%) than in *Fst* (42%) or *Nvr* (22%). Additionally, the superadditive relationship between *Lch* and *Con* in both regressions may suggest a *Lch* request primes users to expect non-interactive accesses to contacts.

In contrast, we found evidence that the authorization pattern *Fst* implies only a single access. Table 4.4 shows that *Fst* did not have a significant effect ($p = 0.174$) compared to *Nvr*, indicating that an authorization associated with the first user action is no more effective than no authorization when thinking about a second user action. The subadditive relationship between *Loc* and *Fst* (OR 0.5, $p = 0.029$) also suggests that authorization requests associated with a specific event train users to only expect a resource access after an authorization request. This decreases the otherwise relatively high expectation of background location access. These relationships, while not particularly strong, are notable because they imply that the Android M model may in some cases be counterproductive; we explore this further in the Discussion section below.

4.6.5 App v. Expectation

Finally, we observed that the app type did have some effect on the way other variables were perceived. We found no significant difference between the two mock apps with respect to the first user action ($p = 0.086$), and app effects did not appear in the final model for the second user action, suggesting no meaningful relationship with expectation. However, we do observe in Table 4.3 a significant, subadditive relationship between *Fitness* and *Loc*, indicating that location accesses were less expected in the fitness app than the coffee app. We expect that across a wider variety of apps, further relationships between app type and expected resource usage would be observable.

4.7 Conclusions and Design Recommendations

Based on our app survey and user study, we propose several ways to improve Android's and similar authorization systems.

4.7.1 Access Resources As Interactively As Possible

Our app study found that camera, microphone, media, and calendar are already used almost exclusively interactively in popular apps. Moreover, our user study found that users overwhelmingly expect resource access after an explicit click on a related item. We speculate that users would also have higher expectations of resource access under other interactive uses.

Thus, we recommend expecting (or perhaps even requiring) most or all accesses to these four resources to be interactive. Other resources are more frequently used in the background, and it is not always obvious how to associate their uses with a foreground interaction. However, we argue that developers should prioritize (and the Android framework should encourage) making background uses more interactive when possible. For example, a social media app that periodically pulls the user's contacts to recommend new friends could tie this background access to a foreground interaction by asking the user to "turn on" this feature at launch and provide a settings menu where the user could turn the feature off at a later time. While this design is similar to the authorization mechanism provided by Android M, implementing it in the app provides context, which we have found is important to decision making.

We also recommend that when resources that are more typically used interactively will be used in the background, these uses should be documented explicitly in the app's description or a similar user-visible location.

4.7.2 Use Interactions to Grant Authorization

Our results further suggest that if a resource use is interactive, then a separate authorization dialog can be eliminated. We speculate that removing explicit authorization requests in these cases could reduce potential user confusion (e.g., "I just clicked 'Import Contacts,' why is it asking me if I want the app to access contacts?"). In addition, removing these requests could reduce annoyance and habituation, potentially helping the user to focus on other, less clear authorization decisions. Eliminating request dialogs for interactive use cases could also help motivate developers to prioritize interactivity, as mentioned above. Of course, to handle potentially malicious apps we must be sure the preceding interaction is clearly related to the resource use. For example, clicking on a location icon should not cause the camera to be used.

We envision two main approaches to enforce this principle. One idea is access-control gadgets [82, 91], which we discussed with Related Work. Another approach would be to leave apps as they are, but use a program analysis to ensure they conform. For example, we could use AppTracer to do so, in one of two possible modes. AppTracer could be run ahead of time, e.g., by an app market gatekeeper, to examine app behavior. Even though it would not necessarily observe all app behavior, results from analyzing behaviors users actually encounter would still be useful. AppTracer could also be used for auditing: power users and security experts could use AppTracer to log an app's behavior as they use it and then retroactively check the AppTracer graphs for any suspicious behavior, which could then be reported to the broader public.

4.7.3 Handle background authorization separately

We found that users were much less likely to expect background resource access if authorization dialogs were presented after a prior user action or were not presented at all. Thus, we recommend requesting authorization separately and explicitly for background uses. Based on our study, it may be preferable to do so when apps are first launched. However, because our study showed the increase in expectation is small (especially compared to the expectation after a click), it may be important to also show background notifications of use (which also increased expectation) so users remain aware. We note that while authorization on launch informs many users that background accesses should be anticipated, Figure 4.5 suggests there are others who do not recognize this possibility. Further research into the best approach is still needed.

Resources that have a broad mix of interactive and background uses—such as contacts—might particularly benefit from separate background authorization and limited requests for interactive uses. This could help avoid potential misconceptions about interactive uses being the only uses.

Future work could shed light on how differences in background use cases affect user expectations and preferences. For example, users might be expected to react differently when contacts are accessed in the background for pre-fetch (a use case identified in our app survey) compared to advertising. We also expect that the frequency of access will impact the user's decision. For example, it may be possible to tie high-frequency background uses to some foreground passive notification (e.g., a notification tray icon), similarly to the design presented by Balebako et al. for informing the user of data leakage [6]. This could make the user aware of the accesses without requiring additional authorization effort. Program analysis tools such as AppTracer could potentially be used to separate these cases and apply different authorization policies accordingly.

We thank the anonymous reviewers, This research was supported in part by NSF CNS-1064997, a UMIACS contract under the partnership between the University of Maryland and DoD, and a Google Research Award.

Chapter 5: Permission-Use Provenance in Android Using Sparse Dynamic Analysis

5.1 Introduction

Android apps can request permission to access a wide range of sensitive device resources such as contacts, calendar, GPS location, etc. However, if we wish to determine whether an app is secure, inspecting its permissions is insufficient, because permissions give apps unconstrained access to their corresponding resources. For example, an app that accesses the microphone whenever the app is running might be suspicious, while an app that accesses the microphone only after the "Mic" button is clicked may be secure. Yet both have the same microphone permission.

Thus, a better approach to understanding an app's security implications is to determine the *context* of sensitive resource uses [4, 8, 106, 107]: e.g., whether a use occurred after the user clicked a related button [23]. However, doing so today is difficult. Manual reverse engineering of app bytecode is time-consuming and error-prone. Program analysis of Android apps is difficult because Android is a large, complicated framework and Android apps are callback-oriented, and thus even an app's control flow is difficult to model. Moreover, existing program analyses

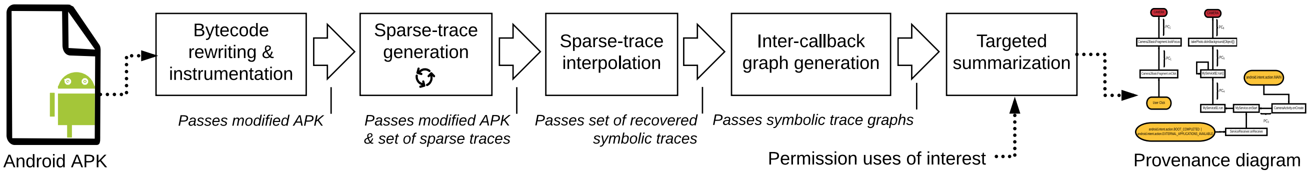

Android APK
Bytecode rewriting & instrumentation
Passes modified APK
Sparse-trace generation
Passes modified APK & set of sparse traces
Sparse-trace interpolation
Passes set of recovered symbolic traces
Inter-callback graph generation
Passes symbolic trace graphs
Targeted summarization
Permission uses of interest
Provenance diagram

Figure 5.1: Overview of Hogarth.

for Android focus primarily on finding taint flows [108–112] and common malware behaviors [113–117] rather than on explaining sensitive resource uses. (Section 5.6 discusses related work in more detail.)

This paper introduces Hogarth,[1] a prototype tool demonstrating a new approach to program analysis for sensitive resource accesses. Hogarth produces *provenance diagrams* showing the context of app permission uses. A provenance diagram is a graph where nodes are either API methods that need permissions, e.g., setAudioSource(...MIC), or callbacks invoked by the Android framework, e.g., a doInBackground method of an asynchronous task. There is an edge from node A to node B labeled with formula ϕ if A may call B when ϕ holds. Thus, provenance diagrams reveal deep information about the context of sensitive resource uses. For example, Section 5.2 describes a provenance diagram, inferred from android malware, in which a background photo capture is triggered by a command received from the network.

Hogarth uses a novel combination of dynamic analysis, symbolic execution, and abstract interpretation. More specifically, Hogarth performs dynamic analysis by instrumenting app bytecode so that, when an app is run, it produces a log *trace* of the app's execution. Then, Hogarth uses symbolic execution to replay the paths seen in the trace and infer *path conditions* capturing the conditions under which

[1] In the movie *The Iron Giant*, the character Hogart helps a robot come out of hiding. Our tool is designed to reveal potentially hidden Android permission uses.

each observed bytecode instruction occurred. During this process, Hogarth may introduce some overapproximation, similarly to abstract interpretation, to shrink the size of the path conditions and improve performance.

This combination of analyses enables Hogarth to produce provenance diagrams with very precise information for realistic apps. By using dynamic analysis, Hogarth restricts its focus to only feasible paths actually seen during program execution. But then symbolic execution and abstract interpretation enable Hogarth to generalize observed paths beyond the concrete values seen at runtime. The symbolic execution and abstract interpretation also allow the dynamic analysis instrumentation to be *sparse* so that instrumented apps remain responsive. (Section 5.2 gives an overview of our approach, and Section 5.3 describes provenance inference formally.)

Hogarth is implemented on top of Redexer [9], a Dalvik bytecode rewriting tool, and SymDroid [118], a Dalvik bytecode symbolic executor. To validate the Hogarth prototype, we applied it to a set of five apps, selected from F-Droid [24] and the Contagio Malware dump [25]. We found that Hogarth discovers provenance diagrams that match information one of the authors produced manually with a time-consuming reverse engineering effort. (Section 5.4 describes our implementation, and Section 5.5 discusses our evaluation.)

In summary, Hogarth introduces a new approach to automatically infer the provenance of a sensitive resource use, using a novel combination of dynamic analysis, symbolic execution, and abstract interpretation. The approach demonstrated in Hogarth has potential applications for app auditing, reverse engineering, and security evaluation more generally.

5.2 Overview

Figure 5.1 gives an overview of Hogarth's architecture. Its input is an Android APK file, which contains the app's Dalvik bytecode. Its first step is then to instrument the app so that, when run, it produces a *trace* of the app's execution. The trace is sparse in that it only logs key program points needed to reproduce the observed execution.

The user runs the modified app to produce a corpus of representative traces, and then Hogarth performs a process of *sparse-trace interpolation* to infer a *path condition* for every program point in traced callbacks. A path condition is a formula among inputs to the callback (both its arguments and data it receives from the Android framework) that holds if that program point is reached.

Next, Hogarth connects distinct handlers into a inter-callback graph, adding from callback A to callback B if A registered B. In this case, we say A is the *registrar*. Finally, Hogarth performs a *targeted summarization* step which coalesces all runs of a callback that preceeded a permission use of interest. This process yields a *provenance diagram*, which is a directed graph where nodes are either callback methods or API calls that need permissions. An edge from A to B labeled with path condition ϕ indicates that callback A may register callback B, or directly use permission B, under that condition (ϕ) on its inputs. Thus, we can use a provenance diagram to understand the circumstances under which a permission is used, both in terms of inter-callback control flow and conditions on app data.

```
class MyService$2 extends Thread { ...
  public void run() { ...
    while (true) { ...
      v14 = v12.readLine(); ...
      if (v14.contains("takephoto(")) {
        if (((String)v15.get(0)).equalsIgnoreCase("front(")) {
          ...
          new takePhoto(myService, FRONT).execute(new String[0]);
        } else
          new takePhoto(myService, BACK).execute(new String[0]);
        } } } }

public class takePhoto extends AsyncTask { ...
  private void initialiseCamera() { ...
    if (PackageManager.PERMISSION_GRANTED ==
        ContextCompat.checkSelfPermission(myService,
          android.Manifest.permission.CAMERA)) { ...
      cameraManager.openCamera(camId,mStateCallback,cameraHandler);
      cameraDevice.createCaptureSession(outputs,
      mccsStateCallback, cameraHandler);
  } ... } ...
  protected Object doInBackground(Object[] arg2) {
    return this.doInBackground(((String[])arg2));
  }
  protected String doInBackground(String[] arg5) {
    initialiseCamera();
    return "Executed";
  } }
```

Figure 5.2: Code excerpt of the malicious camera use.

5.2.1 Running Example

In the remainder of this section, we illustrate Hogarth on Camera2Evil, an app we produced by injecting the Dendroid [119] malware into the Camera2Basic Android example app [120], which allows the user to take a picture by pressing a button. Once Dendroid is injected into the app, it allows a remote command-and-control server to stealthily take a picture, without the user pressing a button, and upload it.

Figure 5.2 gives an excerpt from the Camera2Evil code, which is a combination of the open source Camera2Basic app for the Google sample library and the source of Dendroid was extracted using the JEB decompiler [121]. Here MyService$2.run, running in a background thread, listens for (adversary) commands. If it receives the right command, it creates a new instance of takePhoto, which is an AsyncTask, and executes it (callback registrar shown in blue). This queues up the takePhoto.doInBackground(Object[]) method to be called by Android in the future. Once invoked, this method calls initialiseCamera, which checks to see that the camera permission has been acquired already and, if so, it surreptitiously takes a photo (sensitive API calls shown in red).

5.2.2 Bytecode Instrumentation and Trace Generation

The first step of Hogarth's process is to add logging instrumentation to the Camera2Evil bytecode. We do so using Redexer [9], a Dalvik bytecode rewriting tool. More specifically, we insert a new method log(...) that writes its arguments and the current thread id to a ConcurrentLinkedQueue. We also insert code that creates a background thread to dequeue log entries, introspect on these data, and asynchronously write them to a file. This design helps ensure logging does not slow down the main app thread.

Then, we add calls to log(...) to record key events. More specifically, we insert code to record the method arguments (including the receiver) at every app method's entry. For example, we add logging to the beginning of doInBackground(String[]) that records the takePhoto receiver and the String[] argument. We also log the arguments and return values to every API call, e.g., the execute calls in our running example. Finally, we insert a call to log(...) at the entry of every basic block. Hogarth uses this information later in its process to recover exact control flow.

Figure 5.3 shows two portions of the trace generated by running the instrumented Camera2Evil app. The trace at the top is for MyService$2.run, which is an app method as indicated by the Method > line in the log. Hogarth infers that this is in fact a callback invoked by the Android framework because this trace entry is not nested inside any other method call entry. That same trace line lists the *thread id*, in this case 977, so that we can distinguish otherwise intertwined log entries from different threads. It also includes the arguments to the run method, in this case only the receiver object, which is recorded as the object's class and method (here abbreviated) followed by the *object id*, in this case 244505 (every object has a unique integer *id* in the Java runtime). Notice that we do not record object fields—we only use these values to match up callback registrations to the actual callbacks, as discussed below.

```
Method > 977 com..MyService$2.run(com..MyService$2@244505)
API > 977 java.io.BufferedReader.readLine(java.io.BufferedReader@84370)
API < 977 java.io.BufferedReader.readLine(java.lang.String@765419)
...
API > 977 java.lang.String.contains(java.lang.String@765419,
  java.lang.String@923754)
API < 977 java.lang.String.contains(java.lang.Boolean@490195)
BBEntry 977 1119442
...
API > 977 android.os.AsyncTask.execute(com..takePhoto@6720952,
  [Ljava.lang.String;@263773)

...

Method > 985 com..takePhoto.doInBackground(com..takePhoto
  @6720952,[Ljava.lang.String;@263773)
...
Method > 985 com..takePhoto.initialiseCamera(
  com..takePhoto@6720952)
...
API > 985 java.lang.Object.checkSelfPermission(
  com..MyService@1131473,java.lang.String@7204364)
API < 985 java.lang.Object.checkSelfPermission(
  java.lang.Integer@3382498)
BBEntry 985 1127344
...
API > 985 android..CameraManager.openCamera(
  android..CameraManager@1134037,java.lang.String@9683765,
  com..takePhoto$1@2286550,android.os.Handler@2206143)
...
API > 985 android..CameraDevice.createCaptureSession(
  android..CameraDeviceImpl@122346893,java..ArrayList@122060324,
  com..takePhoto$3@367746,android.os.Handler@2206143)
...
```

Figure 5.3: Two partial sparse traces for Camera2Evil.

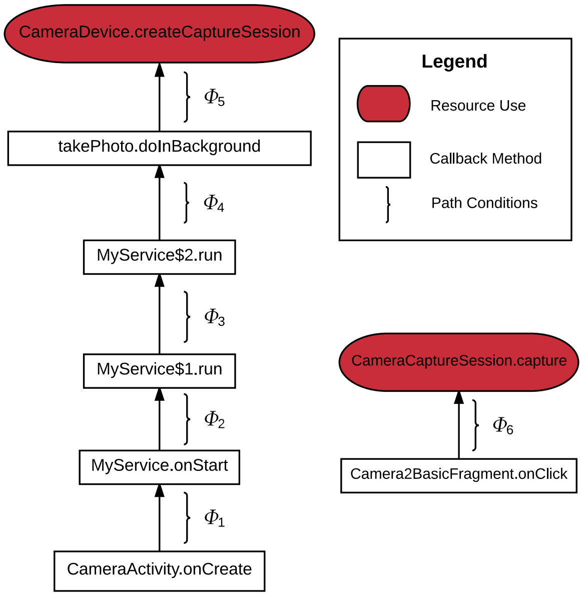
The next lines in the top portion of the trace record the call to (API > 977) and return (API < 977) from readLine. Redexer marks a call API whenever it cannot find the target method in the app's class definitions. During sparse trace interpolation, API call results are represented abstractly so that path conditions may be in terms of API calls (details below).

Then after a call/return pair to contains (on line 5 in the code), the line BBEntry 977 1119442 records the entry to basic block number 1119442, where the number is assigned by Hogarth. In this case, reaching that basic block tells us that the true branch of the first conditional (the result of testing contains) was taken. We will use this information below to build up a path condition. After some additional entries, this portion of the trace ends with a call to execute.

The bottom portion of Figure 5.3 shows a portion of the log from the subsequent call to takePhoto.doInBackground. Notice that this may occur an arbitrary amount of time later in the log. The log entries are similar to above, with some API calls, a basic block entry that records taking the true branch of the condition in initialiseCamera, and calls to two methods needing the camera permission.

5.2.3 Sparse Trace Interpolation

Next, Hogarth performs *sparse trace interpolation* by replaying the logs using a hybrid of abstract interpretation [122, 123] and symbolic execution [124, 125] to derive a *path condition* at every point, within app methods, reached while the trace was generated. Critically, Hogarth can do so even though the logged information is



ϕ_1	true
ϕ_2	true
ϕ_3	((equals(“goldfish”,staticfd_119) == 0) & (startsWith(staticfd_118,“generic”) == 0) & (startsWith(staticfd_117,“generic”) == 0) & (equals(“google_sdk”,staticfd_120) == 0) & (equals(“google_sdk”,staticfd_121) == 0) & (!(booleanValue(var872550) == 0)) & (getBoolean(getDefaultSharedPreferences(getApplicationContext(var830452)), “Start”,0) == 0))
ϕ_6	(getId(argument11) == #x7f0a0059)

Figure 5.4: Camera2Evil: Partial camera-use provenance diagram.

sparse, in particular it only includes method entries and exits, API calls, and the start of each basic block.

Hogarth implements sparse trace interpolation on top of SymDroid [118], a Dalvik bytecode symbolic executor. For each callback observed in the trace (in our example, MyService$2.run and takePhoto.doInBackground), Hogarth steps through the method's instructions (and the instructions of any app method called), following the path seen in the trace and building up a path condition in terms of the callback method's inputs and the return values from API calls and field accesses (the latter not shown in this example).

For example, Hogarth will start stepping through `MyService$2.run`. The receiver this will be bound to an *abstract value* that records the actual value seen in the log, in this case `com..MyService$2@244505`. As Hogarth continues, it builds more complex abstract values to encode primitive operations and represent returns from framework calls, as necessary. For example, Hogarth will eventually reach the call to readLine. For simplicity, suppose that local variable v12 is bound to an abstract value of the same name (*v12*). Then after the call, v14 will simply be bound to the abstract symbolic value *BufferedReader.readLine*(*v12*), where the receiver is listed as the first argument.

Whenever there is a branch, Hogarth follows the path taken in the trace and records which way the branch test went in the path condition. For example, recall that the trace took the true branch when testing the return value of contains. Thus, Hogarth will take the same path, stepping into the then branch of the conditional, and it will conjoin a symbolic value

String.contains(BufferedReader.readLine(v12), ..@923754)

to the path condition. As it continues execution, Hogarth will eventually reach the `execute` call on line 8 and determine that the following path condition, which we will refer to as ϕ_4, holds at the call:

(!(isConnected(getActiveNetworkInfo(getSystemService(
getApplicationContext(var832364),“connectivity”))) == 0)) &
(!(getActiveNetworkInfo(getSystemService(getApplicationContext(var832364),

“connectivity”)) == 0)) & (!(readLine(newClsAt834888) == 0)) &

(find(matcher(compile(“\((([ˆ)]+)\)”),

readLine(newClsAt834888))) == 0) &

(!(contains(readLine(newClsAt834888),“takephoto(”) == 0)) &

(equals(group(matcher(compile(“\((([ˆ)]+)\)”),

readLine(newClsAt834888)),1),“”)==0)

(!(equalsIgnoreCase(get(newClsAt838062,0),“front(”) == 0)) &

We discuss the path condition in more detail below.

Similarly, when Hogarth executes takePhoto.doInBackground, it will eventually reach the call to createCaptureSession with the path condition ϕ_5:

(! (getCameraIdList(getSystemService(

getApplicationContext(var862168),“camera”)).length <= 1)) &

(checkSelfPermission(var862168,“android.permission.CAMERA”) == 0)

We also discuss this path condition below.

5.2.4 Inter-callback Graphs and Summarization

Hogarth’s final steps are to connect distinct callback traces and summarize them to construct the provenance diagram from coalesced callbacks and their inferred path conditions.

Figure 5.4 shows a portion of the provenance diagram for Camera2Evil. The diagram is constructed as follows. First, by default Hogarth assumes that all permission-related API calls observed in the trace are of interest. Hogarth adds a node to the graph for each such method, in this case we have createCaptureSession and capture. We will begin by focusing on the createCaptureSession, which is the malicious camera use case. That method was called while doInBackground was executed, so Hogarth also adds doInBackground to the graph and adds an edge between the nodes labeled with the path condition, ϕ_5, that held when the call occurred. If there were multiple such path conditions, e.g., because createCaptureSession was called inside doInBackground multiple times, Hogarth labels the edge with the disjunction of the path conditions.

Next, Hogarth determines which callbacks were triggered by which other callbacks. For example, recall that in Camera2Evil, the run method scheduled doInBackground. Hogarth finds these dependencies using a heuristic: We use the EdgeMiner [126] data set to identify which API calls might register callbacks. Then if the trace contains a call to a registration method that passes some object which is later used as the receiver object in a callback, Hogarth assumes the registration caused the callback and adds an appropriate edge to the graph.

For example, EdgeMiner reports that AsyncTask.execute is such a registration method. The trace in Figure 5.3 for MyService$2.run contains a call to execute (in blue) with object @6720952 as an argument. Subsequently in the trace, the doInBackground method is invoked with @670952 as the receiver. Thus, Hogarth adds an edge to the provenance diagram from MyService$2.run to doInBackground, labeled with the path condition, ϕ_4, that held when execute was called.

In addition to using EdgeMiner to determine callback connections, there are some cases where this is not possible (e.g. Intent passing) and we have to manually establish these connections. We discuss this further in Section 5.4.

Hogarth continues this process, adding callbacks to the graph until it reaches a fixpoint. We can then use the resulting graph to investigate under what circumstances permissions are used. If we trace back the diagram in Figure 5.4, we see that when the app launches (onCreate), it starts the malicious service (MyService.onStart) which kicks off another thread (MyService$1.run). Both ϕ_1 and ϕ_2 are shown as true because these transitions through the program will always occur after the app is started. In MyService$1.run, ϕ_3 tells us that the malware checks its DefaultSharedPreferences for two settings, whether it should start and if its Google Play bypass feature is enabled. If the bypass feature is enabled, then it checks the build information on the device to determine whether the app is running on an emulator, i.e. the product or model are “google_sdk”, the brand or device are “generic” or the hardware is “goldfish”. If ϕ_3 holds, then a second thread MyService$2 is started to connect to the adversary’s command-and-control server. ϕ_4 states that if the device has an Internet connection (either wifi or cellular), it receives some commands from the server, and the string “take photo(front(” is included in the commands, then it begins a new asynchronous task to take a photo and send that back to the server (takePhoto.doInBackground). Finally, ϕ_5 states that if there is at least one camera available on the device and the app has been granted the permission “android.permission.CAMERA”, then the malware calls CameraDevice.createCaptureSession to surreptitiously take a picture.

Similarly, Hogarth generates the provenance diagram for the instance of createCaptureSession on the right, the benign case. In the benign case, when onClick is triggered by a user click, ϕ_6 states that if ID of the button that was clicked is equal to #x7f0a0059 (the ID of the "take picture" button), then CameraCaptureSession.capture is called to take a picture. Using these provenance diagrams auditors can better understand the conditions of a resource use and differentiate between the malicious and benign cases.

5.3 Provenance Inference

This section gives a formal presentation of the analysis just described. To keep our presentation compact, we show how to infer provenance using the simplified Dalvik bytecode language in Figure 5.5. Here and below, we write $\vec{x}$ for a sequence of zero or more x's, and we write x_i to signify the ith element of such a sequence (starting from index 0).

In this language, based on SymDroid's μ-Dalvik intermediate representation, a program *prog* consists of a sequence of class definitions $\overrightarrow{class}$. A single class definition consists of a class name C, its superclass, a sequence of method definitions $\overrightarrow{method}$, and a sequence of field names $\vec{f}$. Each method definition includes the method name m, a sequence of registers $\vec{r}$ for the formal parameters, and a sequence of instructions $\vec{i}$ for the method body.

Instructions are fairly standard. An unconditional jump goto j sets the program counter so the instruction at index j is executed next. A conditional jump if r then j else branches to instruction j if the contents of register r is true. Assignments of the form $r \leftarrow c$ write a constant integer, string, or boolean into register

$$
\begin{array}{rcl}
\mathit{prog} & ::= & \overrightarrow{\mathit{class}} \\
\mathit{class} & ::= & C <: C\ \overrightarrow{\mathit{method}}\ \vec{f} \\
\mathit{method} & ::= & m(\vec{r})\ \vec{i} \\
i & ::= & \mathsf{goto}\ \ j \mid \mathsf{if}\ r\ \mathsf{then}\ j\ \mathsf{else}\ \ \mid r \leftarrow c \mid r \leftarrow r \mid r \leftarrow r \oplus r \\
 & \mid & r \leftarrow r.f \mid r.f \leftarrow r \mid r \leftarrow \mathsf{new}\ C \mid r \leftarrow r.m(r, \ldots) \\
 & \mid & \mathsf{ret}\ r \\
c & ::= & n \mid \mathit{str} \mid \mathit{true} \mid \mathit{false} \\
\oplus & ::= & \{+, -, *, <, \neg, \wedge, \vee, \ldots\}
\end{array}
$$

$$
\begin{array}{rcl rcl}
C & \in & \mathit{classes} & m & \in & \mathit{methods} \\
f & \in & \mathit{fields} & r & \in & \mathit{regs} \\
n & \in & \mathit{integers} & \mathit{str} & \in & \mathit{strings}
\end{array}
$$

Figure 5.5: Simplified Dalvik bytecode.

r; assignments $r_1 \leftarrow r_2$ copy r_2 to r_1; and assignments $r_1 \leftarrow r_2 \oplus r_3$ apply some operation $\oplus$ to r_2 and r_3, storing the result in r_1. Fields are read with $r_1 \leftarrow r_2.f$ and written with $r_1.f \leftarrow r_2$. Allocation $r \leftarrow \mathsf{new}\ C$ creates a fresh instance of class C and stores a pointer to it in r. Method invocation $r \leftarrow r_0.m(r_1, \ldots)$ performs dynamic dispatch of method m with the given receiver r_0 and arguments $r_1, \ldots$, assigning the result to r. Lastly, $\mathsf{ret}\ r$ exits the current method, returning r.

Note that this language omits many features of Dalvik bytecode, such as arrays, static methods and fields, etc. We discuss details of handling full Android apps in Section 5.4.

5.3.1 Sparse Trace Generation

As previously discussed, provenance inference begins by instrumenting and executing the program to gather a set of dynamic traces. In our implementation (Section 5.4), we modify the app's bytecode to add tracing instrumentation. Here we elide that step, and simply describe the trace this instrumentation yields.

$$\begin{array}{rcll} pt & ::= & \vec{ct} & \text{[program trace]} \\ ct & ::= & C.m(\vec{v})\ \vec{ti} & \text{[callback trace]} \\ ti & ::= & C.m & \text{[app call]} \\ & \mid & C.m(\vec{v}) & \text{[API call]} \\ & \mid & \texttt{then} \mid \texttt{else} & \text{[branch]} \\ v & ::= & \epsilon \mid C@id & \text{[value]} \end{array}$$

Figure 5.6: Dynamic traces.

Figure 5.6 gives a grammar for *program traces* pt, which consists of a sequence of *callback traces* $\vec{ct}$. Each callback trace records what happens from the time Android invokes an app callback to the time the callback returns to the framework. A callback trace $C.m(\vec{v})\ \vec{ti}$ records the class C and method m called by the framework, along with the argument values $\vec{v}$, where v_0 encodes the method receiver. Each value is either ignored, written ϵ, or a class C paired with an id. In our implementation, we log all constants as ϵ for performance reasons (specifically, writing all strings to the trace is expensive). Notice that we do not record object fields—we only use these values to match up callback registrations to the actual callbacks, as discussed below.

Each callback trace also includes a sequence of *trace items* $\vec{ti}$ that occurred during the callback. There are four kinds of trace items. First, $C.m$ logs a call to an app method m of class C. We elide arguments because these will be recovered via symbolic execution. Second, $C.m(\vec{v})$ logs a call to an API method m of class C. In this case, we do record the argument values $\vec{v}$ since they may include possible callback registrations. Lastly, `then` and `else` log which way each `if` instruction branched. These model the `BBEntry` trace entires in Section 5.2.

5.3.2 Sparse Trace Interpolation

Next, Hogarth uses a hybrid abstract interpretation and symbolic execution to simulate the app bytecode, in order to infer path conditions describing the paths seen in the trace. Figure 5.7 formalizes this process as a series of operational rules over machine *states* $\langle \vec{i}, \mathit{rf}, \sigma, \kappa, \phi, \vec{ti} \rangle$. Here $\vec{i}$ is the sequence of instructions remaining to be executed. The register file rf maps registers to *abstract values* a, which include constants, object ids paired with classes, and operations among abstract values, which are standard. Abstract values also include r, which stands for the value read from a register (here, always a parameter to the method); $a.f$, which stands the value read from a field of an object; and $C.m(\vec{a})$, which stands for the value returned from an API call with arguments $\vec{a}$ (where, again, the first argument is the receiver object). These last three forms allow Hogarth to track the conditions on "inputs" to the callback from parameters or from values returned by the Android framework.

A machine state also includes the store/heap σ, which maps abstract values (representing object locations) and field names to abstract values. The stack κ is a (possibly-empty) sequence of triples $(r, \mathit{rf}, \vec{i})$, where r is the register to be written upon returning, rf is the previous register file to reinstate, and $\vec{i}$ is the sequence of instructions to resume upon returning. Finally, the path condition ϕ is a (boolean) abstract value, and $\vec{ti}$ is the trace to be followed.

Figure 5.7b lists the machine's operational rules. [Jump] replaces the instruction sequence with those at the target address, using helper function $\mathit{instr}(j)$ (not formalized). [Then] and [Else] handle conditional branches, using the observed trace to guide the machine. When the head of the trace is `then`, the concrete execution took the true branch, so [Then] conjoins the branch condition $rf(r)$ to the current path condition ϕ (meaning the abstract value at $rf(r)$ must correspond to true) and jumps. When the head of the trace is `else`, execution fell through, so [Else] simply steps past the branch instruction and conjoins the negation of the branch condition with the path condition.

[AssnC], [AssnR], and [AssnOp] update the register file so the left-hand side register r maps to the given constant, register contents, or operation, respectively. Fields are stored lazily, so when they have never previously been read or assigned, they will not have a mapping in the store. [AssnF] looks up the contents of a field that has been previously written into the store. [AssnFExt] handles the case when $f \notin dom(\sigma(a))$, *i.e.*, when some field is accessed that was not previously written by the current callback. In this case, the field was written by some external code (either in Android or a different callback), and Hogarth binds abstract value $a.f$ to the left-hand side to represent the field read. [FWrite] updates the store so that the abstract object pointed to by $rf(r)$ has an f field that maps to the symbol denoted by the right-hand side $rf(r')$. [New] handles an allocation, in which a fresh *id* (meaning one not chosen before and not in the trace) is paired with C and bound in the register file.

[Call] handles an invocation of one of the app's methods. At these control points, the log has recorded $C.m$, the class and method that was invoked at run time. Thus, the rules use *lookup* (not formalized) to retrive the corresponding method definition. [Call] then pushes the return register, the current register file, and the remaining instruction sequence onto the stack. It then begins executing the callee's instructions under a new register file mapping the formal parameters to the actual arguments. [Ret] handles a return by popping the stack frame and updating the return register. Finally, [API] binds an abstract value representing the call.

To derive path conditions for a program trace $pt = ct_0, ct_1, \ldots$, for each $ct_j = C.m(\vec{v})\ \vec{ti}$, we begin with an entry-point state:

$$\langle \mathit{lookup}(C, m), \mathit{rf}[r_i \mapsto a(r_i, v_i)], \varnothing, \epsilon, \mathit{true}, \vec{ti}\rangle$$

We begin executing the body of $C.m$ with a register file where each formal parameter r_i is bound to $a(r_i, v_i)$, where $a(r_i, C@id) = C@id$ and $a[(r_i, \epsilon) = r_i$. The initial store and stack are empty, the initial path condition is *true*, and the initial sequence of trace items comes from the callback trace. We write $ct_j \rightsquigarrow^* \mathit{state}$ if *state* is reachable in zero or more steps of the machine starting in the initial (entry-point) state for ct_j.

5.3.3 Inter-callback Graphs and Summarization

Finally, we can build the provenance diagram. We begin by first constructing an *exploded graph* as follows. Let $pt = ct_0, ct_1, \ldots$ be the program trace. Initially, we add to the graph every API method $C.m$ that requires a permission of interest. Then, until we reach a fixpoint, we pick a node $C.m$ in the graph and add an edge

$C'.m' \xrightarrow{\phi} C.m$ for every ϕ, either: (1) such that we have $C'.m'(\vec{v'})\ \vec{ti} \in pt$ and

$$C'.m'(\vec{v'})\ \vec{ti}' \rightsquigarrow^* \langle \_, \_, \_, \_, \phi, C.m(\vec{v}) : \vec{ti} \rangle$$

for some C, m, $\vec{v}$, and $\vec{ti}$, or, (2) such that $C.m(\vec{v})\ \vec{ti} \in pt$ and there exists, in any recovered trace, a state

$$\langle \_, \_, \_, \_, \phi, C'.m'(\vec{v}') : \vec{ti}' \rangle, \text{where } v_0 = v'_i, \text{for some } i$$

This first case handles adding edges from callbacks ($C'.m'$) to API calls that preceed target permissions. At the first iteration of this process, all these API calls will be the target permissions themselves. The second case handles connecting callbacks ($C.m$) back to the API calls that may register them[2]. This is done by making an edge back to registrar calls that pass the receiver object (v_0) as some argument (v'_i). After this, case (1) will connect these registrars back to their callbacks and the process continues to work backward. In total, this fixpoint computation builds a directed graph of callbacks and API calls that lead to a target sensitive resource use.

The exploded graph may have multiple edges, with different path conditions, between the same pair of nodes. To create the final *summarized graph* we combine these edges. For every connected pair of nodes $C'.m'$ and $C.m$ we consider all their

[2] calls that could be a callback registration according to the EdgeMiner database

path conditions

$$C'.m' \xrightarrow{\phi_0} C.m \qquad C'.m' \xrightarrow{\phi_1} C.m \qquad \ldots$$

and replace them with a single edge

$$C'.m' \xrightarrow{\phi_0 \vee \phi_1 \vee \ldots} C.m$$

Our final provenance diagrams are simply these summarized graphs where each pair of sequential callback and registrar is rendered as a single node.

5.4 Implementation

We implemented Hogarth by building on top of Redexer [9] and SymDroid [118]. Our extensions were written in Java (for the logging machinery) and OCaml (for additions to Redexer and Symdroid), and required on the order of 10,000 lines of code. There were several unique challenges in implementing our approach for full Android apps.

5.4.1 Logging instrumentation.

We observed that, in practice, it is crucial to ensure Hogarth's instrumentation does not affect app performance too much, especially in the UI thread, since Android kills applications whose UI thread becomes non-responsive. Thus, our inserted log calls perform no introspection themselves. They simply add messages to a ConcurrentLinkedQueue, which uses a wait-free algorithm to communicate with a separate worker thread that retrieves messages from the queue to produce the trace. In total, the message passing interface adds between 10 and 20 Dalvik instructions for each inserted log call.

Recall that we log only app code and not Android framework code. For performance, we also excluded methods in android.support.* and allow the user to specify other paths to exclude. In these cases we mark such methods as API calls and do not add logging instrumentation inside them. In practice this is critical for scalability, since it greatly reduces the overall amount of logging performed.

In our formalism, API calls do not themselves carry an internal trace. However, that is insufficient in practice. For example, consider java.util.Collections.sort, which we treat as an API method. When called, it in turn will call the app's compare method for objects passed in, which will have logging calls inside it. To handle this case, our implementation maintains a *phantom context* representing the unknown portion of the stack inside the API call. Our logging instrumentation then records both the entry and exit of every API call (unlike app methods) so that we know how to delineate these phantom contexts. Otherwise, we might be unsure whether encountering log items from user code meant the API call had returned or simply made another call back into app code.

5.4.2 Sparse Trace Interpolation.

In addition to the language features formalized, Hogarth must handle dynamic dispatch for instance methods, static methods, static fields, and primitive arrays.

Dynamic dispatch is performed on a receiver object by looking at the next basic block address encountered in the log. Static methods and fields are handled in a manner analagous to instance methods and fields. An initial symbol for static fields is set in the store and replaced at each assignment as would a register or instance field's symbol.

Recall that the abstract values in Figure 5.7a have a fairly rich structure. As a fairly typical symbolic executor, SymDroid only directly supports constants, symbolic variables, and operations among symbolic expressions. Thus, Hogarth encodes registers, field accesses, and method calls as symbolic variables with special names. For example, an API call value $C.m(a)$ where C is a *BufferedReader*, m is a *readLine*, and a is a `new` C', may be encoded as a symbolic variable named BufferedReader.readLine(newClsAt534998).

One issue with this encoding is that, in the presence of loops and recursion, we might wind up reusing a symbolic variable name. This could cause the same symbolic variable to stand for multiple values, which might then yield multiple, possibly contradictory branch conditions. To sidestep this issue, we observe that path condition clauses relevant for permission uses typically do not involve variables that change with loop iterations. Thus, if Hogarth is in a loop and is about to reuse a symbolic variable already in the path condition, it heuristically removes all clauses involving that variable before reusing it. In practice this means path conditions will only include information about the last iteration of a loop, which in our experience is sufficient.

As we developed Hogarth, we found that path conditions often contain many abstract values for arrays. In practice those values were uninteresting and their presence in the path conditions decreases performance significantly. Thus, our implementation includes a special abstract value $\top$ that represents any possible abstract value, and we model all arrays as $\top$. Constraints on $\top$ are discarded and not added to the path condition, and complex abstract values involving $\top$ are simply reduced to $\top$, e.g., $\top.f$ is $\top$.

One last issue in sparse trace interpolation is invocations of <clinit> methods, which are invoked whenever the Dalvik Virtual Machine decides to load a class. Thus, there is no syntactic call site for such calls, and therefore they do not really fit well in a provenance diagram. We opted to simply elide such calls from our analysis. To do so correctly, we add extra logging to the end of <clinit> methods so we know when they exit (whereas we do not log the return of other app methods).

Inter-callback connections. Recall that we use the EdgeMiner [126] database to identify possible registrar methods, and then we connect up a registrar with a callback if the receiver of the latter was an argument to the former, using the object *id* for comparison.

While this approach is largely successful, there are a few cases where Android reuses the same object for different callbacks, particularly Intents and Threads; thus we cannot rely on their object *id*s. We address this issue by using a different *id* in these cases. For Intents (which are essentially key-value maps), we add a *magic id* field that gets a fresh value each time, and use that in place of the object *id*. For Threads, we use the thread *id* in place of the object *id*.

					# Triggers		**# Clauses**		
App	**Resources**	**# Log Lines**	**Time (s)**	**Mem (Mb)**	RE	Found	RE	Found	Missed
Camera2Evil	Camera	12,298	97	919	2	2	14	30	0
Call Recorder	Microphone	4,520	1	799	1	1	5	5	0
Misbothering	Contacts	674	1	694	1	1	3	3	0
Contact Merger	Contacts	602,422	38	1,606	4	1	8	18	1
Smart Studio Proxy	SMS	3,151,374	18	1,606	4	1	4	14	10

Table 5.1: Performance metrics for Hogarth when run across our apps. For number of triggers, RE indicates ground truth, and Found indicates how many Hogarth found. For number of clauses, RE indicates the clauses identified by our reverse engineer, Found indicates the number of clauses identified by Hogarth (which may include some deemed irrelevant in the manual analysis), and Missed indicates clauses identified by the manual analysis but not Hogarth.

5.5 Evaluation

We validated Hogarth on a set of five apps, in order to a) check that it worked correctly and b) evaluate the impact of its approximations on the end result. We obtained ground truth by using five moderately-sized apps that were manually reversed engineered by the third author, who has professional experience reverse engineering Android apps. While reverse engineering these apps, we generated a provenance diagram in the style of Figure 5.4. Then, we ran Hogarth on each app to ensure that it produced the same graph of handlers and correctly inferred the path conditions leading to each registration and permission use.

For our analysis, we selected five apps that use permissions in the background. The first app is our running example, the Camera2Evil app. Three others are benign apps from the open-source F-Droid [24] repository. We include an additional malicious app from the Contagio Malware dump [25].

To obtain ground truth, we first manually identified the permission uses of interest within the app's source code. Using our knowledge of the Android framework, we then examined all of the app's source code to trace backwards and determine how each method could be reached via handler registrations. As we went back through the app, we also collected path conditions we believed were relevant to the use of the permission. These results were synthesized into a provenance diagram for each app.

Next, we ran Hogarth on each app to generate a provenance diagram. We compared the results of Hogarth with our manually-generated provenance diagram.

We also evaluated Hogarth's performance on each of our apps by measuring the running time and memory consumption for each app we analyzed. Table 5.1 includes the performance numbers for the apps we tested. Because Hogarth's scalability is impacted by the size of the log being analyzed, we include the number of log entries, the running time, and maximum amount of memory consumed. Our tests were performed on a machine with a 3.5 Ghz Core i7 processor and 16 gigabytes of RAM running Ubuntu 16.04.2 LTS. Runtime results were averaged over the course of five runs. Hogarth currently handles app logs of around 50-100 MB before it runs out of memory in processing them. This is because the first step in our analysis is to generate an interpolated state corresponding to each run in the log. In a future version of Hogarth we plan to summarize apps in concert with performing log interpolation, allowing us to scale to much larger logs.

Using this procedure, we found that Hogarth correctly inferred the path conditions for the logs we used. Below, we walk through each example app and discuss how Hogarth works on each.

5.5.1 Camera2Evil

The first app in our validation study was our running example, Camera2Evil. As discussed in previous sections, Camera2Evil has two triggers leading to the use of the CAMERA permission. One of them is the benign behavior, as a result of clicking a button, and the other was the malicious trigger, which is invoked when the app receives a command from the control server.

Hogarth generated a diagram with the same structure as the one we produced by hand, but with some extra clauses in the path conditions. For example, the Hogarth path conditions included some additional clauses that our expert did not identify as relevant to the permission use. These included a few clauses that stated invariants about the lengths of various arrays that the app iterates over.

Hogarth successfully discovered the path leading to the malicious permission use, which happens when a line received from the server contains the string "takephoto(front(." Camera2Evil makes a call to BufferedReader.readLine until it finds a string matching the specific name. As we discussed in Section 5.4, Hogarth uses a strategy to allocate symbolic names in the presence of loops. This helped ensure that Hogarth correctly included the last iteration of the loop in the summary path condition.

5.5.2 Call Recorder

The Call Recorder app [127] allows users to record calls and store a copy on their device. The app registers an Android broadcast receiver [128], which will receive messages from the Android system whenever a phone call takes place and start recording accordingly.

To accomplish this, Call Recorder registers (in its manifest) for a handler named MyPhoneReceiver.onReceive to listen for phone call events. When the user receives a phone call, the Android system sends a message to the app, which the onReceive handles. Within this handler, the app checks that external storage is mounted, that a phone call is currently underway (off-hook), and that a user-controlled flag enabling recording is set. If these conditions are met, the app then creates an intent to start an Android Service, named RecordService. Within this service, in the RecordService.onStartCommand method, the app checks that a phoneNumber field is non-null and that no other recording is currently in progress. If both these conditions are true, the app begins recording the user's audio by callingMediaRecorder.start. This use of the RECORD_AUDIO permission happens outside of the app's UI.

Hogarth infers a provenance diagram leading to the RECORD_AUDIO permission and rooted at the MyPhoneReceiver.onReceive handler. (Note that Hogarth does not currently include machinery to detect which event type—in this case phone calls—has been registered in the manifest, but we can observe this via manual inspection of the manifest.)

Hogarth correctly infers an edge from onReceive to RecordService.onStartCommand, the background service that performs the recording, and correctly infers all path conditions for this edge: that the user-controller flag has been set, that external storage is mounted, and that the phone is off-hook.

With respect to the edge from RecordService.onStartCommand to MediaRecorder.start, Hogarth again correctly infers the path condition: that the phone number is non-null and no other recording is in progress. In total the manual diagram and the Hogarth diagram matched.

5.5.3 Misbothering

Misbothering SMS [129] is an app that mutes notification for any message sent by a user not in the contacts list. It works by checking the sender of all incoming SMS messages and only notifying the user if the sender is the user's contact list. We used Hogarth on Misbothering and were correctly able to discover the circumstances under which it accesses the user's contacts. Upon receiving an SMS, the app enters a handler, which executes a set of checks to ensure that particular app variables are initialized correctly. The handler also checks that a non-empty message was received. The Hogarth diagram matches the diagram generated by our expert for this handler.

Note that Misbothering registers a receiver for text data from the system. While Hogarth observes that this handler is invoked, the handler is registered via the app's manifest. As in the call recorder app above, Hogarth does not detect manifest registrations, and so cannot detect which event is tied to the handler, but this can be obtained by inspection.

5.5.4 Contact Merger

The Contact Merger app [130] helps identify duplicate contacts by analyzing a user's contact list and recommending entries that may refer to the same person. Access to the user's contacts can happen in one of five situations: a direct interaction in the form of a button click, on app startup, a timer alarm that triggers in the background at a one-hour interval, a device reboot, or a new app package being installed.

These five situations are associated with four triggers (reboot and package install are triggered by the same broadcast receiver). Of these, the timer and the reboot/install are not covered by our logs, which did not include a restart or a new package or last an hour. Incomplete logs, of course, are a limitation of dynamic analysis; this app demonstrates the importance of capturing longer logs containing a wider variety of events.

Hogarth correctly identified one of the two remaining triggers: app startup. For this trigger, Hogarth identified path conditions to the READ_CONTACTS permission including that the app's internal contacts database has been set up (the source code indicates this is a workaround for a bug), that a specific file (used to store the contacts) exists, and that the contact analysis has not yet been performed. Hogarth also gathers some additional path condition clauses that our reverse engineer deemed irrelevant, such as requiring that a particular loop along the path to READ_CONTACTS has successfully exited.

Hogarth does not find the click trigger because its Android system model is incomplete: the click trigger uses an unusual type of intent handler that we do not currently support. The set of such unusual handlers is small [126], and once they are identified, adding support for them into Hogarth is straightforward.

5.5.5 Smart Studio Proxy

Smart Studio Proxy (also known as Android Trojan Spy [131]) is a piece of malware, discovered in 2015, designed to collect a variety of user data and ship it to the attacker. The malware accesses the user's SMS messages whenever a new message arrives, any message is changed by the user, the phone wakes from sleep mode, the network connectivity changes, or 30 minutes has passed since the last upload to the attacker (because there was no available network connection previously). These situations equate to four triggers, as waking and network change come through the same onReceive handler.

Hogarth was able to correctly identify one of these four triggers: the one for waking and network change. Within this trigger, Hogarth finds the path condition for network change, but not the path condition for waking. As with Contact Merger, the missing triggers and paths reflect the fact that Hogarth's Android system model is not yet complete.

5.6 Related Work

There are several threads of related work.

Contextual Security Analysis for Android. Several researchers have proposed program analyses that aim to infer the context, or provenance, of security-relevant actions in Android apps. Pegasus [115] analyzes apps to infer Permission Event Graphs (PEGs), which describe the relationship between Android events and permission uses. In contrast to Hogarth's provenance diagrams, PEGs do not include predicates about the app state.

AppIntent [110] uses data-flow analysis to identify program paths that may leak private information, and then employs directed symbolic execution on those paths to find inputs that could trigger a leak. As the full Android system is too complicated to effectively apply symbolic execution in a scalable manner, the authors use a system model to assist the analysis. Our approach is more precise in the sense that all behaviors we report are witnessed in actual executions, and by actually running the program, we eschew any need for a system model that would be hard to construct and maintain—as the system is updated frequently.

AppContext [132] identifies inter-procedure control-flow paths from program entry points to potentially malicious behaviors and then performs a data-flow analysis to identify the conditions that may trigger malicious behaviors. AppContext is similar to our approach in that it identifies the crucial branch conditions and inter-procedural contexts that lead to sensitive behaviors. As with AppIntent, our approach is more precise, being grounded with a dynamic analysis (although sound only with respect to observed behaviors).

TriggerScope [113] uses a combination of static analysis and symbolic execution to identify particularly complex trigger conditions associated with potentially malicious code. They attempt to detect “logic bombs” by identifying path conditions that are abnormally complex when simplified. Hogarth has three key differences with TriggerScope. First, our approach depends on gathering a representative corpus of dynamic traces, relying on the conceit that all permission uses will be exercised; TriggerScope specifically looks for triggers that are rare and specific. Second, because we rely on dynamic traces to drive further analysis of the application, we achieve a more precise result because all events observed in our dynamic traces are possible. Third, because we rely on a minimal model of the system, our approach is more resilient to the changes in Android from version to version.

FuzzDroid [133] uses a genetic mutation fuzzer to drive execution of an app toward a specific target location. IntelliDroid [116] uses static analysis to identify an overapproximation of inputs that could trigger malicious activity and then dynamically executes these inputs to prune false positives. These approaches identify a single path that reaches a target, in contrast to Hogarth which tries to identify the set of conditions that could lead to sensitive resource use.

Lastly, AppTracer [23] uses dynamic analysis to discover what user interactions temporally precede sensitive resource uses. Hogarth builds on AppTracer’s tracing infrastructure, which also uses Redexer, but adds inter-callback graph generation, to discover dependencies among callbacks, and symbolic execution/abstract interpretation, to recover path conditions. As a result, Hogarth can infer much richer contextual information about sensitive resource uses than AppTracer.

Taint and Flow Analysis for Android. TaintDroid [109] modifies the Android firmware to perform system-wide dynamic taint-tracking and notifies the user whenever sensitive data is leaked. Phosphor [111] provides similar taint-tracking, but instead modifies the JVM to improve portability. FlowDroid [108] uses static data-flow analysis find sensitive data leaks. These tools all focus on data flow, which is orthogonal to the control-flow (e.g., path conditions) dependencies that Hogarth discovers.

Other Analyses for Android. Yang et al. [114] present a model for tracking callback sequences in Android called *Callback Control-Flow Graphs* (CCFG). By connecting callbacks to their sources through the framework, CCFGs allow a static analysis to traverse context-sensitive control flow paths and identify callbacks that could be triggered. Because it is possible for different callbacks to be triggered based on the invocation context of the handler, e.g. onClick may trigger different callbacks depending on the widget it is associated with. This context-sensitivity improves the precision of static analysis in Android.

Building on the concept of CCFGs, EdgeMiner [126] performs a static analysis that automatically creates API summaries describing the relationship between callbacks and registrations through the framework via static data-flow analysis. As mentioned earlier, we use EdgeMiner's list of API methods that could register callbacks, but refine it by actual observations of data flows.

User-centric dependence analysis [134] uses a dependence analysis of Android apps to characterize the data consumption behaviors along paths from user inputs to sensitive resource uses. This project is a preliminary effort at identifying which user inputs a sensitive API call depends upon.

Concolic Execution. One style of symbolic execution is *concolic execution* [124, 135, 136], in which programs are instrumented to track symbolic expressions at runtime along with their concrete counterparts in the actual run. A benefit of concolic execution is that when a system call is made, any symbolic expressions passed in can be *concretized*, i.e., made equal to, their underlying values. This lets concolic executors avoid needing to model system calls, though at the expense of less power (e.g., not all possible paths that system call could take will be modeled). In a sense, Hogarth's approach is dual to concolic execution: We start with a concrete trace, and we then turn system call returns into *abstract* values so we can track their effect on program execution.

Text-based Contextual Security. Several researchers have explored natural-language contextual security for Android. For example, BACKSTAGE [137] mines Android apps for pairs of UI elements and the API calls they trigger and performs a clustering analysis to find outliers that invoke APIs atypical for their UI elements and textual descriptions. AsDroid [138] performs static analysis of top-most app methods and textual analysis of UI components they are associated with, to detect semantic mismatches. Slavin, et al. [139] produce a map of API methods to privacy policy phrases in order to check that natural-language privacy policies match the API uses in Android apps. TAPVerifier [140] builds a data-flow model of the target app and a natural-language model of its privacy policy to detect a mismatch. Wijesekera, et al. [106] and Olejnik, et al. [107] consider context outside of the app such as whether the app was in the foreground or background, whether the user was home or in public.

Our approach is orthogonal to these in that we use inter-callback control flow and symbolic constraints on data as our notion of context, and rely on a human auditor to interpret reasonable versus malicious triggers based on provenance diagrams.

5.7 Conclusion

In this paper, we introduced a new approach that uses dynamic analysis, symbolic execution, and abstract interpretation to automatically recover the provenance of permission uses in Android apps. We described Hogarth, a prototype implementation of our approach, and applied it to a set of five Android apps, demonstrating that it produces results comparable to a skilled reverse engineer.

In future work, we plan to modify Hogarth to build provenance diagrams incrementally on demand, to improve performance. We also plan to explore how to most usefully visualize provenance diagrams to aid skilled auditors in their analysis. We envision productive interactions between a live visualization of provenance and an auditor that allows probing down into the source code in a fine-grained way.

We believe that Hogarth provides a promising proof of concept, and we think the approach has the potential to be of significant aid in app auditing (at the market level), malware analysis, and reverse engineering.

$$\begin{array}{rcll}
\mathit{state} & ::= & \langle \vec{i}, \mathit{rf}, \sigma, \kappa, \phi, \vec{ti} \rangle & \text{[machine state]} \\
\mathit{rf} & : & \mathit{regs} \rightharpoonup a & \text{[register file]} \\
\sigma & : & a \rightarrow \mathit{fields} \rightharpoonup a & \text{[store]} \\
\kappa & ::= & \epsilon \mid (r, \mathit{rf}, \vec{i}) : \kappa & \text{[stack]} \\
\phi & ::= & a & \text{[path condition]} \\
a & ::= & c \mid C@\mathit{id} \mid \oplus\vec{a} & \text{[abstract value]} \\
 & \mid & r \mid a.f \mid C.m(\vec{a}) &
\end{array}$$

(a) Abstract machine domains.

$\langle \texttt{goto}\ j : \vec{i}, \mathit{rf}, \sigma, \kappa, \phi, \vec{ti} \rangle \rightsquigarrow \langle \mathit{instr}(j), \mathit{rf}, \sigma, \kappa, \phi, \vec{ti} \rangle$	[Jump]
$\langle \texttt{if}\ r\ \texttt{then}\ j : \vec{i}, \mathit{rf}, \sigma, \kappa, \phi, \texttt{then} : \vec{ti} \rangle \rightsquigarrow$ $\langle \mathit{instr}(j), \mathit{rf}, \sigma, \kappa, \mathit{rf}(r) \wedge \phi, \vec{ti} \rangle$	[Then]
$\langle \texttt{if}\ r\ \texttt{then}\ j : \vec{i}, \mathit{rf}, \sigma, \kappa, \phi, \texttt{else} : \vec{ti} \rangle \rightsquigarrow$ $\langle \vec{i}, \mathit{rf}, \sigma, \kappa, \neg\mathit{rf}(r) \wedge \phi, \vec{ti} \rangle$	[Else]
$\langle r \leftarrow c : \vec{i}, \mathit{rf}, \sigma, \kappa, \phi, \vec{ti} \rangle \rightsquigarrow \langle \vec{i}, \mathit{rf}[r \mapsto c], \sigma, \kappa, \phi, \vec{ti} \rangle$	[AssnC]
$\langle r \leftarrow r' : \vec{i}, \mathit{rf}, \sigma, \kappa, \phi, \vec{ti} \rangle \rightsquigarrow \langle \vec{i}, \mathit{rf}[r \mapsto \mathit{rf}(r')], \sigma, \kappa, \phi, \vec{ti} \rangle$	[AssnR]
$\langle r_1 \leftarrow r_2 \oplus r_3 : \vec{i}, \mathit{rf}, \sigma, \kappa, \phi, \vec{ti} \rangle \rightsquigarrow$ $\langle \vec{i}, \mathit{rf}[r_1 \mapsto \oplus \mathit{rf}(r_2), \mathit{rf}(r_3)], \sigma, \kappa, \phi, \vec{ti} \rangle$	[AssnOp]
$\langle r' \leftarrow r.f : \vec{i}, \mathit{rf}, \sigma, \kappa, \phi, \vec{ti} \rangle \rightsquigarrow \langle \vec{i}, \mathit{rf}', \sigma, \kappa, \phi, \vec{ti} \rangle$ where $\mathit{rf}' = \mathit{rf}[r' \mapsto \sigma(a)(f)] \wedge a = \mathit{rf}(r)$	[AssnF]
$\langle r' \leftarrow r.f : \vec{i}, \mathit{rf}, \sigma, \kappa, \phi, \vec{ti} \rangle \rightsquigarrow \langle \vec{i}, \mathit{rf}', \sigma, \kappa, \phi, \vec{ti} \rangle$ where $\mathit{rf}' = \mathit{rf}[r' \mapsto a.f] \wedge a = \mathit{rf}(r) \wedge f \notin \mathit{dom}(\sigma(a))$	[AssnFExt]
$\langle r.f \leftarrow r' : \vec{i}, \mathit{rf}, \sigma, \kappa, \phi, \vec{ti} \rangle \rightsquigarrow \langle \vec{i}, \mathit{rf}, \sigma', \kappa, \phi, \vec{ti} \rangle$ where $\sigma' = \sigma[\mathit{rf}(r) \mapsto f \mapsto \mathit{rf}(r')]$	[FWrite]
$\langle r \leftarrow \texttt{new}\ C : \vec{i}, \mathit{rf}, \sigma, \kappa, \phi, \vec{ti} \rangle \rightsquigarrow$ $\langle \vec{i}, \mathit{rf}[r \mapsto C@\mathit{id}], \sigma, \kappa, \phi, \vec{ti} \rangle$ where id fresh	[New]
$\langle r'' \leftarrow r.m(\vec{r}) : \vec{i}, \mathit{rf}, \sigma, \kappa, \phi, C.m : \vec{ti}' \rangle \rightsquigarrow$ $\langle \vec{i}', \mathit{rf}', \sigma, \kappa', \phi, \vec{ti}' \rangle$ where $\mathit{rf}' = [\vec{r'} \mapsto \mathit{rf}(\vec{r})] \wedge m(\vec{r'})\ \vec{i}' = \mathit{lookup}(C.m)$ $\wedge\ \kappa' = (r'', \mathit{rf}, \vec{i}) : \kappa$	[Call]
$\langle \texttt{ret}\ r : \vec{i}, \mathit{rf}, \sigma, (r', \mathit{rf}', \vec{i}') : \kappa, \phi, \vec{ti} \rangle \rightsquigarrow$ $\langle \vec{i}', \mathit{rf}'[r' \mapsto \mathit{rf}(r)], \sigma, \kappa, \phi, \vec{ti} \rangle$	[Ret]
$\langle r' \leftarrow r.m(\vec{r}) : \vec{i}, \mathit{rf}, \sigma, \kappa, \phi, C.m(\vec{v}) : \vec{ti} \rangle \rightsquigarrow$ $\langle \vec{i}, \mathit{rf}[r' \mapsto C.m(\mathit{rf}(\vec{r}))], \sigma, \kappa, \phi, \vec{ti} \rangle$	[API]

(b) Abstract machine semantics.

Figure 5.7: Formalism for sparse trace interpolation.

Chapter 6: Conclusion and Future Directions

In this dissertation, I defined interaction-based security as security that is informed by the user's interaction with an app's UI. I then presented several pieces of work showing how we could move the space of mobile apps closer towards accomodating interaction-based security. In this chapter, I will briefly review each piece of work and discuss how it fits in to the larger space of mobile apps and systems going forward, and detail some future directions of work.

6.1 Binary Rewriting for Android Apps

Much of the work in my dissertation relied on the ability to perform binary instrumentation of Android apps. Binary instrumentation is a powerful mechanism that allows us to perform a range of dynamic analyses, only some of which have been presented here. Chapter 2 detailed some of the key issues involved in transforming Android apps correctly. I described Redexer—my binary rewriter for Android—and its implementation, along with a demonstration of how Redexer can be used to study location truncation in Android apps.

Redexer currently works at scale on arbitrarily large Android apps. In my work for Chapter 4, we used Redexer on most top apps from Google Play. The logging framework was pushed even further during my work on Chapter 5, wherein we logged every method invocation in several production-sized apps. Generally, Redexer offers a fairly flexible API for app rewriting and could be used for applications not described here. For example, we are currently working on using Redexer to understand which data apps back up. I also envision using Redexer to enforce the security policies of the sort we discovered might be natural to users in Chapter 5. For example, this might happen by using Hogarth to figure out the policy for an app and then using a dynamic enforcement mechanism in Redexer to exit the app if the app goes down a path not observed in the set of analyzed runs.

6.2 Checking Interaction-Based Declassification Policies for Android Using Symbolic Execution

While Redexer allows rewriting apps to change their dynamic behavior, Chapter 3 explored the complications of understanding information-flow in apps. Information flow is a tricky property, because it relies on quantifying over sets of runs, rather than over a single run.

I did not mention information flow in subsequent chapters, and instead focused on permission uses. I would like to go back and study user preferences and security expectations in the presence of information flow. As we saw in Chapter 3, information-flow is still challenging to apply to production apps, compared to, e.g., permission systems. I plan to study what has to be done to support information flow for production apps and if it would even be beneficial to users (versus better permission systems).

6.3 User Interactions and Permission Use on Android

Chapter 4 detailed my work in understanding the relation between user interactions and permission in Android apps. I found that the Android system is largely heading in the right direction by encouraging apps to request permissions on first use. However, we found that users may not fully understand background permission uses, specifically when they occur in apps which also have foreground uses of those permissions.

I plan to continue this work trying to understand how we could better design UIs to help users understand background permission uses. This was challenging in AppTracer, because AppTracer was driven by a temporal notion of influence: it was hard to reason about why a permission use happened when its cause was far removed in time from when the use occurred. This motivated me to build Hogarth, which allows understanding this dependence in a principled way by reasoning about an interhandler execution graph. However, I have not applied Hogarth to large apps of the variety in Chapter 4. I plan to continue work on tools like Hogarth, so that I can apply them to production apps and understand why apps use background resources. I hope this will help inform knowledge of why these uses are necessary and what we can do to explain them to users.

6.4 Permission-Use Provenance in Android Using Sparse Dynamic Analysis

Finally, Chapter 5 introduced Hogarth, a system which uses app logs as the basis for program understanding. Understanding interaction-based security requires very precise knowledge about paths through an app. It is challenging to use traditional program analyses in this setting because those analyses achieve scalability by sacrificing precision. Hogarth provides a very precise understanding of why a permission is used in an app, but sacrifices soundness to do so.

There are many exciting directions to explore following Hogarth. First, the initial implementation of Hogarth needs to be scaled up to work on realistically sized apps. I have a number of ideas for how to do so. For example, one limiting factor in Chapter 5 was that Hogarth ran out of memory processing very large logs. This was because a symbolic state was created for each entry in the log. Instead, I will modify Hogarth so that it performs on-the-fly minimization, merging newly discovered symbolic states as it interpolates the log rather than generating all of the symbolic states before minimization.

Next, I would like to understand how to make fundamental improvements to Hogarth. For example, Hogarth will frequently include in its path conditions the post conditions for each loop through which it passes. Although this is the correct behavior, I have frequently found that this information is not usually useful. Also, Hogarth uses only positive examples as it performs its minimization. I speculate that Hogarth may be able to present a smaller path condition if it used negative paths (which did not lead to a permission use).

More generally, I believe Hogarth is just a first step towards using program analysis to aid reverse engineering. I am uncertain what real reverse engineers would think of Hogarth. I assume that we would discover many interesting problems if we were to give Hogarth to professional reverse engineers. I am very much interested in doing so, and using other techniques from program analysis to help inspire advances in program understanding.

6.5 High level results and conclusions

Throughout this dissertation I studied how user interaction can help reason and inform decisions about security in mobile apps. I conclude that user interaction informs the user's mental model of app behavior. I formalized interaction-based security in the form of interaction-based declassification, and I presented several tools which help check and enforce security interaction-based security policies in apps. User interaction has both programmatic and conceptual components, and I conclude that integrating the two can lead to increased security in mobile apps.

Bibliography

[1] Pew Research Center, January 2017. http://www.pewresearch.org/about/use-policy/ (Accessed 5-23-17).

[2] James Vincent. 99.6 percent of new smartphones run android or ios, Feb 2017.

[3] Serge Egelman, Lorrie Faith Cranor, and Jason Hong. You've been warned: An empirical study of the effectiveness of web browser phishing warnings. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '08, pages 1065–1074, New York, NY, USA, 2008. ACM.

[4] Helen Nissenbaum. Privacy as Contextual Integrity. *Washington Law Review*, 79:119–157, 2004.

[5] Jennifer King. "how come i'm allowing strangers to go through my phone?": Smartphones and privacy expectations. In *Proceedings of the Ninth Symposium on Usable Privacy and Security*, SOUPS '12, Washington, DC, USA, 2012. USENIX.

[6] Rebecca Balebako, Jaeyeon Jung, Wei Lu, Lorrie Faith Cranor, and Carolyn Nguyen. "little brothers watching you": Raising awareness of data leaks on smartphones. In *Proceedings of the Ninth Symposium on Usable Privacy and Security*, SOUPS '13, pages 12:1–12:11, New York, NY, USA, 2013. ACM.

[7] Huiqing Fu, Yulong Yang, Nileema Shingte, Janne Lindqvist, and Marco Gruteser. A field study of run-time location access disclosures on android smartphones. In *Workshop on Usable Security (USEC)*, San Diego, California, USA, 2014. Internet Society.

[8] Primal Wijesekera, Arjun Baokar, Ashkan Hosseini, Serge Egelman, David Wagner, and Konstantin Beznosov. Android permissions remystified: A field study on contextual integrity. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security '15)*, pages 499–514, Washington, D.C., August 2015. USENIX Association.

[9] Jinseong Jeon, Kristopher K. Micinski, Jeffrey A. Vaughan, Ari Fogel, Nikhilesh Reddy, Jeffrey S. Foster, and Todd Millstein. Dr. Android and Mr.

Hide: Fine-grained Permissions in Android Applications. In *ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, pages 3–14, 2012.

[10] Kristopher Micinski, Philip Phelps, and Jeffrey S. Foster. An Empirical Study of Location Truncation on Android. In *Mobile Security Technologies (MoST)*, San Francisco, CA, May 2013.

[11] Dorothy Elizabeth Robling Denning. *Secure Information Flow in Computer Systems.* PhD thesis, West Lafayette, IN, USA, 1975. AAI7600514.

[12] Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. *J. Comput. Secur.*, 17(5):517–548, October 2009.

[13] Aslan Askarov and Andrei Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, SP '07, pages 207–221, Washington, DC, USA, 2007. IEEE Computer Society.

[14] Stephen Chong and Andrew C. Myers. Security policies for downgrading. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, pages 189–209, October 2004.

[15] David Walker. A type system for expressive security policies. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '00, pages 254–267, New York, NY, USA, 2000. ACM.

[16] Peng Li and Steve Zdancewic. Downgrading policies and relaxed noninterference. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 158–170, New York, NY, USA, 2005. ACM.

[17] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *Computer Security Foundations, 2005. CSFW-18 2005. 18th IEEE Workshop*, pages 255–269, June 2005.

[18] Kevin R. O'Neill, Michael R. Clarkson, and Stephen Chong. Information-flow security for interactive programs. In *Proceedings of the 19th IEEE Workshop on Computer Security Foundations*, CSFW '06, pages 190–201, Washington, DC, USA, 2006. IEEE Computer Society.

[19] Rayna Dimitrova, Bernd Finkbeiner, Máté Kovács, Markus N. Rabe, and Helmut Seidl. Model checking information flow in reactive systems. In *Proceedings of the 13th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'12, pages 169–185, Berlin, Heidelberg, 2012. Springer-Verlag.

[20] Kristopher Micinski, Jonathan Fetter-Degges, Jinseong Jeon, Jeffrey S. Foster, and Michael R. Clarkson. Checking Interaction-Based Declassification Policies for Android Using Symbolic Execution. In *European Symposium on Research in Computer Security (ESORICS)*, volume 9327 of *Lecture Notes in Computer Science*, pages 520–538, Vienna, Austria, September 2015.

[21] Jinseong Jeon, Kristopher K. Micinski, and Jeffrey S. Foster. SymDroid: Symbolic Execution for Dalvik Bytecode. Technical Report CS-TR-5022, Department of Computer Science, University of Maryland, College Park, July 2012.

[22] Kristopher Micinski, Daniel Votipka, Rock Stevens, Nikolaos Kofinas, Michelle L Mazurek, and Jeffrey S Foster. User interactions and permission use on android. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, pages 362–373. ACM, 2017.

[23] Kristopher Micinski, Daniel Votipka, Rock Stevens, Nikolaos Kofinas, Michelle L Mazurek, and Jeffrey S Foster. User interactions and permission use on android. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, pages 362–373. ACM, 2017.

[24] F-Droid Limited. F-droid - free and open source android repository, 2017. (Accessed 4-11-2017).

[25] Mila Parkour. Contagio mobile, 2017. (Accessed 4-11-2017).

[26] Connor Tumbleson and Ryszard Wiśniewski. *Apktool - A tool for reverse engineering Android apk files*. Google, 2016.

[27] The Android Open Source Project. Bytecode definition file. https://android.googlesource.com/platform/dalvik/+/kitkat-release/opcode-gen/bytecode.txt (Accessed 5-24-17).

[28] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. Technical report, Rochester, NY, USA, 1995.

[29] Alastair R. Beresford and Frank Stajano. Mix zones: User privacy in location-aware services. In *IEEE Conference on Pervasive Computing and Communications Workshops (PERCOMW)*, pages 127–, 2004.

[30] Reza Shokri, George Theodorakopoulos, Jean-Yves Le Boudec, and Jean-Pierre Hubaux. Quantifying location privacy. In *IEEE Symposium on Security and Privacy (SP)*, pages 247–262, 2011.

[31] Reza Shokri, George Theodorakopoulos, Carmela Troncoso, Jean-Pierre Hubaux, and Jean-Yves Le Boudec. Protecting location privacy: optimal strategy against localization attacks. In *2012 ACM conference on Computer and Communications Security (CCS)*, pages 617–627, 2012.

[32] Claudio Bettini, X. Sean Wang, and Sushil Jajodia. Protecting privacy against location-based personal identification. In *VDLB International Conference on Secure Data Management (SDM)*, pages 185–199, 2005.

[33] Baik Hoh and M. Gruteser. Protecting location privacy through path confusion. In *Security and Privacy for Emerging Areas in Communications Networks*, pages 194–205, 2005.

[34] Marco Gruteser and Dirk Grunwald. Anonymous usage of location-based services through spatial and temporal cloaking. In *International Conference on Mobile Systems, Applications and Services (MobiSys)*, pages 31–42, 2003.

[35] A.J. Bernheim Brush, John Krumm, and James Scott. Exploring end user preferences for location obfuscation, location-based services, and the value of location. In *ACM International Conference on Ubiquitous Computing (Ubicomp)*, pages 95–104, 2010.

[36] John Krumm. A survey of computational location privacy. *Personal Ubiquitous Comput.*, 13(6):391–399, August 2009.

[37] Marco Gruteser and Baik Hoh. On the anonymity of periodic location samples. In *International Conference on Security in Pervasive Computing (SPC)*, pages 179–192, 2005.

[38] John Krumm. Inference attacks on location tracks. In *International Conference on Pervasive Computing (PERVASIVE)*, pages 127–143, 2007.

[39] National Geospatial-Intelligence Agency. Length of degree calculator, February 2013.

[40] Richard H Rapp. Geometric geodesy: Part i. *Lecture Notes*, 1991.

[41] VI Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707, 1966.

[42] Jinseong Jeon and Jeffrey S. Foster. Troyd: Integration Testing for Android. Technical Report CS-TR-5013, Department of Computer Science, University of Maryland, College Park, August 2012.

[43] Alastair R. Beresford, Andrew Rice, Nicholas Skehin, and Ripduman Sohan. MockDroid: trading privacy for application functionality on smartphones. In *HotMobile*, 2011.

[44] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *ASIACCS*, pages 328–332, 2010.

[45] Y. Zhou, X. Zhang, X. Jiang, and V. Freeh. Taming information-stealing smartphone applications (on android). *Trust and Trustworthy Computing*, pages 93–107, 2011.

[46] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *ACM Conference on Computer and Communications Security (CCS)*, pages 639–652, 2011.

[47] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *CCS*, pages 235–245, 2009.

[48] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically rich application-centric security in android. In *ACSAC*, pages 340–349, 2009.

[49] D. Barrera, H.G. Kayacik, P.C. van Oorschot, and A. Somayaji. A methodology for empirical analysis of permission-based security models and its application to android. In *CCS*, pages 73–84, 2010.

[50] Timothy Vidas, Nicolas Christin, and Lorrie Faith Cranor. Curbing Android Permission Creep. In *W2SP*, 2011.

[51] 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.

[52] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N.t Sheth. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, 2010.

[53] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A Study of Android Application Security. In *USENIX Security*, 2011.

[54] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing Inter-Application Communication in Android. In *MobiSys*, 2011.

[55] A.P. Felt, H. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission redelegation: Attacks and defenses. In *USENIX Security*, 2011.

[56] Michael Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. Systematic Detection of Capability Leaks in Stock Android Smartphones. In *NDSS*, 2012. To appear.

[57] Shahriyar Amini, Janne Lindqvist, Jason I. Hong, Maladau Mou, Rahul Raheja, Jialiu Lin, Norman Sadeh, and Eran Tochb. Caché caching location-enhanced content to improve user privacy. *SIGMOBILE Mob. Comput. Commun. Rev.*, 14(3):19–21, December 2010.

[58] Philippe Golle and Kurt Partridge. On the anonymity of home/work location pairs. In *International Conference on Pervasive Computing (PERVASIVE)*, pages 390–397, 2009.

[59] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[60] S. Zdancewic and A.C. Myers. Observational determinism for concurrent program security. In *Computer Security Foundations Workshop, 2003. Proceedings. 16th IEEE*, pages 29–43, 2003.

[61] Leonardo de Moura and Nikolaj BjÃÿrner. Z3: An efficient SMT solver. In C.R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin Heidelberg, 2008.

[62] Google. Managing the Activity Lifecycle, 2015.

[63] Kristopher Micinski, Jonathan Fetter-Degges, Jinseong Jeon, Jeffrey S. Foster, and Michael R. Clarkson. *Checking Interaction-Based Declassification Policies for Android Using Symbolic Execution*, pages 520–538. Springer International Publishing, Vienna, Austria, 2015.

[64] Orna Lichtenstein, Amir Pnueli, and Lenore Zuck. The glory of the past. In Rohit Parikh, editor, *Logics of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 196–218. Springer Berlin Heidelberg, 1985.

[65] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *J. Comput. Secur.*, 18(6):1157–1210, September 2010.

[66] Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and CÃľsar SÃąnchez. Temporal logics for hyperproperties. In MartÃŋn Abadi and Steve Kremer, editors, *Principles of Security and Trust*, volume 8414 of *Lecture Notes in Computer Science*, pages 265–284. Springer Berlin Heidelberg, 2014.

[67] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *Commun. ACM*, 56(2):82–90, February 2013.

[68] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks. Directed symbolic execution. In *Proceedings of the 18th International Conference on Static Analysis*, SAS'11, pages 95–111, Berlin, Heidelberg, 2011. Springer-Verlag.

[69] Franziska Roesner, Tadayoshi Kohno, Alexander Moshchuk, Bryan Parno, Helen J. Wang, and Crispin Cowan. User-driven access control: Rethinking permission granting in modern operating systems. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 224–238, Washington, DC, USA, 2012. IEEE Computer Society.

[70] Ka-Ping Yee. Aligning security and usability. *Security Privacy, IEEE*, 2(5):48–55, Sept 2004.

[71] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[72] Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and X. Sean Wang. Appintent: analyzing sensitive data transmission in Android for privacy leakage detection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer &#38; communications security*, CCS '13, pages 1043–1054, New York, NY, USA, 2013. ACM.

[73] Kevin Zhijie Chen, Noah M. Johnson, Vijay D'Silva, Shuaifu Dai, Kyle MacNamara, Tom Magrino, Edward XueJun Wu, Martin Rinard, and Dawn Xiaodong Song. Contextual policy enforcement in Android applications with permission event graphs. In *NDSS*. The Internet Society, 2013.

[74] Limin Jia, Jassim Aljuraidan, Elli Fragkaki, Lujo Bauer, Michael Stroucken, Kazuhide Fukushima, Shinsaku Kiyomoto, and Yutaka Miyake. Run-time enforcement of information-flow properties on Android. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *Computer Security âĂŞ ESORICS 2013*, volume 8134 of *Lecture Notes in Computer Science*, pages 775–792. Springer Berlin Heidelberg, 2013.

[75] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard OS abstractions. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 321–334, New York, NY, USA, 2007. ACM.

[76] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4(2-3):167–187, January 1996.

[77] David Clark and Sebastian Hunt. Non-interference for deterministic interactive programs. In Pierpaolo Degano, Joshua Guttman, and Fabio Martinelli, editors, *Proc. Formal Aspects in Security and Trust*, pages 50–66. Springer-Verlag, Berlin, Heidelberg, 2009.

[78] Stephen Chong, K. Vikram, and Andrew C. Myers. SIF: Enforcing confidentiality and integrity in web applications. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, SS'07, pages 1:1–1:16, Berkeley, CA, USA, 2007. USENIX Association.

[79] Andrew C. Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 228–241, New York, NY, USA, 1999. ACM.

[80] Jeffrey A. Vaughan and Stephen Chong. Inference of expressive declassification policies. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 180–195, Washington, DC, USA, 2011. IEEE Computer Society.

[81] Willard Rafnsson, Daniel Hedin, and Andrei Sabelfeld. Securing interactive programs. In *Proceedings of the 2012 IEEE 25th Computer Security Foundations Symposium*, CSF '12, pages 293–307, Washington, DC, USA, 2012. IEEE Computer Society.

[82] Franziska Roesner, Tadayoshi Kohno, Alexander Moshchuk, Bryan Parno, Helen J. Wang, and Crispin Cowan. User-driven access control: Rethinking permission granting in modern operating systems. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 224–238, Washington, DC, USA, 2012. IEEE Computer Society.

[83] 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, New York, NY, USA, 2012. ACM.

[84] Google. *Requesting Permissions at Run Time*, 2016.

[85] Sven Bugiel, Stephen Heuser, and Ahmad-Reza Sadeghi. Flexible and fine-grained mandatory access control on android for diverse security and privacy policies. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Security '13)*, pages 131–146, Washington, D.C., 2013. USENIX.

[86] Adrienne Porter Felt, Serge Egelman, and David Wagner. I've got 99 problems, but vibration ain't one: A survey of smartphone users' concerns. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '12, pages 33–44, New York, NY, USA, 2012. ACM.

[87] Hazim Almuhimedi, Florian Schaub, Norman Sadeh, Idris Adjerid, Alessandro Acquisti, Joshua Gluck, Lorrie Faith Cranor, and Yuvraj Agarwal. Your location has been shared 5,398 times!: A field study on mobile app privacy nudging. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, CHI '15, pages 787–796, New York, NY, USA, 2015. ACM.

[88] How to ask for permission. In *Presented as part of the 7th USENIX Workshop on Hot Topics in Security*, Berkeley, CA, 2012. USENIX.

[89] Marc Stiegler and Mark S. Miller. A capability-based client: The darpabrowser. Technical Report Technical Report, Focused Research Topic 5, Combex Inc, Meadowbrook, PA, June 2002.

[90] Marc Stiegler, Alan H. Karp, Ka-Ping Yee, Tyler Close, and Mark S. Miller. Polaris: Virus-safe computing for windows xp. *Commun. ACM*, 49(9):83–88, September 2006.

[91] Franziska Roesner and Tadayoshi Kohno. Securing embedded user interfaces: Android and beyond. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Security '13)*, pages 97–112, Washington, D.C., 2013. USENIX.

[92] Talia Ringer, Dan Grossman, and Franziska Roesner. Audacious: User-driven access control with unmodified operating systems. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security*, Vienna, Austria, oct 2016. ACM.

[93] 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, New York, NY, USA, 2012. ACM.

[94] Android Developers Reference. Manifest.permission_group, 2016. (Accessed 9-16-2016).

[95] Andrew F Hayes and Klaus Krippendorff. Answering the call for a standard reliability measure for coding data. *Communication methods and measures*, 1(1):77–89, 2007.

[96] Joshua Sunshine, Serge Egelman, Hazim Almuhimedi, Neha Atri, and Lorrie Faith Cranor. Crying Wolf - An Empirical Study of SSL Warning Effectiveness. In *Proceedings of the 18th USENIX Security Symposium*, 2009.

[97] Stuart E Schechter, Rachna Dhamija, Andy Ozment, and Ian Fischer. The Emperor's New Security Indicators. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy.*

[98] Donald Hedeker. Multilevel models for ordinal and nominal variables. In *Handbook of multilevel analysis*, pages 237–274. Springer, 2008.

[99] Hirotugu Akaike. A new look at the statistical model identification. *Automatic Control, IEEE Transactions on*, 19(6):716–723, December 1974.

[100] Ashwini Rao, Florian Schaub, Norman Sadeh, Alessandro Acquisti, and Ruogu Kang. Expecting the unexpected: Understanding mismatched privacy expectations online. In *Twelfth Symposium on Usable Privacy and Security (SOUPS 2016)*, pages 77–96, Denver, CO, June 2016. USENIX Association.

[101] Michael Buhrmester, Tracy Kwang, and Samuel D. Gosling. Amazon's mechanical turk: A new source of inexpensive, yet high-quality, data? *Perspectives on Psychological Science*, 6(1):3–5, 2011.

[102] Aniket Kittur, Ed H. Chi, and Bongwon Suh. Crowdsourcing user studies with mechanical turk. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '08, pages 453–456, New York, NY, USA, 2008. ACM.

[103] Julie S. Downs, Mandy B. Holbrook, Steve Sheng, and Lorrie Faith Cranor. Are your participants gaming the system?: Screening mechanical turk workers. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 2399–2402, New York, NY, USA, 2010. ACM.

[104] Michael Toomim, Travis Kriplean, Claus Pörtner, and James Landay. Utility of human-computer interactions: Toward a science of preference measurement. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, pages 2275–2284, New York, NY, USA, 2011. ACM.

[105] Ruogu Kang, Stephanie Brown, Laura Dabbish, and Sara Kiesler. Privacy attitudes of mechanical turk workers and the u.s. public. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security '14)*, pages 37–49, San Diego, California, USA, 2014. USENIX Association.

[106] Primal Wijesekera, Arjun Baokar, Lynn Tsai, Joel Reardon, Serge Egelman, David Wagner, and Konstantin Beznosov. The feasibility of dynamically granted permissions: Aligning mobile privacy with user preferences. *CoRR*, abs/1703.02090, 2017.

[107] Katarzyna Olejnik, Italo Dacosta, Joana Soares Machado, Kévin Huguenin, Mohammad Emtiyaz Khan, and Jean-Pierre Hubaux. SmarPer: Context-Aware and Automatic Runtime-Permissions for Mobile Devices. In *38th IEEE Symposium on Security and Privacy (S&P)*, San Jose, CA, United States, May 2017. IEEE.

[108] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 259–269, New York, NY, USA, 2014. ACM.

[109] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 393–407, Berkeley, CA, USA, 2010. USENIX Association.

[110] Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and X. Sean Wang. Appintent: analyzing sensitive data transmission in android for privacy leakage detection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer &#38; communications security*, CCS '13, pages 1043–1054, New York, NY, USA, 2013. ACM.

[111] Jonathan Bell and Gail Kaiser. Phosphor: Illuminating dynamic data flow in commodity jvms. In *Proceedings of the 2014 ACM International Conference on*

Object Oriented Programming Systems Languages & Applications, OOPSLA '14, pages 83–101, New York, NY, USA, 2014. ACM.

[112] Kangjie Lu, Zhichun Li, Vasileios P. Kemerlis, Zhenyu Wu, Long Lu, Cong Zheng, Zhiyun Qian, Wenke Lee, and Guofei Jiang. Checking more and alerting less: Detecting privacy leakages via enhanced data-flow analysis and peer voting. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*, 2015.

[113] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna. Triggerscope: Towards detecting logic bombs in android applications. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 377–396, May 2016.

[114] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. Static control-flow analysis of user-driven callbacks in android applications. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 89–99, Piscataway, NJ, USA, 2015. IEEE Press.

[115] Kevin Zhijie Chen, Noah M Johnson, Vijay D'Silva, Shuaifu Dai, Kyle MacNamara, Thomas R Magrino, Edward XueJun Wu, Martin Rinard, and Dawn Xiaodong Song. Contextual policy enforcement in android applications with permission event graphs. In *NDSS*, page 234, 2013.

[116] Michelle Y Wong and David Lie. Intellidroid: A targeted input generator for the dynamic analysis of android malware. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*, 2016.

[117] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 576–587, New York, NY, USA, 2014. ACM.

[118] Jinseong Jeon, Kristopher K Micinski, and Jeffrey S Foster. Symbolic execution for dalvik bytecode, 2012. (Tech Report, CS-TR-5022).

[119] Marc Rogers. Dendroid malware can take over your camera, record audio, and sneak into google play, 2014. (Accessed 4-11-2017).

[120] Google, inc. Camera2basic android sample app, 2017.

[121] Jeb decompiler, 2017. (Accessed 5-19-2017).

[122] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, CA, 1977. ACM Press, New York.

[123] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, TX, 1979. ACM Press, New York.

[124] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*. ACM, 2006.

[125] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, 2008.

[126] Yinzhi Cao, Yanick Fratantonio, Antonio Bianchi, Manuel Egele, Christopher Kruegel, Giovanni Vigna, and Yan Chen. Edgeminer: Automatically detecting implicit control flow transitions through the android framework. In *NDSS*, 2015.

[127] F-Droid Limited. Call recorder for android, 2014. (Accessed 4-11-2017).

[128] Google Inc. and the Open Handset Alliance. Android documentation: Broadcasts, 2017. (Accessed 5-17-2017).

[129] Amir Yalon. Misbothering sms receiver, 2015. (Accessed 4-11-2017).

[130] Rene Treffer. Contact merger, 2014. (Accessed 4-11-2017).

[131] Lukas Stefanko. Android trojan spy goes 2 year undetected, 2015. (Accessed 4-11-2017).

[132] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck. Appcontext: Differentiating malicious and benign mobile app behaviors using context. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 303–313, May 2015.

[133] Siegfried Rasthofer, Steven Arzt, Stefan Triller, and Michael Pradel. Making malory behave maliciously: Targeted fuzzing of android execution environments. In *Proceedings of the 39th International Conference on Software Engineering. ACM. To appear*, 2017.

[134] Karim O Elish, Danfeng Yao, and Barbara G Ryder. User-centric dependence analysis for identifying malicious mobile apps. In *Workshop on Mobile Security Technologies*, 2012.

[135] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[136] Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.

[137] Vitalii Avdiienko, Konstantin Kuznetsov, Isabelle Rommelfanger, Andreas Rau, Alessandra Gorla, and Andreas Zeller. Detecting behavior anomalies in graphical user interfaces.

[138] 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*, ICSE 2014, pages 1036–1046, New York, NY, USA, 2014. ACM.

[139] Rocky Slavin, Xiaoyin Wang, Mitra Bokaei Hosseini, James Hester, Ram Krishnan, Jaspreet Bhatia, Travis D Breaux, and Jianwei Niu. Toward a framework for detecting privacy policy violations in android application code. In *Proceedings of the 38th International Conference on Software Engineering*, pages 25–36. ACM, 2016.

[140] Le Yu, Xiapu Luo, Chenxiong Qian, and Shuai Wang. Revisiting the description-to-behavior fidelity in android applications. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, volume 1, pages 415–426. IEEE, 2016.