PUMA: Programmable UI-Automation for Large Scale Dynamic Analysis of Mobile Apps

Shuai Hao[*], Bin Liu[*], Suman Nath[†], William G.J. Halfond[*], Ramesh Govindan[*]
[*]University of Southern California [†]Microsoft Research

ABSTRACT

Mobile app ecosystems have experienced tremendous growth in the last six years. This has triggered research on dynamic analysis of performance, security, and correctness properties of the mobile apps in the ecosystem. Exploration of app execution using automated UI actions has emerged as an important tool for this research. However, existing research has largely developed analysis-specific UI automation techniques, wherein the logic for exploring app execution is intertwined with the logic for analyzing app properties. PUMA is a programmable framework that separates these two concerns. It contains a generic UI automation capability (often called a Monkey) that exposes high-level events for which users can define handlers. These handlers can flexibly direct the Monkey's exploration, and also specify app instrumentation for collecting dynamic state information or for triggering changes in the environment during app execution. Targeted towards operators of app marketplaces, PUMA incorporates mechanisms for scaling dynamic analysis to thousands of apps. We demonstrate the capabilities of PUMA by analyzing seven distinct performance, security, and correctness properties for 3600 apps downloaded from the Google Play store.

1. INTRODUCTION

Today's smartphone app stores host large collections of apps. Most of the apps are created by unknown developers who have varying expertise and who may not always operate in the users' best interests. Such concerns have motivated researchers and app store operators to analyze various properties of the apps and to propose and evaluate new techniques to address the concerns. For such analyses to be useful, the analysis technique must be robust and scale well for large collections of apps.

Static analysis of app binaries, as used in prior work to identify privacy [21] and security [10, 12] problems, or app clones [9] *etc.*, can scale to a large number of apps. However, static analysis can fail to capture runtime contexts, such as data dynamically downloaded from the cloud, objects created during runtime, configuration variables, and so on. Moreover, app binaries may be obfuscated to thwart static analysis, either intentionally or unintentionally (such as stripping symbol information to reduce the size of the app binary).

Therefore, recent work has focused on dynamic analyses that execute apps and examine their runtime properties (Section 2). These analyses has been used for analyzing performance [26, 27, 15], bugs [22, 25, 19], privacy and security [11, 24], compliance [20] and correctness [18], of apps, some at a scale of thousands of apps. One popular way to scale dynamic analysis to a large number of apps is to use a software automation tool called a "monkey" that can automatically launch and interact with an app (by tapping on buttons, typing text inputs, *etc.*) in order to navigate to various execution states (or, pages) of the app. The monkey is augmented with code tailored to the target analysis; this code is systematically executed while the monkey visits various pages. For example, in DECAF [20], the analysis code algorithmically examines ads in the current page to check if their placement violates ad network policies.

Dynamic analysis of apps is a daunting task (Section 2). At a high level, it consists of *exploration logic* that guides the monkey to explore various app states and *analysis logic* that analyzes the targeted runtime properties of the current app state. The exploration logic needs to be optimized for *coverage*—it should explore a significant portion of the useful app states, and for *speed*—it should analyze a large collection of apps within a reasonable time. To achieve these goals, existing systems have developed a monkey from scratch and have tuned its exploration logic by leveraging properties of the analysis. For example, AMC [18] and DECAF [20] required analyzing one of each type of app page, and hence their monkey is tuned to explore only unique page types. On the other hand, SmartAds [23] crawled data from all pages, so its monkey is tuned to explore *all* unique pages. Similarly, the monkeys of VanarSena [25] and ConVirt [19] inject faults at specific execution points, while that of AMC and DECAF only read specific UI elements from app pages. Some systems even instrument app binaries to optimize the monkey [25] or to access app runtime state [23]. In summary, exploration logic and analysis logic are often intertwined and hence a system designed for one analysis cannot be readily used for another. The end effect is that many of the advances developed to handle large scale studies are only utilizable in the context of the specific analysis and cannot currently be generalized to other analyses.

Contributions. In this paper we propose PUMA (Section 3), a dynamic analysis framework that can be instantiated for a large number of diverse dynamic analysis tasks that, in prior research, used systems built from scratch. PUMA enables analysis of a wide variety of app properties, allows its users to flexibly specify which app states to explore and how, provides programmatic access to the app's runtime state for analysis, and supports dynamic runtime environment modification. It encapsulates the common components of existing dynamic analysis systems and exposes a number of configurable hooks that can be programmed with a high level event-driven scripting language, called PUMAscript. This language cleanly separates analysis logic from exploration logic, allowing its users to (a) succinctly specify navigation hints for scalable app exploration and (b) separately specify the logic for analyzing the app properties.

This design has two distinct advantages. First, it can simplify the analysis of different app properties, since users do not need to develop the monkey, which is often the most challenging part of dynamic analysis. A related benefit is that the monkey can evolve independently of the analysis logic, so that monkey scaling and coverage improvements can be made available to all users. Second, PUMA can *multiplex* dynamic analyses — it can concurrently run similar analyses, resulting in better scaling of the dynamic analysis.

To validate the design of PUMA, we present the results of seven distinct analyses (many of which are presented in prior work) executed on 3,600 apps from Google Play (Section 4). The PUMAscripts for these analyses are each less than 100 lines of code; by contrast, DECAF [20] required over 4,000 lines of which over 70% was dedicated to app exploration. Our analyses are valuable in their own right, since they present fascinating insights into the app ecosystem: there appear to be a relatively small number (about 40) of common UI design patterns among Android apps; enabling content search for apps in the app store can increase the relevance of results and yield up to 50 additional results per query on average; over half of the apps violate accessibility guidelines; network usage requirements for apps vary by six orders of magnitude; and a quarter of all apps fail basic stress tests.

PUMA can be used in various settings. An app store can use PUMA—the store's app certification team can use it to verify that a newly submitted app does not violate any privacy and security policies, the advertising team can check if the app does not commit any ad fraud, the app store search engine can crawl app data for indexing, etc. Researchers interested in analyzing the app ecosystem can download PUMA and the apps of interest, customize PUMA for their target analysis, and conduct the analysis locally. A third-party can offer PUMA as a service where users can submit their analyses written in PUMAscript for analyzing the app ecosystems.

2. BACKGROUND AND MOTIVATION

In this section, we describe the unique requirements of large-scale studies of mobile apps and motivate the need for a programmable UI-based framework for supporting these studies. We also discuss the challenges associated with satisfying these requirements. In Section 3, we describe how PUMA addresses these challenges and requirements.

2.1 Dynamic Analysis of Mobile Apps

Dynamic analysis of software is performed by executing the software, subjecting it to different inputs, and recording (and subsequently analyzing) its internal state and outputs. Mobile apps have a unique structure that enables a novel form of dynamic analysis. By design, most mobile app actions are triggered by user interactions, such as clicks, swipes etc., through the user interface (UI). Mobile apps are also structured to enable such interactions: when the app is launched, a "home page" is shown that includes one or more *UI elements* (buttons, text boxes, other user interface elements). User interactions with these UI elements lead to other pages, which in turn may contain other UI elements. A user interaction may also result in local computation (*e.g.,* updating game state), network communication (*e.g.,* downloading ads or content), access to local sensors (*e.g.,* GPS), and access to local storage (*e.g.,* saving app state to storage). In the abstract, execution of a mobile app can be modeled as a transition graph where nodes represent various pages and edges represent transitions between pages. The goal of dynamic analysis is to navigate to all pages and to analyze apps' internal states and outputs at each page.

UI-Automation Frameworks. This commonality in the structure of mobile apps can be exploited to automatically analyze their dynamic properties. Recent research has done this using a UI automation framework, sometimes called a *monkey*, that systematically explores the app execution space. A monkey is a piece of software that runs on a mobile device or on an emulator, and extracts the user-interface structure of the current page (*e.g.,* the home page). This UI structure, analogous to the DOM structure of web pages, contains information about UI elements (buttons and other widgets) on the current page. Using this information, the monkey can, in an automated fashion, click a UI element, causing the app to transition to a new page. If the monkey has *not* visited this (or a similar) page, it can interact with the page by clicking its UI elements. Otherwise, it can click the "back" button to return to the previous page, and click another UI element to reach a different page.[1] In the abstract, each page corresponds to a *UI-state* and clicking a clickable UI element results in a state transition; using these, a monkey can effectively explore the UI-state transition graph.

2.2 Related Work on Dynamic Analysis of Mobile Apps

As discussed above, our work is an instance of a class of dynamic analysis frameworks. Such frameworks are widely

[1] Some apps do not include back buttons; this is discussed later.

System	Exploration target	Page transition inputs	Properties checked	Actions Taken	Instrumentation
AMC [18]	Distinct types of pages	UI events	Accessibility	None	No
DECAF [20]	Distinct types of pages containing ads	UI events	Ad layouts	None	No
SmartAds [23]	All pages	UI events	Page contents	None	Yes
A^{3}E [8]	Distinct types of pages	UI events	None	None	Yes
AppsPlayGround [24]	Distinct types of pages	UI events, Text inputs	Information flow	None	Yes
VanarSena [25]	Distinct types of pages	UI events, Text inputs	App crashes	Inject faults	Yes
ContextualFuzzing [19]	All pages	UI events	Crashes, performance	Change contexts	No
DynoDroid [22]	Code basic blocks	UI events, System events	App crashes	System inputs	No

Table 1: Recent work that has used a monkey tool for dynamic analysis

used in software engineering for unit testing and random (fuzz) testing. The field of software testing is rather large, so we do not attempt to cover it; the interested reader is referred to [6].

Monkeys have been recently used to analyze several dynamic properties of mobile apps (Table 1). AMC [18] evaluates the conformance of vehicular apps to accessibility requirements; for example, apps need to be designed with large buttons and text, to minimize driving distractions. DECAF [20], detects violations of ad placement and content policies in over 50,000 apps. SmartAds [23] crawls contents of from an app's pages to enable contextual advertising for mobile apps. A^{3}E [8] executes and visits app pages to uncover potential bugs. AppsPlayground [24] examines information flow for potential privacy leaks in apps. VanarSena [25], ContextualFuzzing [19], and DynoDroid [22] try to uncover app crashes and performance problems by exposing them to various external exceptional conditions, such as bad network conditions.

At a high-level, these systems share a common feature: they use a monkey to automate dynamic app execution and use custom code to analyze a specific runtime property as the monkey visits various app states. At a lower level, however, they differ in at least the following five dimensions.

Exploration target: This denotes what pages in an app are to be explored by the monkey. Fewer pages mean the monkey can perform the analysis faster, but that the analysis may be less comprehensive. AMC, A^{3}E, AppsPlayground, VanarSena aim to visit only pages of unique types. Their analysis goals do not require visiting two pages that are of same type but contain different contents (e.g., two pages in a News app that are instantiated from the same page class but displays different news articles), and hence they omit exploring such pages for greater speed. On the other hand, SmartAds requires visiting all pages with unique content. DECAF can be configured to visit only the pages that are of unique types and that are likely to contain ads.

Transition inputs: This denotes the inputs that the monkey provides to the app to cause transitions between pages. Most monkeys generate UI events, such as clicks and swipes, to move from one page to another. Some other systems, such as AppsPlayground and VanarSena, can provide text inputs to achieve a better coverage. Dynodroid can generate system inputs (e.g., the "SMS received" event).

Properties checked: This defines what runtime properties the analysis code checks. Different systems check different runtime properties depending on what their analysis logic requires. For example, DECAF checks various geometric properties of ads in the current page in order to identify ad fraud.

Actions taken at each page: This denotes what action the monkey takes at each page (other than transition inputs). While some systems do not take any actions, VanarSena, ContextualFuzzing, and DynoDroid creates various contextual faults (e.g., slow networks, bad user inputs) to check if the app crashes on those faults.

Binary instrumentation: This denotes whether the monkey runs an unmodified app or an instrumented app. VanarSena instruments app before execution in order to identify a small set of pages to explore. SmartAds instruments apps to retrieve application states.

Due to these differences, each work listed in Table 1 has developed its own automation components from scratch and tuned the tool to explore a specific property of the researchers' interest. The resulting tools have an intertwining of the app exploration logic and the logic required for analyzing the property of interest. This has meant that many of the advances developed to handle large scale studies are only utilizable in the context of the specific analyses and cannot be readily generalized to other analyses.

PUMA. As mentioned in Section 1, our goal is to build a generic framework called PUMA that enables scalable and programmable UI automation, and that can be customized for various types of dynamic analysis (including the ones in Table 1). PUMA separates the analysis logic from the automated navigation of the UI-state transition graph, allowing its users to (a) succinctly specify navigation hints for scalable app exploration and (b) separately specify the logic for analyzing the app properties. This has two distinct advantages. It can simplify the analysis of different app properties, since users do not need to also develop UI automation components, and the UI automation framework can evolve independently of the analysis logic. As we discuss later, the design of scalable and robust state exploration can be tricky, and PUMA users can benefit from improvements to the underlying monkey, since their analysis code is decoupled from the monkey itself. Moreover, PUMA can concurrently run similar analyses, resulting in better scaling of the dynamic analysis. We discuss these advantages below.

2.3 Framework Requirements

Table 1 and the discussion above motivates the following requirements for a programmable UI-automation framework:

- *Support for a wide variety of properties*: The goal of using a UI-automation tool is to help users analyze app properties. But it is hard (if not impossible) for the framework to predefine a set of target properties that are going to be useful for all types of analyses. Instead, the framework should provide a set of necessary abstractions that can enable users to specify properties of interest at a high level.

- *Flexibility in state exploration*: The framework should allow users to customize the UI-state exploration. At a high-level, UI-state exploration decides which UI element to click next, and whether a (similar) state has been visited before. Permitting programmability of these decisions will allow analyses to customize the monkey behavior in flexible ways that can be optimized for the analysis at hand.

- *Programmable access to program state*: Many of the analyses in Table 1 require access to arbitrary app state, not just UI properties, such as the size of buttons or the layout of ads. Examples of app state include dynamic invocations of permissions, network or CPU usage at any given point, or even app-specific internal state.

- *Support for triggered actions*: Some of the analyses in Table 1 examine app robustness to changes in environmental conditions (*e.g.,* drastic changes to network bandwidth) or exceptional inputs. PUMA must support injecting these runtime behaviors based on user-specified conditions (*e.g.,* change network availability just before any call to the network API).

These requirements raise significant research questions and challenges. For example, how can PUMA provide users with flexible and easy-to-use abstractions to specify properties that are unknown beforehand? Recall these properties can range from basic UI attributes to those that aim to diagnose various performance bottlenecks. Also, can it provide flexible control of the state exploration, given that the state space may be huge or even infinite? We now describe how PUMA provides meets these challenges.

3. PROGRAMMABLE UI-AUTOMATION

In this section, we describe PUMA, a programmable framework for dynamic analysis of mobile apps that satisfies the requirements listed in the previous section. We begin with an overview that describes how a user interacts with PUMA and the workflow within PUMA. We then discuss how users can specify analysis code using a PUMAscript, and then discuss the detailed design of PUMA and its internal algorithms.

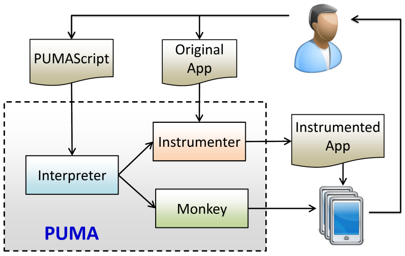

Figure 1: Overview of PUMA

We conclude the section by describing our implementation of PUMA for Android.

3.1 PUMA Overview and Workflow

Figure 1 describes the overall workflow for PUMA. A user provides two pieces of information as input to PUMA. The first is a set of app binaries that the user wants to analyze. The second is the user-specified code, written in a language called PUMAscript[2]. The script contains all information needed for the dynamic analysis.

In the first step of PUMA's workflow, the *interpreter* component interprets the PUMAscript specification and recognizes two parts in the script: monkey-specific directives and app-specific directives. The former provides necessary inputs or hints on how apps will be executed by the monkey tool, which is then translated as input to our *programmable monkey* component. The latter dictates which parts of app code are relevant for analysis, and specifies what actions are to be taken when those pieces of code are executed. These app-specific directives are fed as input to an *app instrumenter* component.

The *app instrumenter* component statically analyzes the app to determine parts of app code relevant for analysis and instruments the app in a manner described below. The output of this component is the instrumented version of input app that adheres to the app-specific directives in PUMAscript.

Then, the *programmable monkey* executes the instrumented version of each app, using the monkey-specific directives specified in the PUMAscript. PUMA is designed to execute the instrumented app either on a phone emulator, or on a mobile device. As a side effect of executing the app, PUMA may produce logs which contain outputs specified in the app-specific directives, as well outputs generated by the programmable monkey. Users can analyze these logs using analysis-specific code; such analysis code is not part of PUMA. In the remainder of this section, we describe these components of PUMA.

3.2 The PUMAScript Language

Our first design choice for PUMA was to either design a new domain-specific language for PUMAscript or implement it as an extension of some existing language. A new lan-

[2] In the rest of paper, we will use PUMAscript to denote both the language used to write analysis code and the specification program itself; the usage will be clear from the context.

guage is more general and can be compiled to run on multiple mobile platforms, but it may also incur a steeper learning curve. Instead, we chose the latter approach and implemented PUMAscript as a Java extension. This choice has its advantage of familiarity for programmers but also limits PUMA's applicability to some mobile platforms. However, we emphasize that the abstractions in our PUMAscript language are general enough and we should be able to port PUMA to other mobile platforms relatively easily, a task we have left to future work.

The next design challenge for PUMA was to identify abstractions that provide sufficient expressivity and enable a variety of analysis tasks, while still decoupling the mechanics of app exploration from analysis code. Our survey of related work in the area (Table 1) has influenced the abstractions discussed below.

Terminology. Before discussing the abstractions, we first introduce some terminology. The visual elements in a given page of the mobile app consist of one or more *UI element*. A UI element encapsulates a UI widget, and has an associated *geometry* as well as *content*. UI elements may have additional *attributes*, such as whether they are hidden or visible, clickable or not, etc.

The layout of a given page is defined by a *UI hierarchy*. Analogous to a DOM tree for a web page, a UI hierarchy describes parent-child relationships between widgets. One can programmatically traverse the UI hierarchy to determine all the UI elements on a given page, together with their attributes and textual content (image or video content associated with a UI element is usually not available as part of the hierarchy).

The *UI state* of a given page is completely defined by its UI hierarchy. In some cases, it might be desirable to define a more general notion of the state of an app page, which includes the internal program state of an app together with the UI hierarchy. To distinguish it from UI state, we use the term *total state* of a given app.

Given this discussion, a monkey can be said to perform a state traversal: when it performs a *UI action* on a UI element (*e.g.,* clicks a button), it initiates a state transition which may, in general, cause a completely different app page (and hence UI state) to be loaded. When this loading completes, the app is said to have reached a new state.

PUMAscript Design. PUMAscript is an event-based programming language. It allows programmers to specify *handlers* for *events*. In general, an event is an abstraction for a specific point in the execution either of the monkey or of a specific app. A handler for an event is an arbitrary piece of code that may perform various actions: it can keep and update internal state variables, modify the environment (by altering system settings), and, in some cases, access UI state or total state. This paradigm is an instance of aspect-oriented programming, where the analysis concerns are cleanly separated from app traversal and execution.

PUMAscript defines two kinds of events: *monkey-specific* events and *app-specific* events.

Monkey-specific Events. A monkey-specific event encapsulates a specific point in the execution of a monkey. A monkey is a conceptually simple tool[3], and Alg. (1) describes the pseudo-code for a generic monkey, as generalized from the uses of the monkey described in prior work (Table 1). The highlighted names in the pseudo-code are PUMA APIs that will be explained later. The monkey starts at an initial state (corresponding to its home page) for an app, and visits other states by deciding which UI action to perform (by selecting a clickable UI element in line 8), and performing the click (line 12). This UI action will, in general, result in a new state (line 13), and the monkey needs to decide whether this state has been visited before (line 15). Once a state has been fully explored, it is no longer considered in the exploration (lines 19-20).

Algorithm 1 Generic monkey tool. PUMA APIs for configurable steps are highlighted.

```
 1: while not all apps have been explored do
 2:     pick a new app
 3:     $S \leftarrow$ empty stack
 4:     push initial page to $S$
 5:     while $S$ is not empty do
 6:         pop an unfinished page $s_i$ from $S$
 7:         go to page $s_i$
 8:         pick next clickable UI element from $s_i$ // Next-Click
 9:         if user input is needed (e.g., login/password) then
10:             provide user input by emulating keyboard clicks // Text Input
11:         effect environmental changes // Modifying Environment
12:         perform the click
13:         wait for next page $s_j$ to load
14:         analyze page $s_j$// In-line Analysis
15:         flag $\leftarrow$ $s_j$ is equivalent to an explored page // State-Equivalence
16:         if not flag then
17:             add $s_j$ to $S$
18:         update finished clicks for $s_i$
19:         if all clicks in $s_i$ are explored then
20:             remove $s_i$ from $S$
21:         flag $\leftarrow$ monkey has used too many resources // Terminating App
22:         if flag or $S$ is empty then
23:             terminate this app
```

In this algorithm, most of the steps are mechanistic, but six steps involve *policy* decisions. The first is the decision of whether a state has been visited before (Line 15): prior work in Table 1 has observed that it is possible to reduce app exploration time with analysis-specific definitions of *state-equivalence*. The second is the decision of which UI action to perform next (Line 8): prior work in Table 1 has proposed using out of band information to *direct* exploration more efficiently, rather than randomly selecting UI actions. The third is a specification of user-input (Line 10): some apps require some forms of text input (*e.g.,* a Facebook or Google login). The fourth is a decision (Line 11) of whether to modify the environment as the app page loads: for example, one prior work [25] modifies network state to reduce benefit, with the

[3] However, as discussed later, the implementation of a monkey can be significantly complex.

aim of analyzing the robustness of apps to sudden resource availability changes. The fifth is analysis (Line 14): some prior work has performed in-line analysis (e.g., ad fraud detection [20]). Finally, the sixth is the decision of whether to terminate an app (Line 21): prior work in Table 1 has used fixed timeouts, but other policies are possible (*e.g.,* after a fixed number of states have been explored). PUMAscript separates policy from mechanism by modeling these six steps as events, described below. When these events occur, user-defined handlers are executed.

(1) State-Equivalence. This abstraction provides a customizable way of specifying whether states are classified as equivalent or not. The inputs to the handler for a state-equivalence event include: the newly visited state s_j, and the set of previously visited states $\mathcal{S}$. The handler should return `true` if this new state is equivalent to some previously visited state in $\mathcal{S}$, and `false` otherwise.

This capability permits an arbitrary definition of state equivalence. At one extreme, two states s_i and s_j are equivalent only if their total states are identical. A handler can code this by traversing the UI hierarchies of both states, and comparing UI elements in the hierarchy pairwise; it can also, in addition, compare program internal state pairwise.

However, several pieces of work have pointed out that this strict notion of equivalence may not be necessary in all cases. For example, to detect ad violations, it suffices to treat two states as equivalent if their UI hierarchies are "similar" in the sense that they have the same kinds of UI elements. Handlers can take one of two approaches to define such fuzzier notions of equivalence.

They can implement app-specific notions of similarity. For example, if an analysis were only interested in UI properties of specific types of buttons (like [18]), it might be sufficient to declare two states to be equivalent if one had at least one instance of each type of UI element present in the other.

A more generic notion of state equivalence can be obtained by defining *features* derived from states, then defining similarity based on distance metrics defined on the feature space. In DECAF [20], we defined a generic feature vector encoding the structure of the UI hierarchy, then used the cosine-similarity metric[4] with a user-specified similarity threshold, to determine state equivalence. This state equivalence function is built into PUMA, so a PUMAscript handler can simply invoke this function with the appropriate threshold.

A handler may also define a different set of features, or different similarity metrics. The exploration of which features might be appropriate, and how similarity thresholds affect state traversal is beyond the scope of this work.

(2) Next-Click. This event permits handlers to customize how to specify which item to click next. The input to a handler is the current UI state, together with the set of UI elements that have already been clicked before. A handler should return a pointer to the next UI element to click.

Handlers can implement a wide variety of policies with this flexibility. A simple policy may decide to explore UI elements sequentially, which may have good coverage, but increase exploration time. Alternatively, a handler may want to maximize the *types* of items clicked; prioritizing UI elements of different types over instances of a type of UI element that has been clicked before. These two policies are built into PUMA for user convenience.

Handlers can also use out-of-band information to implement *directed exploration*. Analytics from real users can provide insight into how real users prioritize UI actions: for example, an expert user may rarely click a *Help* button. Insights like these, or even actual traces from users, can be used to direct exploration to visit states that are more likely to be visited by real users. Another input to directed exploration is static analysis: the static analysis may reveal that button A can lead to a particular event handler that sends a HTTP request, which is of interest to the specific analysis task at hand. The handler can then prioritize the click of button A in every visited state.

(3) Text Input. The handler of this event provides the text input required for exploration to proceed. Often, apps require login-based authentication to some cloud-service before permitting use of the app. The input to the handler is the UI state and the text box UI element which requires input. The handler's output includes the corresponding text (login, password *etc.*), using which the monkey can emulate keyboard actions to generate the text. If the handler for this event is missing, and exploration encounters a UI element that requires text input, the monkey stops exploring the app.

(4) Modifying the Environment. This event is triggered just before the monkey clicks a UI element. The corresponding handler for this event takes as input the current UI state, and the UI element to be clicked. Based on this information, the handler may enable or disable devices, dynamically change network availability using a network emulator, or change other aspects of the environment in order to stress-test apps. This kind of modification is coarse-grained, in the sense that it occurs before the entire page is loaded. It is also possible to perform more fine-grained modifications (*e.g.,* reducing network bandwidth just before accessing the network) using app-specific events, described below. If a handler for this event is not specified, PUMA skips this step.

(5) In-line Analysis. The in-line analysis event is triggered after a new state has completed loading. The handler for this event takes as input the current total state; the handler can use the total state information to perform analysis-specific computations. For example, an ad fraud detector can analyze the layout of the UI hierarchy to ensure compliance to ad policies [20]. A PUMAscript may choose to forgo this step and perform all analyses off-line; PUMA outputs the explored state transition graph together with the total states for

[4] http://en.wikipedia.org/wiki/Cosine_similarity

this purpose.

(6) Terminating App Exploration. Depending on the precise definition of state equivalence, the number of states in the UI state transition graph can be practically limitless. A good example of this is an app that shows news items. Each time the app page that lists news items is visited, a new news item may be available which may cause the state to be technically not equivalent to any previously visited state. To counter such cases, most prior research has established practical limits on how long to explore an app. PUMA provides a default *timeout* handler for the termination decision event, which terminates an app after its exploration has used up a certain amount of wall-clock time. A PUMAscript can also define other handlers that make termination decisions based on the number of states visited, or CPU, network, or energy resources used.

App-specific Events. In much the same way that monkey-specific events abstract specific points in the execution of a generic monkey, an *app-specific event* abstracts a specific point in app code. Unlike monkey-specific events, which are predetermined because of the relative simplicity of a generic monkey, app-specific events must be user-defined. In a PUMAscript, an app-specific event is defined by naming an event and associating the named event with a codepoint set [16]. A codepoint set is a set of instructions (*e.g.,* bytecodes or invocations of arbitrary functions) in the app binary, usually specified as a regular expression on class names, method names, or names of specific bytecodes. Thus, a codepoint set defines a set of points in the app binary where the named event may be said to occur.

Once named events have been described, a PUMAscript can associate arbitrary handlers with these named events. These handlers have access to app-internal state and can manipulate program state, can output state information to the output logs, and can also perform finer-grained environmental modifications.

A sample PUMAscript. Listing 1 shows the listing of a PUMAscript designed to count the network usage of apps. A PUMAscript is effectively a Java extension, where a specific analysis is described by defining a new class inherited from a `PUMAScript` base class. This class (in our example, `NetworkProfiler`) defines handlers for monkey-specific events (lines 2-7), and also defines events and associated handlers for app-specific events. It uses the inbuilt feature-based similarity detector with a threshold that permits fuzzy state equivalence (line 3), and uses the default next-click function, which traverses each UI element in each state sequentially (line 6). It defines one app-specific event, which is triggered whenever execution invokes the HTTPClient library (lines 10-11), and defines two handlers, one (line 21) before the occurrence of the event (*i.e.,* the invocation) and another after (line 24) the occurrence of the event. These handlers respectively log the size of the network request and response. The total network usage of an app can be obtained by post-facto analysis of the log.

```
1  class NetworkProfiler extends PUMAScript {
2    boolean compareState(UIState s1, UIState s2) {
3      return MonkeyInputFactory.stateStructureMatch(s1,
            s2, 0.95);
4    }
5    int getNextClick(UIState s) {
6      return MonkeyInputFactory.nextClickSequential(s);
7    }
8    void specifyInstrumentation() {
9      Set<CodePoint> userEvent;
10     CPFinder.setBytecode("invoke.*", "HTTPClient.
          execute(HttpUriRequest request)");
11     userEvent = CPFinder.apply();
12     for (CodePoint cp:userEvent) {
13       UserCode code = new UserCode("Logger", "
            countRequest", CPARG);
14       Instrumenter.place(code, BEFORE, cp);
15       code = new UserCode("Logger", "countResponse",
            CPARG);
16       Instrumenter.place(code, AFTER, cp);
17     }
18   }
19 }
20 class Logger {
21   void countRequest (HttpUriRequest req) {
22     Log(req.getRequestLine().getUri().getLength());
23   }
24   void countResponse (HttpResponse resp) {
25     Log(resp.getEntity().getContentLength());
26   }
27 }
```

Listing 1: Count Network Usage

3.3 PUMA Design

PUMA incorporates a generic monkey (Alg. (1)), together with support for events and handlers. One or more PUMAscripts are input to PUMA, together with the apps to be analyzed. The PUMAscript interpreter instruments each app in a manner designed to trigger the app-specific events. One way to do this is to instrument apps to transfer control back to PUMA when the specified code point is reached. The advantage of this approach is that app-specific handlers can then have access to the explored UI states, but it would have made it harder for PUMA to expose app-specific internal state. Instead, PUMA chooses to instrument apps so that app-specific handlers are executed directly within the app context; this way, handlers have access to arbitrary program state information. For example, in line 22 of Listing 1, the handler can access the size of the HTTP request made by the app.

After each app has been instrumented, PUMA executes the algorithm described in Alg. (1), but with explicit events and associated handlers. The six monkey-specific event handlers are highlighted in Alg. (1) and are invoked at relevant points. Because app-specific event handlers are instrumented within app binaries, they are implicitly invoked when a specific UI element has been clicked (Line 12).

PUMA can also execute multiple PUMAscripts concurrently. This capability provides scaling of the analyses, since each app need only be run once. However, arbitrary concurrent execution is not possible, and concurrently executed scripts must satisfy two sets of conditions.

Consider two PUMAscripts A and B. In most cases, these scripts can be run concurrently only if the handlers for each monkey-specific event for A are identical to or a strict subset of the handlers for B. For example, consider the state equivalence handler: if A's handler visits a superset of the states visited by A and B, then, it is safe to concurrently execute A and B. Analogously, the next-click handler for A must be identical with that of B, and the text input handler for both must be identical (otherwise, the monkey would not know which text input to use). However, the analysis handler for the two scripts can (and will) be different, because this handler does not alter the sequence of the monkey's exploration. By a similar reasoning, for A and B to be run concurrently, their app-specific event handlers must be disjoint (they can also be identical, but that is less interesting since that means the two scripts are performing identical analyses), and they must either modify the environment in the same way or not modify the environment at all.

In our evaluation, we demonstrate this concurrent PUMA-script execution capability. In future work, we plan to derive static analysis methods by which the conditions outlined in the previous paragraph can be tested, so that it may be possible to automate the decision of whether two PUMAscripts can run concurrently. Finally, this static analysis can be simplified by providing, as PUMA does, default handlers for various events.

3.4 Implementation of PUMA for Android

We have designed PUMA to be broadly applicable to different mobile computing platforms. The abstractions PUMA uses are generic and should be extensible to different programming languages. However, we have chosen to instantiate PUMA for the Android platform because of its popularity and the volume of active research that has explored Android app dynamics.

The following paragraphs describe some of the complexity of implementing PUMA in Android. Much of this complexity arises because of lack of a complete native UI automation support in Android.

Defining a Page State. The UI state of an app, defined as the current topmost foreground UI hierarchy, is central to PUMA. The UI state might represent part of a screen (*e.g.,* a pop-up dialog window), a single screen, or more than one screen (*e.g.,* a webview that needs scrolling to finish viewing). Thus, in general, a UI state may cover sections of an app page that are not currently visible.

In Android, the UI hierarchy for an app's page can be obtained from `hierarchyviewer` [1] or the `uiautomator` [2] tool. We chose the latter because it supports many Android devices and has built-in support for UI event generation and handling, while the former only works on systems with debugging support (*e.g.,* special developer phones from google) and needs an additional UI event generator. However, we had to modify the `uiautomator` to intercept and access the UI hierarchy programmatically (the default tool only allows dumping and storing the UI state to external storage).

Supporting Page Scrolling. Since smartphones have small screens, it is common for apps to add scrolling support to allow users to view all the contents in a page. However, `uiautomator` only returns the part of the UI hierarchy currently visible. To overcome this limitation, PUMA scrolls down till the end of the screen, extracts the UI hierarchy in each view piecemeal, and merges these together to obtain a composite UI hierarchy that represents the UI state. This turns out to be tricky for pages that can be scrolled vertically and/or horizontally, since `uiautomator` does not report the direction of scrollability for each UI widget. For those that are scrollable, PUMA first checks whether they are horizontally or vertically scrollable (or both). Then, it follows a zig-zag pattern (scrolls horizontally to the right end, vertically down one view, then horizontally to the left end) to cover the non-visible portions of the current page. To merge the scrolled states, PUMA relies on the `AccessibilityEvent` listener to intercept the scrolling response, which contains hints for merging. For example, for `ListView`, this listener reports the start and the end entry indices in the scrolled view; for `ScrollView` and `WebView`, it reports the coordinate offsets with respect to the global coordinate.

Detecting Page Loading Completion. Android does not have a way to determine when a state has been completely loaded. State loading can take arbitrary time, especially if content needs to be fetched over the network. PUMA uses a heuristic that detects `WINDOW_CONTENT_CHANGED` events signaled by the OS, since this event is fired whenever there is a content change or update in the current view. For example, a page that relies on network data to update its UI widgets will trigger one such event every time it receives new data that causes the widget to be rendered. PUMA considers a page to be completely loaded when there is no content-changed event in a window of time that is conservatively determined from the inter-arrival times of previous events.

Instrumenting Apps. PUMA uses SIF [16] in the backend to instrument app binaries. However, other tools that are capable of instrumenting Android app binaries can also be used.

Environment Modifications by Apps. We observed that when PUMA runs apps sequentially on one device, it is possible that an app may change the environment (e.g., some apps turn off WiFi during their execution), affecting subsequent apps. To deal with this, PUMA restores the environment (turning on WiFi, enabling GPS, etc.) after completing each app, and before starting the next one.

Implementation Limitations. Currently, our implementation uses Android's `uiautomator` tool that is based on the underlying `AccessibilityService` in the OS. So any UI widgets that do not support such a service cannot be supported by our tool. For example, some user-defined widgets do not use any existing Android UI support at all, so are inaccessible to PUMA. However, in our evaluations described later, we find relatively few instances of apps that use user-defined widgets, likely because of Android's extensive support for UI programming.

Finally, PUMA does not support non-deterministic UI events like swipes, or other customized user gestures, which are fundamental problems for any monkey-based automation tool. In particular, this limitation rules out analysis of games, which is an important category of Android apps. To our knowledge, no existing monkeys have overcome this limitation. It may be possible to overcome this limitation by passively observing real users and "learning" user-interface actions, but we have left this to future work.

4. EVALUATION

The primary motivation for PUMA is rapid development of large-scale dynamic mobile app analyses. In this section, we validate that PUMA enables this capability: *in a space of two weeks, we were able to develop 7 distinct analyses and execute each of them on a corpus of 3600 apps.* Beyond demonstrating this, our evaluations provide novel insights into the Android app ecosystem. Before discussing these analyses, we discuss our methodology.

4.1 Methodology

Apps. We downloaded 18,962 top free apps[5], in 35 categories, from the Google Play store with an app crawler [4] that implements the Google Play API. Due to the incompleteness of the Dalvik to Java translator tool we use for app instrumentation [16], some apps failed the bytecode translation process, and we removed those apps. Then based on the app name, we removed foreign-language apps, since some of our analyses are focused on English language apps, as we discuss later. We also removed apps in the game, social, or wallpaper categories, since they either require many non-deterministic UI actions or do not have sufficient app logic code (some wallpaper apps have no app code at all). These filtering steps resulted in a pool of 9,644 apps spread over 23 categories, from which we randomly selected 3,600 apps for the experiments below. This choice was dictated by time constraints for our evaluation.

Emulators vs Phones. We initially tried to execute the monkey on emulators running concurrently on a single server. Android emulators were either too slow or unstable, and concurrency was limited by the performance of the graphics cards on the server. Accordingly, our experiments use 11 phones, each running an instance of PUMA: 5 Galaxy Nexus, 5 HTC One, and 1 Galaxy S3, all running Android 4.3. The corpus of 3,600 apps is partitioned across these phones, and the PUMA instance on each phone evaluates the apps in its partition sequentially. PUMA is designed to work on emulators as well, so it may be possible to scale the analyses by running multiple cloud instances of the emulator when the robustness of the emulators improves.

4.2 PUMA Scalability and Expressivity

To evaluate PUMA's expressivity and scalability, we used it to implement seven distinct dynamic analyses. Table 2 lists these analyses. In subsequent subsections, we describe these analyses in more detail, but first we make a few observations about these analyses and about PUMA in general.

First, we executed PUMAscripts for three of these analyses concurrently: UI structure classifier, ad fraud detection, and accessibility violation detection. These three analyses use similar notions of state equivalence and do not require any instrumentation. We could also have run the PUMAscripts for network usage profiler and permissions usage profiler concurrently, but did not do so for logistical reasons. These apps use similar notions of state equivalence and perform complementary kinds of instrumentation; the permissions usage profiler also instruments network calls, but in a way that does not affect the network usage profiler. We have verified this through a small-scale test of 100 apps: the combined analyses give the same results as the individual analyses, but use only the resources required to run one analysis. In future work, we plan to design an optimizer that automatically determines whether two PUMAscripts can be run concurrently and performs inter-script optimizations for concurrent analyses.

Second, we note that for the majority of our analyses, it suffices to have fuzzier notions of state equivalence. Specifically, these analyses declare two states to be equivalent if the cosine similarity between the feature vectors derived from the UI structure are above a specified threshold. In practice, this means that two states whose pages have different content, but similar UI structure, will be considered equivalent. (This is shown in Table 2, with the value "structural" in the "State-equivalent function" column.) For these analyses, *we are able to run the analysis to completion for each of our 3600 apps*: i.e., the analysis terminates when all applicable UI elements have been explored. For the single analysis that required an identical match, we had to limit the exploration of an app to 20 minutes. This demonstrates the importance of exposing programmable state equivalence in order to improve the scalability of analyses.

Third, PUMA enables extremely compact descriptions of analyses. Our largest PUMAscript is about 20 lines of code. Some analyses require non-trivial code in user-specified handlers; this is labeled "user code" in Table 2. The largest handler is 77 lines long. So, for most analyses, less than 100 lines is sufficient to explore fairly complex properties. In contrast, the implementation of DECAF [20] was over 4,300 lines of code, almost *50× higher*; almost 70% of this code went towards implementing the monkey functionality. Note that, some analyses require post-processing code; we do not count this in our evaluation of PUMA's expressivity, since that code is presumably comparable for when PUMA is used or when a hand-crafted monkey is used.

Finally, another measure of scalability is the speed of the monkey. PUMA's programmable monkey explored 15 apps per hour per phone, so in about 21 hours we were able to

[5] The versions of these apps are those available on Oct 3, 2013.

run our structural similarity analysis on the entire corpus of apps. This rate is faster than the rates reported in prior work [18, 20]. The monkey was also able to explore about 65 app *states* per hour per phone for a total of over 100,000 app states across all 7 analyses. As discussed above, PUMA ran to completion for our structural similarity-based analyses for every app. However, we do not evaluate coverage, since our exploration techniques are borrowed from prior work [20] and that work has evaluated the coverage of these techniques.

4.3 Analysis 1: Accessibility Violation Detection

Best practices in app development include guidelines for app design, either for differently-abled people or for use in environments with minimal interaction time requirements (*e.g.,* in-vehicle use). Beyond these guidelines, it is desirable to have automated tests for *accessibility compliance*, as discussed in prior work [18]. From an app store administrator's perspective, it is important to be able to classify apps based on their accessibility support so that users can be more informed in their app choices. For example, elderly persons who have a choice of several email apps may choose the ones that are more accessible (*e.g.,* those that have large buttons with enough space between adjacent buttons.)

In this dynamic analysis, we use PUMA to detect a subset of accessibility violations studied in prior work [18]. Specifically, we flag the following violations: if a state contains more than 100 words; if it contains a button smaller than 80mm^2; if it contains two buttons whose centers are less than 15mm apart; and if it contains a scrollable UI widget. We also check if an app requires a significant number of user interactions to achieve a task by computing the maximum shortest *round-trip path* between any two UI states based on the transition graph generated during monkey exploration.

This prior work includes other accessibility violations: detecting distracting animations can require a human-in-the-loop, and is not suitable for the scale that PUMA targets; and analyzing the text contrast ratio requires OS modifications. Our work scales this analysis to a much larger number of apps (3,600 vs. 12) than the prior work, demonstrating some of the benefits of PUMA.

Our PUMAscript has 11 lines of code, and is similar in structure to ad fraud detection. It uses structural matching for state equivalence, and detects these accessibility violations using an in-line analysis handler.

Table 3 shows the number of apps falling into different categories of violations, and the number of apps with more than one type of violation. We can see that 475 apps have maximum round-trip paths greater than 10 (the threshold used in [18]), 552 for word count, 1,276 for button size, 1,147 for button distance and 2,003 for scrolling. Thus, almost 55% of our apps violate the guideline that suggests not having a scrollable widget to improve accessibility. About one third of the violating apps have only one type of violation and less than one third have two or three types of violations. Less than one tenth of the apps violate all five properties.

	user actions per task	words count	button size	button distance	scrolling
#apps	475	552	1276	1147	2003
	1 type	2 types	3 types	4 types	5 types
#apps	752	683	656	421	223

Table 3: Accessibility violation results

This suggests that most apps in current app stores are not designed with general accessibility or vehicular settings in mind. An important actionable result from our findings is that our analyses can be used to automatically tag apps for "accessibility friendliness" or "vehicle unfriendliness". Such tags can help users find relevant apps more easily, and may incentivize developers to target apps towards segments of users with special needs.

4.4 Analysis 2: Content-based App Search

All app stores allow users to search for apps. To answer user queries, stores index various app metadata (e.g., app name, category, developer-provided description, etc.) That index does not use app content—content that an app reveals at runtime to users. Thus, a search query (e.g., for a specific recipe) can fail if the query does not match any metadata, even though the query might match the dynamic runtime content of some of these apps (e.g., culinary apps).

One solution to the above limitation is to crawl app content (by dynamic analysis) and index this content as well. We program PUMA to achieve this. Our PUMAscript for this analysis contains 14 lines of code and specifies a strong notion of state equivalence: two states are equivalent only if their UI hierarchy is identical and their contents are identical. Since the content of a given page can change dynamically, even during exploration, the exploration may, in theory, never terminate. So, we limit each app to 20 minutes (using PUMA's terminating app exploration event handler). Finally, the PUMAscript scrapes the textual content from the UI hierarchy in each state and uses the in-line analysis event handler to log this content.

We then post-process this content to build three search indices: one that uses the app name alone, a second that includes the developer's description, and a third that also includes the crawled content. We use Apache Lucene[6] an open source, full-featured text search engine, for this purpose.

We now demonstrate the efficacy of content-based search for apps. For this, we use two search-keyword datasets to evaluate the generated indices: (1) 200 most popular app store queries[7] and (2) a trace of 10 million queries from the Bing search engine. By re-playing those queries on the three indices, we find (Table 4), that the index with crawled con-

[6] http://lucene.apache.org/core/

[7] http://goo.gl/JGyO5P

	Properties studied	State-equivalence	App instrumentation	PUMAscript (LOC)	User code (LOC)
Accessibility violation detection	UI accessibility violation	structural	no	11	77
Content-based app search	in-app text crawling	exact	no	14	0
UI structure classifier	structural similarity in UI	structural	no	11	0
Ad fraud detection	ad policy violation	structural	no	11	74
Network usage profiler	runtime network usage	structural	yes	19	8
Permission usage profiler	permission usage	structural	yes	20	5
Stress testing	app robustness	structural	yes	16	5

Table 2: List of analyses implemented with PUMA

Keyword Type	Number	Search Type	Rate of Queries with Valid Search Return ($\geq$1)	Statistics of Valid Return			
				Min	Max	Mean	Median
App Store Popular Keywords	200	Name	68%	1	115	17	4
		Name + Desc.	93%	1	1234	156.54	36.50
		Name + Desc. + Crawl	97%	1	1473	200.46	46
Bing Trace Search Keywords	9.5 million	Name	54.09%	1	311	8.31	3
		Name + Desc.	81.68%	1	2201	199.43	66
		Name + Desc. + Crawl	85.51%	1	2347	300.37	131

Table 4: Search result

tent yields at least 4% more non-empty queries than the one which uses app metadata alone. More importantly, on average, each query returns about 50 more apps (from our corpus of 3,600) for the app store queries and about 100 more apps for the Bing queries.

Here are some concrete examples to demonstrate the value of indexing app content. For the search query "Jewelery deals", the metadata-based index returned many "deals" and "jewelery" apps, while the content-based index returned as the top result an app (*Best Deals*) that was presumably advertising a deal for a jewelry store[8]. Some queries (*e.g.,* "xmas" and "bejeweled") returned no answers from the metadata-based index, but the content-based index returned several apps that seemed to be relevant on manual inspection. These examples show that app stores can greatly improve search relevance by crawling and indexing app contents, and PUMA provides a simple way to crawl the data.

4.5 Analysis 3: UI Structure Classifier

In this analysis, we program PUMA to cluster apps based on their UI state transition graphs so that apps within the same cluster have the same "look and feel". The clusters can be used as input to clone detection algorithms [14], reducing the search space for clones: the intuition here is that the UI structure is the easiest part to clone and cloned apps might have very similar UI structures to the original one. Moreover, developers who are interested in improving the UI design of their own app can selectively examine a few apps within the same cluster as their own apps and do not need to exhaustively explore the complete app space.

The PUMAscript for this analysis is only 11 lines and uses structural page similarity to define state equivalence. It simply logs UI states in the *in-line analysis* event handler. After the analysis, for each app, we represent its UI state transition graph by a binary adjacent matrix, then perform Singular Value Decomposition[9] (SVD) on the matrix, and extract the Singular Value Vector. SVD techniques have been widely used in many areas such as general classification, pattern recognition and signal processing. Since the singular vector has been sorted by the importance of singular values, we only keep those vector elements (called *primary singular values*) which are greater than ten times the first element. Finally, the Spectral Clustering[10] algorithm is employed to cluster those app vectors, with each entry of the similarity matrix being defined as follows:

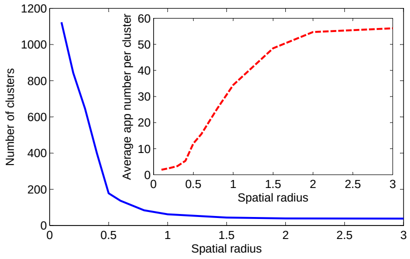

Figure 2: App Clustering for UI Structure Classification

$$m_{ij} = \begin{cases} 0 & , dim(\mathbf{v}_i) \neq dim(\mathbf{v}_j) \text{ or } d_{ij} > r_{spatial} \\ e^{-d_{ij}} & , \text{otherwise} \end{cases}$$

where $\mathbf{v}_i$ and $\mathbf{v}_j$ are the singular vectors of two different apps i and j, and d_{ij} is the Euclidean distance between them. $dim()$ gives the vector dimension, and we only consider two apps to be in a same cluster if the cardinality of their primary singular values are the same. Finally, the radius $r_{spatial}$ is a tunable parameter for the algorithm – the larger the radius, the farther out the algorithm searches for clusters around a given point (singular vector).

Following the above process, Figure 2 shows the number

[8] In practice, for search to be effective, apps with dynamic content need to be crawled periodically.

[9] http://en.wikipedia.org/wiki/Singular_value_decomposition

[10] http://en.wikipedia.org/wiki/Spectral_clustering

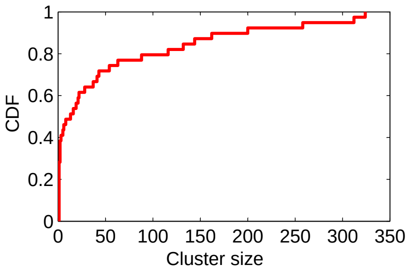

Figure 3: Cluster size for $r_{spatial} = 3$

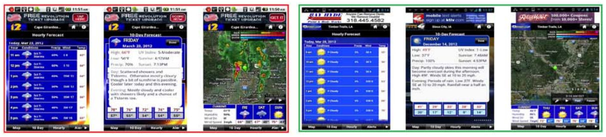

Figure 4: An app clone example (one app per rectangle)

of clusters and average apps per cluster for different spatial radii. As the radius increases, each cluster becomes bigger and the number of clusters increases, as expected. The number of clusters stabilizes beyond a certain radius and reaches 38 for a radius of 3. The CDF of cluster size for $r_{spatial} = 3$ is shown in Figure 3. By manually checking a small set of apps, we confirm that apps in a same cluster have pages with very similar UI layouts and transition graphs.

Our analysis reveals a few interesting findings. First, there exists a relatively small number of UI design patterns (i.e., clusters). Second, the number of apps in each cluster can be quite different (Figure 3), ranging from one app per cluster to more than 300 apps, indicating that some UI design patterns are more common than the others. Third, preliminary evaluations also suggest that most apps from a developer fall into the same cluster; this is perhaps not surprising given that developers specialize in categories of apps and likely reuse significant code across their offerings. Finally, manual verification reveal the existence of app clones For example, Figure 4 shows two apps from one cluster have nearly the same UI design with slightly different color and button styles, but developed by different developers[11].

4.6 Analysis 4: Ad Fraud Detection

Recent work [20] has used dynamic analysis to detect various *ad layout frauds* for Windows Store apps, by analyzing geometry (size, position, etc.) of ads during runtime. Examples of such frauds include (a) hidden ads: ads hidden behind other UI controls so the apps appear to be ad-free; (b) intrusive ads: ads placed very close to or partially behind clickable controls to trigger inadvertent clicks; (c) too many ads: placing too many ads in a single page; (d) small ads: ads too small to see. We program PUMA to detect similar frauds in Android apps.

Our PUMAscript for ad fraud detection catches small, intrusive, and too many ads per page. We have chosen not to implement detection of hidden ads on Andoid, since, unlike Microsoft's ad network [5], Google's ad network does not pay developers for ad impressions [3], and only pays them by ad clicks, so there is no incentive for Android developers to hide ads.

violation	small	many	intrusive	1 type	2 types	3 types
#apps	13	7	10	3	3	7

Table 5: Ad fraud violation results

Our PUMAscript requires 11 lines and uses structural match for state equivalence. It checks for ad violations within the in-line analysis handler; this requires about 74 lines of code. This handler traverses the UI view tree, searches for the WebView generated by ads, and checks its size and relationship with other clickable UI elements. It outputs all the violations found in each UI state.

Table 5 lists the number of apps that have one or more violations. About 13 out of our 3,600 apps violate ad policies. Furthermore, all 13 apps have small ads (which can improve the user experience by devoting more screen real estate to the app, but can reduce the visibility of the ad and adversely affect the advertiser). Seven apps show more than one ad on at least one of their pages, and 10 apps display ads in a different position than required by ad networks. Finally, if we examine violations by type, 7 apps exhibit all three violations, 3 apps exhibit one and 3 exhibit two violations.

These numbers appear to be surprisingly small, compared to results reported in [20]. To understand this, we explored several explanations. First, we found that the Google AdMob API enforces ad size, number and placement restrictions, so developers cannot violate these policies. Second, we found that 10 of our 13 violators use use ad providers other than AdMob, like millennialmedia, medialets and LeadBolt. These providers' API gives developers the freedom to customize ad sizes, conflicting with AdMob's policy of predefined ad size. We also found that, of the apps that did not exhibit ad fraud, only about half used AdMob and the rest used a wide variety of ad network providers. Taken together, these findings suggest that the likely reason the incidence of ad fraud is low in Android is that developers have little incentive to cheat, since AdMob pays for clicks and not impressions (all the frauds we tested for are designed to inflate impressions). In contrast, the occurrence of ad fraud in Windows phones is much higher because (a) 90% of the apps use the Microsoft ad network, (b) that network's API allows developers to customize ads, and (c) the network pays both for impressions as well as clicks.

4.7 Analysis 5: Network Usage Profiler

About 62% of the apps in our corpus need to access resources from the Internet to function. This provides a rough estimate of the number of cloud-enabled mobile apps in the Android marketplace, and is an interesting number in its own right. But beyond that, it is important to quantify the network

[11] We emphasize that clone detection requires sophisticated techniques well beyond UI structure matching; designing clone detection algorithms is beyond the scope of the paper

usage of these apps, given the prevalence of usage-limited cellular plans, and the energy cost of network communication [15].

PUMA can be used to approximate the network usage of an app by dynamically executing the app and measuring the total number of bytes transferred. Our PUMAscript for this has 19 lines of code, and demonstrates PUMA's ability to specify app instrumentation. This script specifies structural matching for state equivalence; this can undercount the network usage of the app, since PUMA would not visit similar states. Thus, our results present lower bounds for network usage of apps. To count network usage, our PUMAscript specifies a user-defined event that is triggered whenever the HTTPClient library's `execute` function is invoked (Listing 1). The handler for this event counts the size of the request and response.

Figure 5 shows the CDF of network usage for 2218 apps; the *x-axis is in log scale*. The network usage across apps varies by *6 orders of magnitude* from 1K to several hundred MB.

Half the apps use more than 206KB of data, and about 20% use more than 1MB of data. More surprisingly, 5% apps use more than 10MB data; 100 times more than the lowest 40% of the apps. The heaviest network users (the tail) are all video streaming apps that stream news and daily shows. For example, "CNN Student News" app, which delivers podcasts and videos of the top daily news items to middle and high school students has a usage over 700MB. We looked at 508 apps that use more than 1MB data and classified based on their app categories. The top five are "News and Magazines", "Sports", "Library and Demo", "Media and Video", and "Entertainment". This roughly matches our expectation that these heavy hitters would be heavy users of multimedia information.

This diversity in network usage suggests that it might be beneficial for app stores to automatically tag apps with their approximate network usage, perhaps on a logarithmic scale. This kind of information can help novice users determine whether they should use an app when Wi-Fi is unavailable or not, and may incentivize developers to develop bandwidth-friendly apps.

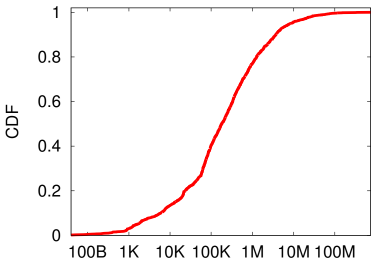

Figure 5: Network traffic usage

4.8 Analysis 6: Permission Usage Profiler

Much research has explored the Android security model, and the use of permissions. In particular, research has tried to understand the implication of permissions [13], designed better user interfaces to help users make more informed decisions [28], and proposed fine-grained permissions [17].

In this analysis, we explore the runtime use of permissions and relate that to the number of permissions requested by an app. This is potentially interesting because apps developers may request more permissions than are actually used in the code. Static analysis can reveal an upper bound on the permissions needed, but provides few hints on actual permissions usage.

With PUMA, we can implement a permission usage profiler, which logs every permission usage during app execution. This provides a lower bound on the set of permission required. We use the permission maps provided by [7]. Our PUMAscript has 20 lines of code. It uses a structural-match monkey and specifies a user-level event that is triggered when any API call that requires permissions is invoked (these API calls are obtained from [7]). The corresponding instrumentation code simply logs the permissions used.

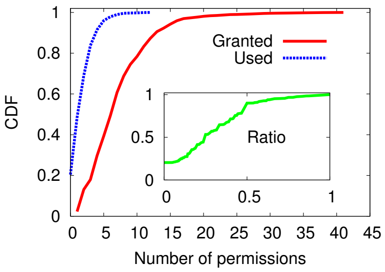

Figure 6: Permission usage: granted vs used

Figure 6 shows the CDF of the number of permissions requested and granted to each of the 3600 apps as well as those used during app exploration. We can see that about 80% are granted less than 15 permissions (with a median of 7) but this number can be as high as 41. Apps at the high end of this distribution include antivirus apps, a battery optimization tool, or utilities like "Anti-Theft" or "AutomateIt". These apps need many permissions because the functionalities they provide require them to access various system resources, sensors and phone private data.

At runtime, apps generally use fewer permissions than granted; about 90% of them used no more than 5 permissions, or no more than half of granted ones. While one expects the number of permissions used in runtime is always less than granted, but the surprisingly low runtime permission usage (about half the apps use less than 30% of their permissions) may suggest that some app developers might request for more permissions than actually needed, increasing the security risks.

4.9 Analysis 7: Stress Testing

Mobile apps are subject to highly dynamic environments, including varying network availability and quality, and dynamic sensor availability. Motivated by this, recent work [25] has explored random testing of mobile apps at scale using a monkey in order to understand app robustness to these dynamics.

In this analysis, we demonstrate PUMA can be used to script similar tests. In particular, we focus on apps that use HTTP and inject null HTTP responses by instrumenting the app code, with the goal of understanding whether app developers are careful to check for such errors. The PUMAscript for this analysis has 16 lines of code to specify a structural-match monkey and defines the same user-defined event as the network usage profile (Listing 1). However, the corresponding event handler replaces the HTTPClient library invocation with a method that returns a null response. During the experiment, we record the system log (`logcat` in Android) to track exception messages and apps that crash (the Android system logs these events).

In our experiments, apps either crashed during app exploration, or did not crash but logged a null exception, or did not crash and did not log an exception. Out of 2218 apps, 582 (or 26.2%) crashed, 1287 (or 58%) continued working without proper exception handling. Only 15.7% apps seemed to be fairly robust to our injected fault.

This is a fairly pessimistic finding, in that a relatively small number of apps seem robust to a fairly innocuous error condition. Beyond that, it appears that developers don't follow Android development guidelines which suggest handling network tasks in a separate thread than the main UI thread. The fact that 26% of the apps crash suggest that their network handling was performed as part of the main UI thread, and they did not handle this error condition gracefully. This analysis suggests a different usage scenario for PUMA: as an online service that can perform random testing on an uploaded app.

5. CONCLUSION

In this paper, we have described the design and implementation of PUMA, a programmable UI automation framework for conducting dynamic analyses of mobile apps at scale. PUMA incorporates a generic monkey and exposes an event driven programming abstraction. Analyses written on top of PUMA can customize app exploration by writing compact event handlers that separate analysis logic from exploration logic. We have evaluated PUMA by programming seven qualitatively different analyses that study performance, security, and correctness properties of mobile apps. These analyses exploit PUMA's ability to flexibly trade-off coverage for speed, extract app state through instrumentation, and dynamically modify the environment. The analysis scripts are highly compact and reveal interesting findings about the Android app ecosystem.

6. REFERENCES

[1] Android hierarchyviewer. http://developer.android.com/tools/help/hierarchy-viewer.html.

[2] Android uiautomator. http://developer.android.com/tools/help/uiautomator/index.html.

[3] Google admob. http://www.google.com/ads/admob/.

[4] Google Play crawler. https://github.com/Akdeniz/google-play-crawler.

[5] Microsoft advertising. http://advertising.microsoft.com/en-us/splitter.

[6] Software Testing Research Survey Bibliography. http://web.engr.illinois.edu/~taoxie/testingresearchsurvey.htm.

[7] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. PScout: analyzing the Android permission specification. In *Proc. of ACM CCS*, 2012.

[8] T. Azim and I. Neamtiu. Targeted and Depth-first Exploration for Systematic Testing of Android Apps. In *Proc. of ACM OOPSLA*, 2013.

[9] J. Crussell, C. Gibler, and H. Chen. Attack of the Clones: Detecting Cloned Applications on Android Markets. In *Proc. of ESORICS*, 2012.

[10] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting Privacy Leaks in iOS Applications. In *Proc. of NDSS*, 2011.

[11] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proc. of ACM OSDI*, 2010.

[12] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A Study of Android Application Security. In *Proc. of Usenix Security*, 2011.

[13] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android Permissions: User Attention, Comprehension, and Behavior. In *Proc. of SOUPS*, 2012.

[14] C. Gibler, R. Stevens, J. Crussell, H. Chen, H. Zang, and H. Choi. AdRob: Examining the Landscape and Impact of Android Application Plagiarism. In *Proc. of ACM MobiSys*, 2013.

[15] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan. Estimating Mobile Application Energy Consumption Using Program Analysis. In *Proc. of ICSE*, 2013.

[16] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan. SIF: A Selective Instrumentation Framework for Mobile Applications. In *Proc. of ACM MobiSys*, 2013.

[17] J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. Millstein. Dr. Android and Mr. Hide: Fine-grained Permissions in Android Applications. In *Proc. of SPSM*, 2012.

[18] K. Lee, J. Flinn, T. Giuli, B. Noble, and C. Peplin. AMC: Verifying User Interface Properties for Vehicular Applications. In *Proc. of ACM MobiSys*, 2013.

[19] C.-J. M. Liang, D. N. Lane, N. Brouwers, L. Zhang, B. Karlsson, R. Chandra, and F. Zhao. Contextual Fuzzing: Automated Mobile App Testing Under Dynamic Device and Environment Conditions. Technical Report MSR-TR-2013-100, Microsoft Research, 2013.

[20] B. Liu, S. Nath, R. Govindan, and J. Liu. DECAF: Detecting and Characterizing Ad Fraud in Mobile Apps. In *Proc. of NSDI*, 2014.

[21] B. Livshits and J. Jung. Automatic Mediation of Privacy-Sensitive Resource Access in Smartphone Applications. In *Proc. of Usenix Security*, 2013.

[22] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An Input Generation System for Android Apps. In *Proc. of ESEC/FSE*, 2013.

[23] S. Nath, X. F. Lin, L. Ravindranath, and J. Padhye. SmartAds: Bringing Contextual Ads to Mobile Apps. In *Proc. of ACM MobiSys*, 2013.

[24] V. Rastogi, Y. Chen, and W. Enck. AppsPlayground: automatic security analysis of smartphone applications. In *Proc. of ACM CODASPY*, 2013.

[25] L. Ravindranath, S. Nath, J. Padhye, and H. Balakrishnan. Automatic and Scalable Fault Detection for Mobile Applications. Technical Report MSR-TR-2013-98, Microsoft Research, 2013.

[26] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh. AppInsight: mobile app performance monitoring in the wild. In *Proc. of ACM OSDI*, 2012.

[27] L. Ravindranath, J. Padhye, R. Mahajan, and H. Balakrishnan. Timecard: Controlling User-perceived Delays in Server-based Mobile Applications. In *Proc. of ACM SOSP*, 2013.

[28] S. Rosen, Z. Qian, and Z. M. Mao. AppProfiler: A Flexible Method of Exposing Privacy-related Behavior in Android Applications to End Users. In *Proc. of ACM CODASPY*, 2013.