CuriousDroid: Automated User Interface Interaction for Android Application Analysis Sandboxes

Patrick Carter[1], Collin Mulliner[1], Martina Lindorfer[2], William Robertson[1], and Engin Kirda[1]

[1] Northeastern University, Boston, MA, USA
{pdc,crm,wkr,ek}@ccs.neu.edu
[2] SBA Research, Vienna, Austria
mlindorfer@iseclab.org

Abstract. Mobile computing has experienced enormous growth in market share and computational power in recent years. As a result, mobile malware is becoming more sophisticated and more prevalent, leading to research into dynamic sandboxes as a widespread approach for detecting malicious applications. However, the event-driven nature of Android applications renders critical the capability to automatically generate deterministic and intelligent user interactions to drive analysis subjects and improve code coverage. In this paper, we present CuriousDroid, an automated system for exercising Android application user interfaces in an *intelligent*, *user-like* manner. CuriousDroid operates by decomposing application user interfaces on-the-fly and creating a context-based model for interactions that is tailored to the current user layout. We integrated CuriousDroid with Andrubis, a well-known Android sandbox, and conducted a large-scale evaluation of 38,872 applications taken from different data sets. Our evaluation demonstrates significant improvements in both end-to-end sample classification as well as increases in the raw number of elicited behaviors at runtime.

Keywords: User Interface Analysis, Android, Dynamic Analysis

1 Introduction

Mobile computing has experienced enormous growth since the introduction of Apple's iPhone in 2007. A 2011 poll conducted by the Pew Research Center showed that 85% of Americans owned a cell phone, of which more than 50% were smartphones [22]. The Android operating system was released in 2008 and has since gained a significant share of the market, comprising 85% of smartphone shipments in the second quarter of 2014, up from 75% just over a year before [24]. With the growing number of mobile users worldwide, the increasing power of these devices, and the corresponding growth of the mobile economy, mobile malware has similarly grown in both sophistication and prevalence [11].

In response, considerable research has focused on dynamic analysis sandboxes as a general approach for detecting malicious applications [21,23,26]. In contrast to static approaches [8–10], dynamic analysis is able to precisely characterize the

runtime behavior of an application under test, or AUT, over concrete inputs, as well as deal with several techniques that pose significant difficulty for static analysis such as static code obfuscation, dynamic code loading, and the use of native code.

Despite its advantages, dynamic analysis can fail to provide useful insight into test subject behavior due to incomplete coverage. One feature of the Android platform that exacerbates this problem is the event-driven nature of its applications, where security-relevant code is often only executed in response to external stimuli such as interactions with the user interface. Current standard practice, therefore, is to use standard off-the-shelf tools such as the Monkey [4] or MonkeyRunner [3], which provide random sequences of user inputs and execute pre-scripted UI tests, respectively. Both of these tools are problematic in the context of large-scale dynamic analysis for different reasons: pre-scripting user interactions simply does not scale, and on the other hand random inputs lead to low code coverage.

In this paper, we introduce CuriousDroid, an automated system for driving Android application user interfaces in an *intelligent*, *user-like* manner. CuriousDroid operates by decomposing application user interfaces on-the-fly and creating a context-based model for interactions with an application. Unlike the Monkey, it is designed to deliver user interactions based upon actual application layouts discovered at draw-time. It significantly improves upon the capabilities of the MonkeyRunner since it can determine a set of natural, user-like interactions without prior knowledge of the application. Using structural decomposition of on-screen layouts and automated identification and classification of interactive views, CuriousDroid is able to generate a series of interactions that emulate typical human interaction.

In this paper, we show that CuriousDroid would be highly useful in a malware triage role, greatly reducing the burden on manual analysts in terms of numbers of new malicious samples to analyze. In particular, one of our evaluation data sets contained 8,827 applications that could not be classified as either benign or malicious. Using CuriousDroid to reclassify the data set resulted in 2,246 likely malicious applications, a significant reduction.

While prior work has examined more sophisticated user input generation [5, 6, 15, 16, 20, 27], we distance ourselves from these efforts by precisely quantifying the effects of human-like user interactions for large-scale dynamic analysis, as well as requiring no modifications to the operating system nor any static analysis component.

To summarize, this paper makes the following contributions.

- We introduce CuriousDroid, a system for automatically generating user-like UI interactions for Android applications. CuriousDroid uses dynamic instrumentation, application layout decomposition, and heuristic input generation to explore Android applications in an intelligent, user-like manner.
- We integrated CuriousDroid with the well-known Andrubis malware analysis sandbox, replacing the Monkey with CuriousDroid to drive the UI of analysis subjects.

– We conducted a large-scale evaluation of CuriousDroid using 38,872 applications from different data sets. Our evaluation demonstrates that our system improves the analysis results of Andrubis, eliciting behaviors that were not observed when relying upon random UI interactions.

2 Background and Motivation

Android is an open source mobile operating system that has enjoyed enormous success in recent years. The backbone of the OS is a modified Linux kernel targeted towards embedded devices with limited power, memory, and storage. Applications are written primarily in Java with the option of utilizing the Java Native Interface (JNI) via Android's Native Development Kit (NDK) to leverage existing libraries and optimize for performance.

Android applications can consist of four basic component types: `Activity`, `Service`, `BroadcastReceiver`, and `ContentProvider`. Activities provide the basic UI structure of an application, where each screen displayed to the user corresponds to an `Activity`. Services are meant to run in the background, separate from the main UI thread and are useful for operations that should not affect the UI thread, e.g., downloading content. Broadcast receivers are used to listen for system-wide events such as incoming SMS, phone calls, or emails. Content providers allow applications to make data available for use by other applications.

Activities. Activities are the most important component of the system in terms of the UI. An application consists of one or more activities, only one of which can be visible to the user at any given time. Normally, applications specify the UI design for each activity using resource files. Whenever the Android framework wants to display a given activity, the resource file is loaded and displayed to the user. Additionally, an application can create and add UI elements programmatically during runtime. These UI elements are not part of the resource files contained in an Android application package (APK).

To cover both statically- and dynamically-generated UI elements, CuriousDroid analyzes an application's UI at runtime. Our runtime analysis is based on dynamic instrumentation of the target application. Using dynamic instrumentation, we hook the functionality that is responsible for managing the application's UI. Dynamic instrumentation has the further benefit that we do not have to modify the Android framework, source code, or the application binary.

Dynamic analysis and UI exploration. Because Android applications are event-driven and many important events occur through UI interactions, performing a dynamic analysis of Android applications using the Monkey as a driver – which simply generates random event sequences – is problematic. Consider an activity that requires a user to enter an email address and password to register an account before using the application. If the application performs any kind of input validation on those fields, as is often the case, it is highly unlikely that the Monkey would be able to provide a value that satisfies the validity check, if it is able to enter any input into the required fields at all.

CuriousDroid is intended to remedy this problem by driving Android applications using intelligent, user-like interactions in order to increase the likelihood that any malicious behavior contained therein will be identified by the analysis as a whole.

3 System Overview

Since Android applications are mostly UI-driven, applications only execute the majority of their code after receiving external input, such as from a human user. Without realistic inputs tailored to the current application UI context, dynamic analyses might not explore interesting, security-relevant code, leading to inaccurate classification results. CuriousDroid aims to solve this problem by interacting with applications as normal users would, with the goal of increasing application coverage and eliciting more runtime behaviors in order to improve the results of the entire analysis.

To that end, CuriousDroid iterates over Android activities in three phases: *user interface decomposition*, *input inference*, and *input generation*. For each activity discovered by the system, the hierarchy of views contained in the activity is extracted using dynamic instrumentation. Then, the system uses a number of heuristics to infer the types of user inputs, or interactions, the views expect (if any). Finally, suitable inputs are generated. Any observed transitions to subsequent activities are added to a work queue for later processing. Activities that have previously been explored are recorded and, if encountered again, CuriousDroid attempts to explore a different path from that point in the UI. CuriousDroid intercepts events that might lead to early termination of the exploration, for instance when the Back button is pressed and the current activity is at the top of the activity stack. An overview of this process is shown in Figure 1a.

Supporting this process are two components: the UIAnalyzer and InputDriver. The UIAnalyzer uses dynamic instrumentation to inject itself into the target application, and is responsible for analyzing the UI, inferring context, and tracking visited activities. The InputDriver is executed as a separate process, and is responsible for sending user inputs to AUT.

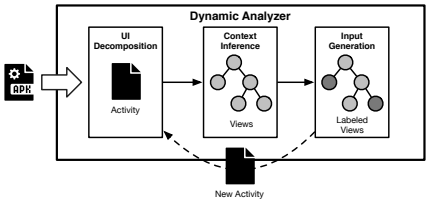
We designed CuriousDroid to be agnostic of its environment – that is, it is intended to run on any device or emulator with minimal effort. As one of the goals of CuriousDroid is to provide a generic automated UI interaction tool that can be deployed on any kind of Android application or malware analysis platform, our system does not require modification of the Android platform or the application that is tested (aside from automated dynamic instrumentation). The only requirement is that a device be rooted.

4 User Interface Decomposition

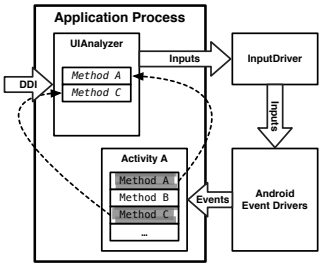
User interface decomposition is the first phase of CuriousDroid for a given activity, where the goal is to recover the hierarchy of user interface views contained in an activity. As stated in Section 3, CuriousDroid uses dynamic instrumentation to interpose on event callback invocations in order to extract this information. In the following, we describe the instrumentation framework used to accomplish this, and then outline how view hierarchies are recovered.

4.1 Dynamic Dalvik Instrumentation

CuriousDroid leverages the Dynamic Dalvik Instrumentation (DDI) framework [18] to instrument Android applications. This framework allows for in-memory injection of arbitrary code into application processes, enabling dynamic hooking and



(a) CuriousDroid overview. Application activity user interfaces are decomposed into view hierarchies. Individual views are labeled with the class of expected input using input inference. For each labeled view, an appropriate input is then generated. If a transition to a new activity is observed after submitting this input, the new activity is scheduled for analysis. This process iterates until a fixpoint is reached – i.e., no new activities are observed.



(b) Components comprising CuriousDroid. The UIAnalyzer is injected directly into the AUT using DDI, and hooks application event callbacks to interpose on UI-related operations in order to extract view hierarchies. The InputDriver resides in an external process that receives sequences of user interactions to drive the AUT.

Fig. 1: CuriousDroid overview and components.

interposition on both managed and native code, including transitions between application and Android framework code.

In particular, DDI is used to inject the aforementioned UIAnalyzer component into the process corresponding to the AUT. The UIAnalyzer consists of a native library that in turn executes the AUT within the process. Once the application's code has been loaded into memory, the DVM API is used to instrument specific application methods concerned with UI-related events. Figure 1b depicts this process.

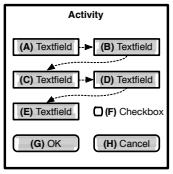
4.2 User Interface Analysis

For each activity, the UIAnalyzer decomposes the current view hierarchy. Starting from the root view (`PhoneWindowDecorView`) below which all other views in the activity are attached, all of its descendants are recursively explored using class introspection for identification and attribute extraction. Typically, the direct descendants of the root view are instances of one or more container views such as `LinearLayout`, `GridLayout`, or `RelativeLayout`. Each of these containers holds either further nested layouts or concrete views.

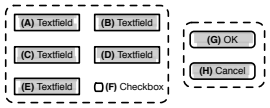
As the view hierarchy is explored, the UIAnalyzer records any *interactive* views that it discovers, such as editable text fields, buttons, spinners, radio buttons, and checkboxes, which we refer to as *widgets*. These *widgets* often contain attributes that indicate the expected types of inputs, and are recorded for later use as described in the next section.

5 Input Inference

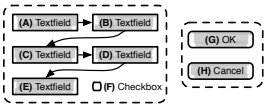
The second phase of CuriousDroid's user interface exploration is input inference, where the goal is to determine the type of interaction a widget expects. The point of input inference is to ensure that CuriousDroid drives the execution of the application in a way similar to how the developers would expect a human to



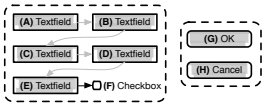
(a) The example activity layout as displayed to the user. Dashed lines indicate a `nextFocus` relationship between widgets.



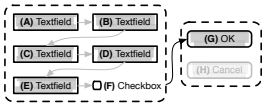
(b) The UIAnalyzer places widgets into two equivalence classes: buttons, and everything else.



(c) `nextFocus` relationships for (A–E) are overlaid onto the current ordering.



(d) Position-based heuristics indicate that the next link should be the checkbox (F).



(e) Label extraction identifies (G) as the correct button to exercise. At this point, a candidate ordering has been inferred.

Fig. 2: Inference of an ordering on user inputs for an example activity UI layout.

do so. The underlying assumption is that blindly exercising an application's UI, while perhaps useful for simple or widget-less activities, will not cover as much of the UI – and, therefore, application code. Likewise, attempting to exhaustively explore all possible targeted interactions within an activity can quickly become intractable for complicated activities.

To that end, one concrete aim of input inference is to not only identify the set of widgets that require input in order to trigger behavior or launch further activities, but also to tailor *meaningful* input for each identified widget. For example, instead of simply providing random text to a text field, CuriousDroid attempts to identify the class of input a field requires, such as an email address or phone number, and generates a realistic input drawn from the inferred class.

However, inferring expected classes of input for each widget is not sufficient to properly explore an activity. Indeed, the ordering of inputs is also important because a basic requirement of many widgets is that they are populated with a (well-formed) value before performing an action or launching a new activity. Therefore, CuriousDroid also needs to infer this partial ordering on all widgets in the current UI layout so that such constraints are satisfied.

5.1 Widget Orderings

The UIAnalyzer considers four widget attributes when determining an ordering of widgets to exercise in a UI layout: widget type, the `nextFocus` property (if present), widget screen position, and widget text labels. These attributes, taken together, allow the UIAnalyzer to construct a simple, yet accurate, model of how a user might interact with any given UI. Figure 2 presents an overview of the ordering inference process for an example activity UI layout.

The incorporation of widget type information into the ordering inference process is motivated by the fact that exercising certain widgets implies a transition

to another activity. The most straightforward concrete example of this is an OK or Cancel button that submits or dismisses a form, respectively. Since most or all of the other widgets must generally be populated prior to successful submission of a form (or to trigger behavior that changes the state of the current activity), it follows that this class of widget – in particular, OK buttons – must be exercised last. Therefore, the first step in the ordering inference is to group widgets into two classes: buttons, and everything else (2b).

The optional widget `nextFocus` property provides developers with a mechanism for encoding within UI layouts exactly the ordering that users are intended to follow when interacting with an activity. This manifests in the user experience as an automatic shift of focus from one widget to another when the Next button on the keyboard is pressed. The UIAnalyzer considers the presence of this property as ground truth of the intended interaction with an activity, and so the next step of the inference process is to incorporate this ordering information (2c).

The UIAnalyzer assumes that widgets are exercised in a top-to-bottom, left-to-right order, and uses screen coordinates of the remaining, unordered non-button widgets to heuristically include them into the current ordering (2d).

Finally, the terminal user input is selected from the button class of widgets. Here, the UIAnalyzer uses each button's label as an indicator of whether it is likely to produce some update to the activity's state and, potentially, produce a transition to another activity (2e).

5.2 Expected Input Classes

Given an inferred ordering of widgets to exercise, the next step is to decide *what* inputs the UIAnalyzer should provide. For this, three attributes are taken into account: widget hints, labels, and contextual information. Most developers add hints to editable text fields that indicate the type of input that is expected, such as an email address, phone number, postal code, or name.

In the absence of such information, label text is extracted from either the widget directly – e.g., placeholder text – or from label widgets directly adjacent to the widget in question. In our observations, these labels are almost as accurate as explicit hint properties. The labels are then matched against manually-compiled keyword lists to map them to a canonical class identifier. For instance, a Register label on a button would map to the OK class, and a Mobile label on a text field would map to the Phone class. The text contained in these keyword-lists is translated to several languages, including English, Russian, Korean, Japanese, and Chinese.

For those widgets that the UIAnalyzer is successfully able to identify a corresponding input class, an appropriate input is generated. The text is drawn from lists corresponding to a specific class of input – for example, in the case of editable text fields, the UIAnalyzer contains lists for names, addresses, phone numbers, passwords, and many more.

If the input inference process is unable to determine an input class for a widget, a random interaction will be supplied. In the case of one or more image buttons lacking any descriptive text, a single button is randomly chosen.

6 Input Generation

Given an inferred widget ordering and expected input classes for each widget, the next stage of the UI exploration for each activity is to actually drive the current UI. To that end, the UIAnalyzer communicates with the InputDriver, providing it with ordering and input class information. The InputDriver translates this information into concrete user input events, which it then injects directly to the Android event drivers. An overview of this is shown in Figure 1b.

6.1 Input Translation

Translating an interaction to a set of input commands first requires determining the type of interaction that is expected. Currently, CuriousDroid can generate both taps and swipes. To click a button, it is only necessary to inject one tap at the button's on-screen position. To enter text into a widget, the InputDriver takes the desired text as input and maps the location of each character to a position – or positions – on the virtual keyboard. Multiple positions could be required for characters that require multiple taps, such as capitals, numbers, and special characters.

The InputDriver defines a function called `genericPress` that takes as parameters the desired (x, y) coordinates of the tap, and the amount of time in microseconds to wait before initiating the tap, and returns each of the values needed to populate an event structure for the touchscreen event driver.

In addition to the function described above, the InputDriver also provides functions for the menu button, the back button, and a function that returns random events for fuzzing. After the formatted command string for all interactions has been completely generated, the command string is ready for injection into the AUT.

6.2 Input Injection

Input injection is achieved via two approaches: *event injection* and *random event generation*. Event injection writes the commands from the UIAnalyzer to the Android touchscreen event driver, while the random event generation function creates random taps and swipes to write to the event drivers. Similar to RERAN [12], the InputDriver can speed up or slow down the execution of an application; however, we choose to execute at a speed resembling human usage.

In the case that an activity does not contain (known) widgets in its layout, our interactions will not induce the execution of a new activity. It is therefore necessary to have a fallback mechanism that attempts to get an application out of a "stalled" execution state. We have implemented that system from within the InputDriver module.

In particular, if a preset period of time has elapsed and no observable action has occurred in response to an input sequence, the InputDriver initiates the random event generation process. The process assumes the application is in a stalled state and attempts to perturb it by sending random events to the system driver. If this fails to advance the execution of the application, it presses the Back button, reverting the state of the application to the previous activity.

7 Evaluation

In this section, we evaluate the efficacy of CuriousDroid as a driver for dynamic analysis. In particular, we compare the results of standard Andrubis [1, 14], a well-known dynamic analysis sandbox for Android applications, to the results of composing Andrubis with CuriousDroid, and show significant improvements in the analysis results.

7.1 Andrubis

Andrubis is a large-scale public malware analysis system for Android applications. It provides a comprehensive analysis report that includes results from static code analysis as well as runtime behavior using dynamic analysis in the QEMU emulator, on both the Dalvik VM and system level.

Static analysis of the application's manifest and certificate yields meta information such as requested permissions, services, broadcast receivers, activities, package name, SDK version, and information about the author. Furthermore, static code analysis extracts APIs calls and identifies the permissions actually used in the application's bytecode in contrast to the permissions required in the manifest.

During dynamic analysis, Andrubis monitors applications through an instrumented Dalvik VM. It records data leaks, filesystem activity, phone activity such as sending SMS and making phone calls, network activity, and the dynamic loading of DEX code as well as native code through JNI. In order to drive program execution, by default Andrubis utilizes the pseudorandom user interaction sequences generated by the Monkey.

In addition to the analysis report, Andrubis provides a *malice score* for each application [13]. Based on static and dynamic features learned from a set of known benign and malicious applications, Andrubis leverages an SVM-based classifier to assign scores on a scale from 0 to 10, with 0 being benign and 10 being malicious.

7.2 Experimental Setup

Our evaluation was performed over a data set of 51,571 randomly selected Android applications from five categories: applications that received a borderline classification from standard Andrubis, those that contain SMS-related code, those that perform dynamic code loading, those that perform native library loading, and those that interact with the network. Except for the first category, each of these represents a specific behavior that is potentially (and often) indicative of malware [28].

We found that out of the 51,571 applications tested, 12,699 drew no activities. In this case, CuriousDroid was not invoked and we were unable to collect results for those runs. Therefore, only the 38,872 remaining applications were considered in our evaluation.

7.3 Activity Coverage

A natural measure of CuriousDroid's effectiveness in exploring application user interfaces is in terms of activity coverage, as activities are the main container for UI layouts on Android. The list of activities for an application is extracted from its manifest by Andrubis during static analysis. In addition, CuriousDroid logs

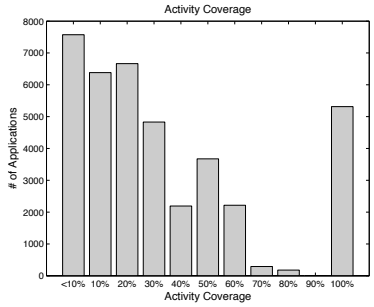


Fig. 3: Activity coverage by CuriousDroid for all applications.

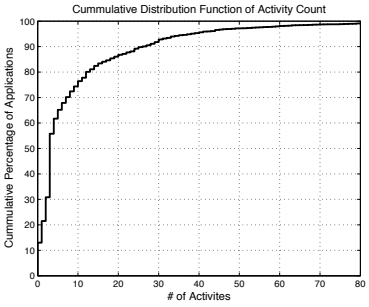


Fig. 4: Cumulative distribution function for application activity counts. The steep curve indicates the majority of applications contain fewer than 10 activities.

each activity it visits during its exploration. We compare these lists of activities to calculate our total activity coverage.

Figure 3 shows the activity coverage for all applications in the data set. In this figure, coverage is binned in 10-percent increments. We note, in Figure 4, that the majority of applications contain fewer than 10 activities, but as many as 284 have been found in a single application. Applications with such a large number of activities are guaranteed to produce low coverage numbers primarily due to the analysis time limit enforced by CuriousDroid.

7.4 Borderline Classification

To measure the impact of CuriousDroid on Andrubis' application classifier, we analyzed 8,827 applications from the data set that received a score on the interval $[4, 5]$ – i.e., a borderline score that is neither definitively benign nor malicious. As described above, Andrubis' classifier constructs a feature vector from a mix of static and dynamic features. CuriousDroid has no impact on the static analysis performed by Andrubis, so the static features used in our analysis are the same as in the original Andrubis analysis. Therefore, any change in an application's score can be attributed to a change in dynamic features observed during UI exploration. Figure 5 plots both the original scores assigned by standard Andrubis as well as the scores generated when incorporating CuriousDroid for this test set.

We note that CuriousDroid produces a significant increase in the quality of Andrubis' classifier as measured by score spread. There is an observable density of scores around 1.0, which demonstrates that the additional runtime behavior elicited by CuriousDroid was able to allow a relabeling from unknown to benign for many applications. Also observable are three smaller bands around 8.5, 8.75, and 10, which demonstrates that a (relatively) smaller group of applications could be reclassified from unknown to malicious due to CuriousDroid.

We plot the number of dynamic features used by Andrubis during classification in Figure 6. On average, more than nine additional dynamic features were used when incorporating CuriousDroid. The average number of dynamic features used by CuriousDroid increased to 30.8 from 21.6 with the Monkey. Additionally, the total number of features generated increased by an average of more than 27 features per application, all of which we can assume are dynamic. The average

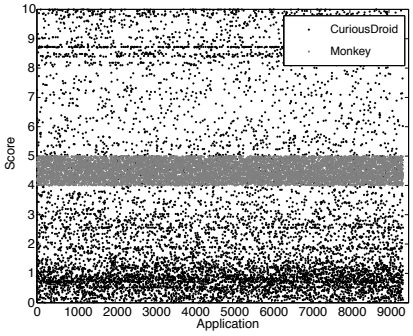


Fig. 5: Effect of CuriousDroid on borderline Andrubis scores. Scores from modified Andrubis are in black, and standard Andrubis in gray. Note the significantly greater spread of scores due to incorporating CuriousDroid during UI exploration.

total number of features used increased from 58.6 with the Monkey to 68.4 with CuriousDroid, in line with the increase in the number of dynamic features used.

7.5 Observed Dynamic Behaviors

The remaining four application categories described in the experimental setup refer to a specific behavior that static analysis indicated the application had the capability to perform. However, the applications we consider in each of these categories did not exhibit these behaviors during dynamic analysis using standard Andrubis. An overview of the results of composing Andrubis with CuriousDroid in this respect is shown in Table 1. In the following, we discuss each of the categories separately.

SMS. The SMS category is a set of applications chosen because they were statically determined to possess the capability to send or receive SMS messages. Not only do they request the `SEND_SMS` permission, but they are also found to have actual method calls that send or receive SMS messages. We chose this as a search parameter because it is possible for an application to request a permission that the developer never intends to use, especially in the case of code re-use. We found that CuriousDroid was able to trigger the sending of SMS messages in 440 of the 6,871 applications analyzed. Furthermore, many of the numbers to which messages were sent were short numbers, indicating a higher likelihood that these messages were sent to premium numbers. Such behavior is often indicative of malware.

Dynamic code loading. There are times when utilizing dynamic code loading is useful or necessary – e.g., when the primary application DEX file has more than 64K method references [7]. Andrubis is able to detect dynamic code loading during static analysis. The resulting test set consisted of 8,371 applications, in which CuriousDroid were able to trigger the loading of dynamic code in 358, representing a total of 4.28% of the set.

Native library loading. Android provides developers with an API for loading and running native code using JNI. This functionality is particularly important to developers who need direct access to the GPU or CPU for performance or power-saving purposes. However, malware can exploit the JNI interfaces to hide

Category	# Apps	# Triggered	% Triggered
SMS	6871	440	6.40%
Dynamic Code	8371	358	4.28%
Native Code	7669	1945	25.36%
Networking	7134	2650	37.15%

Table 1: Dynamic application behavior elicited due to CuriousDroid. By comparison, Andrubis used with the Monkey produced *no* behavior in each category.

certain behaviors, such as communicating with remote servers, installation of rootkits, or general obfuscation [28].

Therefore, we analyzed a set of 7,669 applications containing native libraries that were not loaded during dynamic analysis using standard Andrubis. CuriousDroid triggered the loading of these libraries in 1,945 applications, constituting 25.36% of the applications in the set.

Networking. The last test set was composed of applications that requested the `INTERNET` permission, commonly used in both benign and malicious applications. Network connectivity allows applications to access resources over the Internet. This could be as benign as checking account credentials or downloading legitimate content or advertisements. This permission, however, is also often used for nefarious behavior such as downloading dynamic code, drive-by downloads, or connecting to a command-and-control server for mobile bots.

Of the original 10,000 applications tested, 2,866 were found to have drawn zero activities. Therefore, we limited our analysis to the remaining 7,134. CuriousDroid triggered network traffic in 2,650 applications, representing an improvement of 37.15% of the total set.

7.6 Case Study

In the final experiment, we specifically investigated the 440 samples that only send SMS messages when executed by CuriousDroid. Using their MD5, we searched sites such as AndroTotal [17] to determine if a sample is known, determining that 20 samples were not previously known. 15 samples sent SMS messages to premium numbers, while five samples sent messages to regular phone numbers.

We further investigated one of the SMS samples in more detail. We selected the sample based on its change in malice score from 0.8743 to 8.6093 when driven using CuriousDroid. The application asks the user to accept an update of the program's database. If the user accepts, the application starts a larger download. During the download, the application sends five SMS messages to different short codes. CuriousDroid was able to trigger sending the SMS messages because it intelligently drove the UI by pressing the correct button. If the user does not accept the update download, the application immediately terminates without sending the SMS messages.

8 Discussion

Upon inspection of our initial test set, we discovered 12,699 applications where zero activities were drawn. We randomly chose a subset of applications to manually inspect in an emulator and found several reasons for this phenomenon.

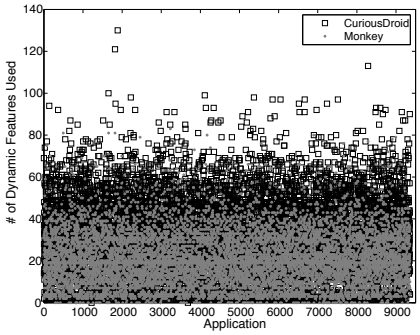


Fig. 6: Number of observed dynamic features for each application in the test set. Again, CuriousDroid elicits significantly more behavior than the Monkey.

The most obvious cause for zero-activity coverage is that some applications simply crash upon startup. Of these, many applications received a score of 9.0 or higher by Andrubis. We postulate that many malicious applications run their payload at startup, and have no intention of ever displaying this bootstrapping activity to the user. In one instance, we observed an application that crashed on startup, but on a second execution opened a dialog box requesting the user download and install an update from a third-party website.

On the other end of the spectrum, we noticed that most applications that achieved 100% coverage had very few activities. Malware is very likely packaged with simple applications, perhaps containing just one or two activities, although we see evidence that simply running an application containing malware is not necessarily enough to trigger the payload. We can infer from Section 7.5 that stimulating a UI with intelligent interactions is more likely to trigger malicious behavior.

When an application transitions from one activity to another, it calls the `startActivityForResult` family of methods which takes an intent as an argument. Applications are not limited to starting activities contained within their own application, and can additionally open activities belonging to other applications such as a browser or email application. CuriousDroid examines the intents passed to `startActivityForResult` and determines whether or not the target activity belongs to the current application. If it does not, it blocks the method call to ensure that the AUT retains focus. Doing so can have unforeseen consequences, sometimes resulting in an application crash. However, in most cases the application remains in the current activity until perturbed by the random event generator.

In a similar situation, when the Back button is pressed, the current activity's `finish` method is invoked. If the current activity is not the root of the activity stack, this is not a problem. However, in the event that the current activity is the root activity of the application, making a call to `finish` causes the application to exit. In an effort to keep the AUT in focus, we block all `finish` calls originating from the root activity. This can cause an application crash – which occurs more often in applications written for older SDK versions – when the root activity changes from one activity to another. When this happens, we have observed on

occasion that the call to the current root activity's `finish` method occurs before the new root activity has been created. If the new root activity has not yet been set, we block the call to the previous root activity's `finish` call, causing an application crash.

9 Related Work

Recent work in Android dynamic analysis has taken numerous, diverse approaches. A recent study [19] performed a comprehensive analysis of 16 dynamic analysis platforms. Some systems rely on a blend of static and dynamic analysis, while others only provide a mechanism for examining dynamic properties and do not include a way to execute applications. Some only rely on fuzz testing, or random interactions, to drive execution, while others attempt to exercise the UI in a more deterministic fashion. We consider several systems that employ these techniques and show how CuriousDroid distinguishes itself from these systems.

To avoid detection, malware can implement sandbox evasion. A recent effort studied the characteristics of different Android sandboxes and how these can be evaded [25]. Since CuriousDroid can be also deployed on a real device, it can be used to analyze evasive malware that attempts to fingerprint emulated environments.

Dynodroid [16] provides a system for interacting with UI widgets dynamically. The authors implement a mechanism that attempts to generate a sequence of intelligent UI interactions and system events by observing the UI's layout, composing and selecting a set of interactions, and executing those actions. Dynodroid leverages the Hierarchy Viewer [2], a tool packaged with the Android platform to infer a UI model during execution, to determine an activity's layout. We note that it was necessary to make changes to the SDK source code to enable this capability. Finally, and perhaps most importantly, Dynodroid requires that a tester has access to the source code of an application, as use of the Android instrumentation framework is necessary. In contrast, CuriousDroid can be used to test any APK file without source code since it dynamically instruments the application bytecode.

SmartDroid [27] uses a hybrid approach, leveraging both static and dynamic analysis, to discover and exercise UI trigger conditions. Using static analysis, SmartDroid constructs a desired activity switch path that leads to the sensitive API calls it wishes to exercise. During dynamic analysis, SmartDroid traverses the view tree of an activity and triggers the event listeners for each UI element. If the event listener invokes the start of a new activity, SmartDroid determines if that activity is on the activity switch path. If not, it blocks the call to that activity and continues to traverse the current activity's view tree until the correct element is stimulated. SmartDroid requires not only modifications to the SDK, but a modified emulator as well. In addition, relying on static analysis to reveal sensitive behaviors will exclude calls to dynamically loaded code or native libraries.

Swifthand [6] is an automated GUI testing tool for Android that leverages machine learning techniques to create a model of an application which it can leverage to generate user inputs in an attempt to visit unexplored areas of the application. Swifthand requires modifications of applications through static in-

strumentation of the binary and it is unclear to us whether or not this process is entirely automated. It makes no attempt to derive context from an application based on the UI, as a human would. The average runtime required by Swifthand tests was three hours per application, making it unsuited for large-scale testing. Finally, only 10 applications were included in the test set. Such as small set does not provide adequate insight into the efficacy of the tool at scale.

A3E [5] provides another system for UI exploration of Android applications with two separate approaches: "Targeted Exploration" which uses a CFG generated during static analysis to develop a strategy for exploration by targeting specific activities, and "Depth-first Exploration" which attempts to mimic user interactions to drive execution in a more systematic, albeit slower, way. A3E is not suitable for large-scale testing due to the long testing time required for each application. A3E was tested on only 25 applications, and has an average runtime of 87 minutes per application for targeted exploration method and 104 minutes per application for the depth-first exploration.

AppsPlayground [20], in addition to acting as a malware detection system, employs a technique for automatically executing Android applications. Similar to CuriousDroid, AppsPlayground attempts to determine context from the UI in order to more intelligently direct an applications execution. This includes inserting relevant text into text boxes, as well as clicking the most appropriate buttons. We employ a similar technique to AppsPlayground, using hints and keywords from UI elements to determine context.

We note that AppsPlayground requires modification of the SDK and OS and can only be used with an emulator. CuriousDroid has been tested on physical devices as low as API level 4, up to API level 16, and requires no modifications to the OS or SDK. We leverage the techniques used in RERAN [12] to pass interactions to the device, such as taps, swipes, and hardware button pushes. This means that when text is entered into a field, it is passed in as a series of actual taps to keys on the virtual keyboard. In contrast, AppsPlayground uses a modified version of the MonkeyRunner to pass interactions to the application.

AppsPlayground was tested on just under 4,000 applications, only three of which were known malware samples. Finally, the authors of AppsPlayground provided no measurements of the end-to-end time required to test an application. Therefore we are unable to determine the suitability of AppsPlayground as a system for large-scale measurement.

10 Conclusion

In this paper, we introduced CuriousDroid, an automated system to drive Android applications in an *intelligent*, *user-like* manner. Our system is generic and can be deployed in any Android sandbox and even on real Android devices. To evaluate our system, we integrated it into the well-known Andrubis sandbox. We evaluated CuriousDroid using 38,872 applications that we randomly selected from different categories. The results of our evaluation demonstrate that our system was able to elicit significantly more runtime behavior over the standard combination of Andrubis and random input generated by the Monkey, improving Andrubis' ability to categorize previously borderline applications as either definitively benign or malicious. This capability in particular suggests that Cu-

riousDroid would prove very helpful for malware triage, greatly reducing the numbers of applications that would require manual analysis to classify.

Acknowledgements. This material is based upon work supported by the National Science Foundation under Grant No. CNS-1409738. The research leading to these results has received funding from the FFG – Austrian Research Promotion under grant COMET K1 and has been carried out within the scope of u'smile, the Josef Ressel Center for User-Friendly Secure Mobile Environments.

References

1. Andrubis. http://anubis.iseclab.org/
2. Hierarchy Viewer. http://developer.android.com/tools/help/hierarchy-viewer.html
3. MonkeyRunner. http://developer.android.com/tools/help/monkeyrunner_concepts.html
4. UI/Application Exerciser Monkey. http://developer.android.com/tools/help/monkey.html
5. Azim, T., Neamtiu, I.: Targeted and Depth-first Exploration for Systematic Testing of Android Apps. In: International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA) (2013)
6. Choi, W., Necula, G., Sen, K.: Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning. In: International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA) (2013)
7. Chung, F.: Android Developers Blog. http://android-developers.blogspot.com/2011/07/custom-class-loading-in-dalvik.html (2011), [Online; accessed May 5, 2014]
8. Egele, M., Brumley, D., Fratantonio, Y., Kruegel, C.: An Empirical Study of Cryptographic Misuse in Android Applications. In: ACM Conference on Computer and Communications Security (CCS) (2013)
9. Enck, W., Ongtang, M., McDaniel, P.D., et al.: Understanding Android Security. IEEE Security and Privacy (Oakland) (2009)
10. Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.: Android Permissions Demystified. In: ACM Conference on Computer and Communications Security (CCS) (2011)
11. Felt, A.P., Finifter, M., Chin, E., Hanna, S., Wagner, D.: A Survey of Mobile Malware in the Wild. In: ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM) (2011)
12. Gomez, L., Neamtiu, I., Azim, T., Millstein, T.: RERAN: Timing- and Touch-Sensitive Record and Replay for Android. In: International Conference on Software Engineering (ICSE) (2013)
13. Lindorfer, M., Neugschwandtner, M., Platzer, C.: Marvin: Efficient and Comprehensive Mobile App Classification Through Static and Dynamic Analysis. In: Annual International Computers, Software & Applications Conference (COMPSAC) (2015)
14. Lindorfer, M., Neugschwandtner, M., Weichselbaum, L., Fratantonio, Y., van der Veen, V., Platzer, C.: Andrubis - 1,000,000 Apps Later: A View on Current Android Malware Behaviors. In: Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS) (2014)
15. Liu, B., Nath, S., Govindan, R., Liu, J.: DECAF: Detecting and Characterizing Ad Fraud in Mobile Apps. In: USENIX Conference on Networked Systems Design and Implementation (NSDI) (2014)

16. MacHiry, A., Tahiliani, R., Naik, M.: Dynodroid: An Input Generation System for Android Apps. In: Foundations of Software Engineering (2013)
17. Maggi, F., Valdi, A., Zanero, S.: AndroTotal: A Flexible, Scalable Toolbox and Service for Testing Mobile Malware Detectors. In: ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM) (2013)
18. Mulliner, C.: Dynamic Dalvik Intrumentation (DDI). `https://github.com/crmulliner/ddi`
19. Neuner, S., Van der Veen, V., Lindorfer, M., Huber, M., Merzdovnik, G., Mulazzani, M., Weippl, E.R.: Enter Sandbox: Android Sandbox Comparison. In: IEEE Mobile Security Technologies Workshop (MoST) (2014)
20. Rastogi, V., Chen, Y., Enck, W.: AppsPlayground: Automatic Security Analysis of Smartphone Applications. In: Conference on Data and Application Security and Privacy (CODASPY) (2013)
21. Reina, A., Fattori, A., Cavallaro, L.: A System Call-Centric Analysis and Stimulation Technique to Automatically Reconstruct Android Malware. In: European Workshop on Systems Security (EuroSec) (2013)
22. Smith, A.: Americans and Mobile Computing: Key Trends and Consumer Research. `http://www.slideshare.net/PewInternet/americans-and-mobile-computing-key-trends-in-consumer-research` (2011), [Online; accessed May 7, 2014]
23. Spreitzenbarth, M., Freiling, F., Echtler, F., Schreck, T., Hoffmann, J.: Mobile-Sandbox: Having a Deeper Look into Android Applications. In: Symposium on Applied Computing (SAC) (2013)
24. Strategy Analytics: Android Captures Record 85 Percent Share of Global Smartphone Shipments in Q2 2014. `http://www.prnewswire.com/news-releases/strategy-analytics-android-captures-record-85-percent-share-of-global-smartphone-shipments-in-q2-2014-269301171.html` (2014)
25. Vidas, T., Christin, N.: Evading Android Runtime Analysis via Sandbox Detection. In: ACM Symposium on Information, Computer and Communications Security (ASIACCS) (2014)
26. Yan, L.K., Yin, H.: DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In: USENIX Security Symposium (2012)
27. Zheng, C., Zhu, S., Dai, S., Gu, G., Gong, X., Han, X., Zou, W.: SmartDroid: An Automatic System for Revealing UI-based Trigger Conditions in Android Applications. In: ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM) (2012)
28. Zhou, Y., Jiang, X.: Dissecting Android Malware: Characterization and Evolution. In: IEEE Symposium on Security and Privacy (Oakland) (2012)